Neural Machine Translation Architectures and Applications

Alexandre Bérard

École doctorale n° 72 : Sciences Pour l'Ingénieur

Thèse de Doctorat

préparée au sein de CRIStAL, Inria Lille et
du Laboratoire d'Informatique de Grenoble

pour obtenir le grade de docteur délivré par

l'Université de Lille

Spécialité : **Informatique**

présentée par

Alexandre Bérard

le 15 juin 2018

Neural Machine Translation Architectures and Applications

Directeur : **Olivier Pietquin**
Co-directeur : **Laurent Besacier**

Jury

Philippe Langlais,	Professeur à l'Université de Montréal	Rapporteur
Béatrice Daille,	Professeur à l'Université de Nantes	Rapporteur
François Yvon,	Professeur à l'Université Paris Sud	Examinateur
Pascale Sébillot,	Professeur à l'INSA de Rennes	Examinateur
Marc Tommasi,	Professeur à l'Université de Lille	Examinateur
Olivier Pietquin,	Professeur à l'Université de Lille	Directeur
Laurent Besacier,	Professeur à l'Université Grenoble Alpes	Co-directeur

"A Lannister always pays his debts."

Abstract

This thesis is centered on two main objectives: research replication and adaptation of Neural Machine Translation techniques to new tasks. Our efforts towards research replication have led to the production of two resources: MultiVec, a framework that facilitates the use of several techniques related to word embeddings (Word2vec, Bivec and Paragraph Vector); and a framework for Neural Machine Translation that implements several architectures and can be used for regular MT, Automatic Post-Editing, and Speech Recognition or Translation. These two resources are publicly available and now extensively used by the research community.

We extend our NMT framework to work on three related tasks: Machine Translation (MT), Automatic Speech Translation (AST) and Automatic Post-Editing (APE). For the machine translation task, we replicate pioneer neural-based work, and do a case study on TED talks where we advance the state-of-the-art. Automatic speech translation consists in translating speech from one language to text in another language. In this thesis, we focus on the unexplored problem of *end-to-end* speech translation, which does not use an intermediate source-language text transcription. We propose the first model for end-to-end AST and apply it on two benchmarks: translation of audiobooks and of basic travel expressions. Our final task is automatic post-editing, which consists in automatically correcting the outputs of an MT system in a black-box scenario, by training on data that was produced by human post-editors. We replicate and extend published results on the WMT 2016 and 2017 tasks, and propose new neural architectures for low-resource automatic post-editing.

Keywords neural machine translation, automatic post-editing, automatic speech translation, sequence to sequence learning, encoder-decoder, attention-based models, deep learning, neural networks, word embeddings

Résumé

Cette thèse est centrée sur deux principaux objectifs : l'adaptation de techniques de traduction neuronale à de nouvelles tâches, et la reproduction de travaux de recherche existants. Nos efforts pour la reproductibilité ont résulté en la création de deux ressources : MultiVec, un outil permettant l'utilisation de plusieurs techniques liées au word embeddings (Word2vec, Bivec et Paragraph Vector); ainsi qu'un outil pour la traduction neuronale, qui implémente plusieurs architectures permettant la traduction automatique, la post-édition automatique, ou la reconnaissance vocale ou traduction de la parole. Ces deux ressources sont disponibles librement et utilisées de manière intensive par la communauté scientifique.

Nous modifions notre outil de traduction neuronale afin de travailler sur plusieurs tâches : la Traduction Automatique (TA), Traduction Automatique de la Parole, et la Post-Édition Automatique. Pour la tâche de traduction automatique, nous reproduisons des travaux fondateurs basés sur les réseaux de neurones, et effectuons une étude de cas sur des *TED Talks* (sous-titres de conférences), où nous faisons progresser l'état de l'art. La traduction de la parole consiste à traduire la parole dans une langue vers le texte dans une autre langue. Dans cette thèse, nous nous concentrons sur le problème inexploré de traduction de la parole *end-to-end*, qui ne passe pas par une transcription textuelle intermédiaire dans la langue source. Nous proposons le premier modèle de traduction de la parole *end-to-end*, et l'évaluons sur deux problèmes : la traduction de livres audio, et la traduction d'expressions de voyage basiques. Notre tâche finale est la post-édition automatique, qui consiste à corriger de manière automatique les sorties d'un système de traduction dans un scénario "boîte noire", en apprenant à partir de données produites par des post-éditeurs humains. Nous reproduisons et étendons des résultats publiés dans le cadre des tâches de WMT 2016 et 2017, et proposons de nouvelles architectures neuronales pour la post-édition automatique dans un scénario avec peu de ressources.

Mots-clés traduction automatique, réseaux de neurones artificiels, apprentissage profond, traduction automatique de la parole, post-édition automatique, modèles d'attention, plongements de mots.

Remerciements

Je tiens à remercier Laurent Besacier et Olivier Pietquin, qui ont co-dirigé cette thèse. Ils m'ont donné des conditions de travail exceptionnelles, dans des équipes fantastiques. Ils étaient toujours présents, aussi bien d'un point de vue moral que scientifique, tout en me laissant une liberté quasi-totale sur mon travail de recherche.

Je remercie Philippe Langlais et Béatrice Daille pour leur lecture attentive et pour leurs critiques précieuses sur mon travail de thèse.
Je remercie également Marc Tommasi, Pascale Sébillot et François Yvon d'avoir participé au jury et assisté à la soutenance, et pour leurs nombreuses questions et remarques constructives. Une mention spéciale à François Yvon, qui a également lu cette thèse et produit un rapport particulièrement utile et exhaustif.

Un grand merci à mes collègues et amis de l'équipe SequeL à Lille, qui ont contribué à une superbe ambiance, au sein d'une équipe absolument excellente d'un point de vue scientifique. Je remercie tout particulièrement les doctorants: Julien, Jean-Bastien, Florian, Ronan, Daniele, Merwan, Hadrien, Frédéric, Marta et Tomáš.

Je remercie mes collègues et amis du Laboratoire d'Informatique de Grenoble, qui m'ont accueilli en leur sein, sans absolument aucune discrimination d'équipe!

Un grand merci à mes parents, qui m'ont toujours encouragé à poursuivre mes études, sans jamais m'y contraindre.

Et enfin, je remercie surtout Carole pour son soutien moral sans faille.

Contents

Introduction

In the recent years, Natural Language Processing has witnessed two major revolutions. First, thanks to their "Word2vec" techniques, Mikolov et al. (2013a) democratized the use of word embeddings, universal vector representations of words that are automatically estimated from text data. This drastically changed NLP by relaxing the need for careful feature engineering and expensive linguistic knowledge.
Then, Bahdanau et al. (2015) and Sutskever et al. (2014) introduced Deep Neural Networks for Machine Translation (and other sequential tasks). These techniques now achieve considerably better results in high-resource Machine Translation than the former Phrase-Based MT standard. Many other fields, related to Machine Translation (e.g., Speech Recognition, Automatic Post-Editing, Automatic Summarization, etc.) have started to move towards these new methods, which show promise but also pose many new challenges.

This thesis is centered on two main objectives: adaptation of Neural MT techniques to new tasks and research replication. Our efforts towards research replication have led to the production of two resources: **MultiVec**, a framework that facilitates the use of several techniques related to word embeddings (Word2vec, Bivec and Paragraph Vector); and a framework for **Neural Machine Translation** that implements several architectures and can be used for regular MT, Automatic Post-Editing, and Speech Recognition or Translation. These two resources are publicly available and now extensively used by the research community.

We use (and extend) our NMT framework to tackle three related tasks: Machine Translation, where we focus on replicating published results; **Automatic Speech Translation**, for which we present the first end-to-end model; and **Automatic Post-Editing**, where we replicate existing work and propose new architectures.

We structure this introduction around our four contributions (in bold), each time detailing the context and giving a quick overview of the existing methods, and then of the contributions themselves.

Neural Machine Translation

Machine Translation (MT) consists in automatically translating text from one language to another. Generally, the MT task is limited to translating single sentences, independently of the surrounding text. The progress in the field is measured thanks to evaluation campaigns (e.g., WMT), which distribute training data, and measure the translation quality of the participating systems with a careful evaluation methodology: e.g., automatic evaluation on a held-out test set with metrics like BLEU (Papineni et al. 2002).

Context Until recently, the state-of-the-art technique for general domain Machine Translation was Statistical Machine Translation (SMT), and more precisely Phrase-Based Machine Translation (PBMT) (Koehn et al. 2003). This is a statistical model which is trained on thousands or millions of translated sentence pairs (produced by humans). PBMT actually combines several statistical models in a complex pipeline. A language model, estimated on large amounts of target-language data, enforces fluency in the target language. A large probability table is also built that assigns probability scores to pairs of words or phrases. This phrase table is built by combining word-alignments of the training corpus produced by the IBM Models (Brown et al. 1993). Several other features can be included that penalize too long or too short outputs, or that discourage long-distance reordering of words compared to the input. All these features are combined in a log-linear probability model, whose weights are tuned using yet another technique (Och 2003). Phrase-based Machine Translation was the standard until recently (and still is in some tasks), in part thanks to the availability of high-quality open-source implementations of the aforementioned techniques (Koehn 2010; Och et al. 2003).

Sutskever et al. (2014) proposed a new, radically different technique for Machine Translation, based on Deep Neural Networks (LeCun et al. 2015; Schmidhuber 2015). This "sequence to sequence" model encodes the source sequence into a fixed-size representation using a recurrent neural network (the encoder). Then, another recurrent neural network (the decoder) generates a new sequence in the target language, conditioned on this representation. Alternatively, the decoder can use a different representation of the input sequence at each time step, thanks to an attention mechanism (Bahdanau et al. 2015). This latter technique helps the model handle longer sequences by letting the decoder look at any symbol in the input sequence.

The main particularity of these models is that they are trained end-to-end, with parallel data only. Instead of having many components, each trained on a specific task (with their own training objective), which are loosely combined at decoding time, we train a single model to maximize a single translation objective. Except for the data pre-processing / post-processing pipeline (tokenization mostly) and the evaluation metrics, none of the components of the former SMT approach are needed anymore.

Another major difference is the flexibility of these approaches. PBMT does a strong assumption: each word or phrase is translated as one or several word or phrases, generally in the same vicinity. The only assumption that NMT does is that words are generated from left to right, conditioned on the previously generated words and on a "representation" of the input sequence. This opens many possibilities, such as multi-source translation (Zoph et al. 2016a), multi-task training (Luong et al. 2016), multilingual translation (Johnson et al. 2016), ensemble decoding (Sutskever et al. 2014), or character-level translation (Lee et al. 2016).

Neural Machine Translation has since then supplanted PBMT in the WMT translation shared task (Bojar et al. 2016; Sennrich et al. 2016b). It has been deployed in production at Google (Wu et al. 2016) and SYSTRAN (Crego et al. 2016), and now boasts almost human-level performance in certain high-resource tasks (Hassan et al. 2018; Wu et al. 2016).

A state of the art of SMT and NMT is given in **Section 1.1**. A more detailed account of neural networks and sequence to sequence models is given in **Chapter 2** (neural networks) and **Chapter 3** (sequential models).

Challenges Even though Machine Translation has achieved considerable progress thanks to these breakthroughs, there still exist a number of open problems. Koehn et al. (2017) shed light on a specific problem of neural-based approaches: the need for large amounts of data. Not all

of MT is about general domain translation in high resources settings. PBMT is still the state-of-the-art technique in low-resource scenarios. Another problem that they expose is the lack of interpretability of these models compared to PBMT (where an output word can be traced back to the exact source words that generated it). A problem that we address is the (lack of) reproducibility of research results, and the availability of high-quality implementations of NMT techniques for the research community. Reproducibility is a problem in NMT (and Deep Learning in general), as the papers often lack technical details, and a single wrong hyperparameter value can give widely different results. The official implementations (if any) of the techniques described in the literature come in a large variety of programming languages and Deep Learning frameworks (Caffe, CNTK, DyNet, Keras, (Py)Torch, TensorFlow, Theano, etc.) The technical details are often obfuscated, the implementations do not always work as advertised, and trained models are rarely available. Models sometimes take days or even weeks to train, which hinders reproducibility (especially in the academia, where the compute resources are lower than in the industry).

Contributions Our contributions to NMT are the following: we implement several techniques from the literature (Bahdanau et al. 2015; Jean et al. 2015a; Luong et al. 2015b) in a single framework,[1] based on TensorFlow (Abadi et al. 2015). This framework facilitates research replication and experimentation, as it proposes a full MT pipeline: pre-processing, training (single-task, multi-task), decoding (beam search, ensemble) and evaluation. The hyperparameters are entirely configurable thanks to configuration files and command-line parameters. This helps experimenting with different model architectures. We replicate pioneer work on English to French translation (Bahdanau et al. 2015; Jean et al. 2015a) and distribute the trained models.

Finally, we perform extensive experiments (with our framework) in lower-resource scenarios: German to English translation of TED talks, where we beat the previous SOTA results (P.-S. Huang et al. 2018); and French to English translation on the tiny BTEC corpus. Our NMT framework is described in **Section 4.2**, and our experiments are detailed in **Section 4.3**. Our results on BTEC are presented in **Chapter 5** as part of our work on Speech Translation.

Speech Translation

We also worked on two tasks that are related to Machine Translation: Automatic Speech Translation (AST), and Automatic Post-Editing (APE). Even though these two tasks have been explored in the past, they were still relatively untouched by the NMT revolution. Automatic Speech Translation consists in translating segments of speech from one language, either to text or to speech in another language. In this thesis, we are focused on speech-to-text translation.

Context Current methods for Automatic Speech Translation couple two main modules: an Automatic Speech Recognition system (ASR), and a Machine Translation system (MT) (Kumar et al. 2015; Post et al. 2013). Optionally, if one wants to produce speech in the target language, a third module of Text-to-Speech (TTS) can be plugged in. Commercial products like Skype Translator already include this kind of technology (for real-time translation of conference calls).

We explore a new kind of approach: End-to-End Speech Translation, where a single encoder-decoder model is trained to read source-language speech and to output target-language text. This

[1] http://github.com/eske/seq2seq

is possible by adapting the attention-based models that have become popular in ASR (Bahdanau et al. 2016; Chan et al. 2016; Chorowski et al. 2015). The kind of data needed for training an end-to-end AST model is different from the standard approach. It consists in pairs of speech segments in one language, with their translation in the target language.

There are several advantages to doing end-to-end AST. This could drastically change the way data for speech translation is collected, especially in very low-resource scenarios. For instance, in the project DARPA TRANSTAC (speech translation from spoken Arabic dialects), a large effort was devoted to the collection of speech transcripts. A prerequisite to obtain transcripts was often a detailed transcription guide for languages with little standardized spelling. If end-to-end speech-to-text translation obtains satisfactory results, we might consider collecting data by asking bilingual speakers to directly translate target-language samples in their mother tongue. This would be applicable to any unwritten language.

Another advantage is the potential for more compact and faster systems. We also hope that this might increase accuracy, by removing the accumulated errors of chaining two systems. However, in high-resource settings, this kind of approach is likely to be inferior to the standard cascaded approach, whose ASR and MT systems can be trained on larger amounts of data. We expect that techniques like Multi-Task training and Pre-training (the encoder and decoder can be trained on different tasks) will ultimately make end-to-end speech translation competitive, even in this scenario.

Contributions First, we design and implement an attention-based model for AST, inspired from the "Listen, Attend and Spell" (LAS) model for ASR (Chan et al. 2016). As this is similar to standard NMT, we can conveniently implement and experiment with this model in our framework. As a proof of concept in (Bérard et al. 2016a), we evaluate this approach on a synthetic dataset. We build this dataset from French-English BTEC (Basic Travel Expression Corpus), using a high-quality commercial TTS (Voxygen) to generate speech from the French segments. To the best of our knowledge, this is the very first end-to-end speech translation model in the literature.[2]

Our second contribution is an improvement of this model, following some design choices from Weiss et al. (2017), a follow-up work (by Google) of our previous contribution. They train an end-to-end AST model on the Fisher-CALLHOME corpus of Spanish-English telephone conversations (Post et al. 2013).[3] We train our new model on a public corpus of audiobooks, which was built by Kocabiyikoglu et al. (2018). This corpus is the result of an automatic alignment between the LibriSpeech corpus for ASR (containing audiobooks and their transcriptions), and Project Gutenberg (a large library of public domain e-books). We obtain promising results on this challenging "Augmented LibriSpeech" corpus, and establish the first baselines for AST, MT and ASR on this new corpus (that we invite others to challenge). We also extend our experiments on the synthetic BTEC corpus. These contributions are presented in **Chapter 5**.

[2] Another work, by Duong et al. (2016), precedes us, but they either take phonemes as input (and not raw speech), or solve an alignment task (not translation).

[3] They obtain very promising results, but they do not release their implementation, and the corpus that they use is not freely available.

Automatic Post-Editing

Another task that we tackle is Automatic Post-Editing (APE). The goal of this task is to automatically improve the outputs of a Machine Translation system in a black-box scenario. This means that we do not have access (or even knowledge) to the inner workings of this system, but only to its outputs.

A possible use case is when a translation agency wants to correct the systematic errors of a commercial MT system on which they do not have any control (e.g., Google Translate), or when they want to improve its translation quality on a specific domain (e.g., pharmaceutical domain). The training data consists in examples of outputs by this system, which have been manually post-edited by professional translators. In addition to the translation hypothesis and its post-edited version, we also have access to the source-language segment that generated this translation.

The motivation is that human post-editing is now a common practice in translation agencies. Machine Translation is now good enough, that it is more profitable for human translators to post-edit a translation hypothesis, rather than translating the source sentence manually from scratch. Post-editing reduces translation time and even increases the quality of the translations (Green et al. 2013). As human translators generate post-editing data, we should be able to use this data to facilitate their subsequent work. We can do so by training an automatic post-editing system with this data, which can be used as a first pass post-editing step for future segments. A state of the art of Automatic Post-Editing is presented in **Section 1.2**.

In the past three years, the Workshop on Machine Translation (WMT) has hosted a shared APE task (Bojar et al. 2017, 2016, 2015). They provide a post-editing training set, containing triples of source segments, translation hypotheses by an unknown MT system, and post-editing ground truths produced by a human translator. Research teams are invited to submit systems to the task, which are then compared to each other by an evaluation on a held-out test set: with automatic metrics like TER (Snover et al. 2006), and manual evaluation by humans.

Context Conventional methods for Automatic Post-Editing treat this problem as a *Machine Translation* problem, where the source is the translation hypothesis, and the target is its post-editing reference. Until recently, APE solutions involved a Statistical Machine Translation (SMT/PBMT, e.g., Moses), which is trained on monolingual post-editing data. Simard et al. (2007) were the first to propose to use SMT to post-edit the outputs of a rule-based system (RBMT). Later on, Béchara et al. (2011) proposed a way to also make use of the source-language sentence. However, their technique (which concatenates MT words with aligned source words) results in a very large and sparse vocabulary, and the results are not very convincing. Statistical Post-Editing (SPE) has achieved limited success and is effective in the most favorable conditions: RBMT post-editing (Simard et al. 2007), or domain adaptation (Potet et al. 2012a).

With the advent of NMT methods, the field has started to shift towards Neural Post-Editing. Junczys-Dowmunt et al. (2016a) combine two NMT models in an ensemble: a monolingual post-editing model, and a translation model (that takes advantage of the source-language sequence). By training these models on a large synthetic corpus that they built, they obtained exceptional results on the 2016 shared APE task (Bojar et al. 2016). Libovický et al. (2016) train an NMT model that predicts edit operations (insertions, deletions and keep) instead of words. This model beats the MT and SPE baseline on the 2016 APE task, even though it uses only real post-editing data. Then, Junczys-Dowmunt et al. (2017a) propose a multi-source Neural APE model, which reads both the MT sequence and the source sequence (with two encoders). This

model, also trained with large amounts of synthetic data, obtains even more impressive results. Variants of this model (Chatterjee et al. 2017; Junczys-Dowmunt et al. 2017b) obtained the best scores in the 2017 shared APE task (Bojar et al. 2017).

Contributions We implement two techniques from the state of the art in our NMT framework: the large multi-source model from (Junczys-Dowmunt et al. 2017a), and the op-based model from (Libovický et al. 2016). We replicate and analyze their results on the 2016 and 2017 APE tasks. We also evaluate these systems in different settings, with access to varying amounts of training data, from 12k segments of real PE data, up to more than 4M segments of synthetic data. We show that (Junczys-Dowmunt et al. 2017a)'s method also achieves positive results in realistic APE conditions (with smaller amounts of data).

Finally, we extend the op-based model from (Libovický et al. 2016), by proposing model architectures that are better suited for post-editing: a task-specific attention mechanism, and a better way (than multi-source APE) to integrate the source sequence. We complete a thorough evaluation, and observe that these op-based techniques are worse than (Junczys-Dowmunt et al. 2017a)'s in medium-to-high resource settings, but offer a better alternative in low-resource settings. These contributions are presented in **Chapter 6**.

MultiVec

Mikolov et al. (2013a) proposed a set of efficient neural network models for computing vector representations of words. While similar techniques have existed for a while (Y. Bengio et al. 2003; Collobert et al. 2011), a number of tricks (plus the fact that they are linear models) made it possible for these models to be trained on extremely large amounts of data. Mikolov et al. (2013a) released the source code of their "Word2vec" toolkit, which probably facilitated the adoption of these techniques by the community.

Word embeddings are useful in many NLP classification tasks (Collobert et al. 2011). Their main advantages are that they are non-task dependent (they encode a lot of syntactic and semantic information), and they are trained in an unsupervised way (from monolingual data). This means that embeddings can be trained on large amounts of data and distributed for use in various downstream NLP tasks (e.g., sentiment analysis), where smaller amounts of supervised data are available. This can be understood as powerful and effortless feature engineering.

Artetxe et al. (2017), Gouws et al. (2015), Luong et al. (2015a), Mikolov et al. (2013b), Zhang et al. (2014), and Zou et al. (2013) propose techniques for computing multilingual word embeddings, i.e., embeddings in several languages that share the same vector space. They are useful for Machine Translation, and other multilingual tasks (e.g., crosslingual classification). Zhang et al. (2014) and Zou et al. (2013) propose to use bilingual embeddings as additional features for Phrase-based Machine Translation. Artetxe et al. (2018) and Lample et al. (2017) propose techniques to do unsupervised Neural Machine Translation (i.e., MT without parallel data), using unsupervised bilingual embeddings. **Section 2.2** gives a detailed overview of the state of the art on word embeddings and sequence embeddings.

Contributions We propose MultiVec,[4] a framework which implements several techniques from the literature for computing distributed representations of words or sequences of words.

[4] http://github.com/eske/multivec

It allows the user to train word embeddings with the CBOW and Skip-Gram models (Mikolov et al. 2013a), sentence embeddings with the Paragraph Vector algorithm (Q. V. Le et al. 2014), and bilingual embeddings with Bivec (Luong et al. 2015a).

The goal of MultiVec is to regroup multiple techniques from the state of the art in a single toolkit, to facilitate their use and research replication.[5] The existing implementations of the concerned techniques can be obfuscated (e.g., the C code of the original Word2vec), lack documentation, or are not as fast as they could be. MultiVec is written in C++, which makes it more modular and easier to extend than the original Word2vec. It is also as fast as Word2vec, and better documented. Models can be trained thanks to an extensive command-line interface. We also propose a Cython wrapper, which allows the user to train and make use of existing models from Python.

We validate MultiVec on a series of benchmarks: crosslingual document classification, sentiment analysis, and analogical reasoning; and distribute the source code for these benchmarks. MultiVec is publicly available on GitHub, and was promulgated in a publication to LREC (Bérard et al. 2016a). It has been used in multiple research projects, including: Servan et al. (2016)'s work on MT evaluation metrics, work on quality estimation (N.-T. Le et al. 2016), and crosslingual plagiarism detection (Ferrero et al. 2017). The MultiVec toolkit and our validation experiments are presented in **Section 4.1**.

Overview

This thesis is divided in two parts of three chapters each: **Part I** gives the state of the art, and **Part II** describes our contributions. An appendix also gives additional (optional) reading material, in particular regarding our toolkits.

Chapter 1 gives an overview of Statistical Machine Translation and Neural Machine Translation (Section 1.1), and of the state of the art in Automatic Post-Editing (Section 1.2). **Chapter 2** gives a broad description of neural networks and how to train them (Section 2.1), and word embeddings (Section 2.2). **Chapter 3** describes recurrent neural networks (Section 3.1), and sequence to sequence models for machine translation (Sections 3.2, 3.3, and 3.4).

Chapter 4 presents the MultiVec toolkit (Section 4.1), and our NMT framework (Section 4.2), as well as replication experiments that we performed on Machine Translation (Section 4.3). **Chapter 5** details our work on End-to-End Automatic Speech Translation. Section 5.1 presents our proof-of-concept model for Automatic Speech Translation on a synthetic dataset. Section 5.2 describes the Augmented LibriSpeech corpus of audiobooks for Speech Translation. Section 5.3 extends our experiments to this corpus. Our work on Neural Post-Editing is presented in **Chapter 6**. Section 6.1 presents the task and datasets that we use. Section 6.2 describes our replication work. Finally, Section 6.3 details our new architectures for APE and experimental results.

[5] Our initial goal was to use bilingual sentence embeddings as state representations in reinforcement learning for automatic post-editing. However, we were unable to obtain satisfying results due to the too large action space (and to the difficulty of the task and the lack of data), and decided to use Neural Machine Translation instead (which had just come out).

Publications

- Alexandre Bérard, Christophe Servan, Olivier Pietquin, and Laurent Besacier (2016b). "MultiVec: a Multilingual and Multilevel Representation Learning Toolkit for NLP." in: *LREC*
- Alexandre Bérard, Olivier Pietquin, Laurent Besacier, and Christophe Servan (2016a). "Listen and Translate: A Proof of Concept for End-to-End Speech-to-Text Translation." In: *NIPS End-to-end Learning for Speech and Audio Processing Workshop*
- Christophe Servan, Alexandre Bérard, Zied Elloumi, Hervé Blanchon, and Laurent Besacier (2016). "Word2Vec vs DBnary: Augmenting METEOR using Vector Representations or Lexical Resources?" In: *COLING*
- Alexandre Bérard, Olivier Pietquin, and Laurent Besacier (2017). "LIG-CRIStAL System for the WMT17 Automatic Post-Editing Task." In: *WMT - Shared Task Papers*
- Marcely Zanon Boito, Alexandre Bérard, Aline Villavicencio, and Laurent Besacier (2017). "Unwritten Languages Demand Attention Too! Word Discovery with Encoder-Decoder Models." In: *ASRU*
- Alexandre Bérard, Laurent Besacier, Ali Can Kocabiyikoglu, and Olivier Pietquin (2018). "End-to-End Automatic Speech Translation of Audiobooks." In: *ICASSP*
- Pierre Godard, Marcely Zanon Boito, Lucas Ondel, Alexandre Bérard, François Yvon, Aline Villavicencio, and Laurent Besacier (2018). "Unsupervised Word Segmentation from Speech with Attention." In: *Interspeech*

Part I

State of the Art

Chapter 1

Machine Translation and Automatic Post-Editing

In this chapter, we give an overview of the state-of-the-art methods for Machine Translation, and the related task of Automatic Post-Editing. We define both tasks, their training data, and evaluation metrics; and we describe two categories of techniques: Statistical Machine Translation, and the more recent Neural Machine Translation.

1.1 Machine Translation

1.1.1 Definition

Machine Translation encompasses all the techniques that automatically translate (with software) text or speech from one language to another. In this thesis, we study text-to-text translation, which we (somewhat abusively) call Machine Translation (MT); and speech-to-text translation, which we call Automatic Speech Translation (or AST).

Most MT techniques are at the sentence level. The task consists in: given a sentence in one language, find a translation of this sentence in the target language. Also, in this work we assume that the entire sentence is available at once (contrary to computer-aided translation, or real-time MT). Machine Translation generally assumes a word segmentation of its input, obtained by a process called "tokenization". Punctuation symbols, words or numbers are considered as separate "tokens". Some techniques perform more aggressive segmentation (e.g., splitting compound words), or even work at a subword-level or character level.

There are several categories of techniques for (text-to-text) Machine Translation:

- Rule-based MT (RBMT) methods use a set of rules, hand-built and/or automatically extracted. These rules construct an intermediate (more abstract) representation of the input sentence, and then map this representation to the target language. This kind of approach can give good results, but it requires a lot of engineering and costly linguistic resources. These methods also lack flexibility as they are often difficult to port to new domains or new languages.

- Statistical MT (SMT), and more specifically Phrase-based MT (PBMT) consists in training several statistical models on text corpora to extract features (phrase table, language model, etc.) and combining these features into a single model. Contrary to RBMT, the training data here is unstructured as it consists of monolingual texts and translated texts. SMT gets rid of many linguistic considerations that are ubiquitous in RBMT (e.g., morphological analysis, parsing, etc.) and does very few assumptions about the structure of languages.
- Neural MT (NMT), introduced fairly recently, uses artificial neural networks to learn MT models from parallel data. The main difference with the SMT approach is that these models are often trained end-to-end, and make even less assumptions about their inputs (e.g., monotonicity). Also, while SMT methods are very shallow (there is no semantics involved), NMT models learn abstract representations of their inputs.

SMT and NMT are corpus-based techniques. This means that they use statistical models that are automatically trained on corpora of texts. Generally, they make use of *parallel corpora*, which contain pairs of translated sentences in the source and target languages. Sometimes, monolingual data in the target language is also used (often in larger amounts).

In this section, we give a quick description of PBMT, which was considered as the state-of-the-art until recently. Then we give an overview of existing methods in Neural Machine Translation (NMT). The NMT methods that are most relevant to our work, namely sequence to sequence models, are described more in details in Chapter 3.

1.1.2 Evaluation

After prototyping and training a new MT model, an important step is its evaluation: to validate/invalidate certain design choices and further improve the model, or to compare against other models. For this purpose, as with other machine learning tasks, it is advisable to split the available (parallel) data into three sets: a *training set* on which the model is directly optimized; a *development set* (dev) which is used to validate design choices and manually or automatically tune the hyperparameters; and a *test set* for final comparison against the state of the art.

Another important question which comes to mind is how to measure the performance of a given model on the dev and test sets. To do so, we can ask humans to review the translation hypotheses, or use an automatic evaluation metric.

A test (or dev) set contains pairs of sequences: a source language sentence, and a target language reference sentence (the *reference*, *ground truth* or *gold standard*). We call a translation *hypothesis* or *candidate* the automatic translation of a source test sentence into the target language by a given MT model.

Human judgment Depending on the resources at hand, a common practice is to ask professional translators to review the translation hypotheses (of the test set), or to crowd-source the evaluation using platforms like Amazon Mechanical Turk.

Some strategies for human evaluation are:

- Direct assessment: human judges are asked to score each translation candidate on a given scale (e.g., between 1 and 5). They can be asked to give a fluency and an adequacy score.

- Ranking: human judges are asked to rank several candidate translations (from different models) from best to worst.
- Post-Editing effort: time (or some other effort measure) taken by a human translator to post-edit a translation candidate into a correct translation (according to clearly defined quality standards).

These methods require some amounts of normalization between judges (judges may not agree, and some judges may be harsher than others). Furthermore, the exact same method needs to be used to compare systems: a score on its own does not mean anything. Human methods are also costly and time consuming, which makes them unfit for quick research iteration and hyperparameter tuning. These methods are often used to compare several models in-house, and in shared tasks (in workshops or conferences) where several systems are submitted at once.

Automatic metrics When prototyping models, a preferred solution is to use automatic evaluation metrics. A translation metric compares each translation hypothesis against one or several gold-standard references (the target side of the dev/test set). Several metrics have been designed to correlate as close as possible to human judgment.

BLEU (Papineni et al. 2002), a corpus-level metric, is arguably the most popular evaluation metric for MT. It measures a geometric average of n-gram precisions, multiplied by a brevity penalty which penalizes too short candidates. An n-gram is a sequence of n consecutive tokens.[1] The BLEU score between a set of hypotheses $\mathcal{D}$ and corresponding references $\mathcal{D}'$ is computed as follows:

$$\mathrm{BLEU}(\mathcal{D}, \mathcal{D}') = \mathrm{BP}(\mathcal{D}, \mathcal{D}') \exp\left(\frac{1}{4}\sum_{k=1}^{4} \log p_k(\mathcal{D}, \mathcal{D}')\right) \tag{1.1}$$

$$p_k(\mathcal{D}, \mathcal{D}') = \frac{1 + correct_k}{1 + total_k} \tag{1.2}$$

$$correct_k = \sum_{s,s' \in (\mathcal{D},\mathcal{D}')} \sum_{t \in \mathcal{G}_k(s)} \min(\mathcal{G}_k(s)[t], \mathcal{G}_k(s')[t]) \tag{1.3}$$

$$total_k = \sum_{s \in \mathcal{D}} \sum_{t \in \mathcal{G}_k(s)} \mathcal{G}_k(s)[t] \tag{1.4}$$

$$\mathrm{BP}(\mathcal{D}, \mathcal{D}') = \min(1, e^{1 - \frac{|\mathcal{D}|}{|\mathcal{D}'|}}) \tag{1.5}$$

In p_k, the numerator computes a sum of k-gram counts in each hypothesis clipped by the k-gram count in the corresponding reference (i.e., it counts the number of *correct* k-grams). The denominator counts the total number of k-grams in the hypothesis set. $\mathcal{G}_k(s)$ is the set of distinct k-grams in s, and $\mathcal{G}_k(s)[t]$ is the number of occurrences of k-gram t in s. $|\mathcal{D}|$ is the number of words in the hypothesis set. Because BLEU only evaluates n-gram precision, and not recall, very short translations can have high scores. To compensate for this, BLEU adds a brevity penalty $\mathrm{BP}(\mathcal{D}, \mathcal{D}')$, which penalizes hypotheses which are shorter than the reference.

BLEU is also designed to be compatible with multiple references. When several translation references are available, we want to give high scores to hypotheses that are close to any of the references. In this case, $\mathcal{G}_k(s')[t]$ in Equation 1.2 corresponds to the maximum occurrence count

[1] A token is anything delimited by whitespaces, after tokenization: generally a word, a number or a punctuation symbol.

of n-gram t in all references. In Equation 1.5, $|\mathcal{D}'|$ is computed by summing the lengths of the shortest references.

A BLEU score of zero means no correspondence with the reference, while a score of one (or 100%) means an exact match between the hypothesis and the reference. Exact matches are rare, and BLEU scores are generally well below 100%. A big problem with BLEU is that it only looks for exact n-gram matches with a reference. However, a translation can be perfect without having any word in common with a given reference translation.

METEOR (Banerjee et al. 2005) improves BLEU by allowing synonyms or word stems to match. Contrary to BLEU, it takes precision and recall into account. It works at the unigram level, while encouraging monotonous alignments. METEOR correlates better with human judgment than BLEU, but it is slower to compute and requires access to linguistic resources that are not available for every language (e.g., WordNet).

Translation Edit Rate (TER) (Snover et al. 2006) is the minimum number of basic edit operations needed to transform the hypothesis to the gold-standard reference. The available operations are: insertion or deletion of a word, substitution of a word by another, or changing the position (shift) of a group of words. TER is mostly used in post-editing scenarios (HTER), where the available reference is a human post-edited version of the hypothesis (Snover et al. 2009).

1.1.3 Statistical Machine Translation

Definition The concept of statistical machine translation was first introduced in 1949 by *Warren Weaver*, who saw translation like a problem of cryptography:

> *"One naturally wonders if the problem of translation could conceivably be treated as a problem in cryptography. When I look at an article in Russian, I say: 'This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode.' "*

Brown et al. (1993) brought this idea to life, by creating the IBM models. Say we want to build a system that translates from French to English. Using Bayes' rule, the conditional probability of translating a French sentence f into an English sentence e can be expressed as follows:

$$P(e|f) = \frac{P(e)P(f|e)}{P(f)} \tag{1.6}$$

The goal of the translation system is to find the English sentence e which maximizes the conditional probability $P(e|f)$. The probability, $P(f)$ is independent of e, we thus arrive to what Brown et al. (1993) call the "fundamental equation of SMT":

$$e^\star = arg\max_{e \in \mathcal{S}} P(e)P(f|e) \tag{1.7}$$

When translating a new sentence f to English, we look for the English sentence $e^\star$ (among the set $\mathcal{S}$ of all possible strings) that maximizes this product of probabilities.

This is an instance of the noisy channel model. The problem was formulated by Brown et al. (1993) as follows:

> *"We further take the view that when a native speaker of French produces a string of French words, he has actually conceived of a string of English words, which he translated mentally. Given a French string f, the job of our translation system is to find the string e that the native speaker had in mind when he produced f."*

The problem is thus reversed by the noisy channel model: instead of modeling the probability $P(e|f)$ that a French sentence f produces an English sentence e, $P(f|e)$ measures the likelihood that the native speaker produced the output string f, given the English string e that he had originally in mind.

The advantage of modeling machine translation in this manner, is that it splits the problem in two distinct problems: the generation of fluent English language, and the translation from English to French. The probability $P(e)$ is the language model probability. It ensures that the output is fluent English. $P(f|e)$ is the translation model probability. The job of the translation model is to find an adequate translation, regardless of its fluency. An important challenge in SMT is to build an efficient search algorithm, which finds the English sentence that maximizes the product of these two probabilities, or a close enough approximation (the decoding algorithm).

Language model The target language model is completely independent of the source language, which means that it can be trained using monolingual data in the target language (English in the example). Such data is much more common than bilingual data. The language model is thus potentially much better informed about the English language than any translation model trained with bilingual data could be.

The probability of a word sequence $w_1, w_2, \ldots, w_n$ can be expressed as follows:

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \times P(w_2|w_1) \times \cdots \times P(w_n|w_1, \ldots, w_{n-1}) \tag{1.8}$$

Most language models do a k-gram independence assumption, which results in:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i|w_{i-k+1}, \cdots, w_{i-1}) \tag{1.9}$$

Where k is the order of the language model. The language model makes the assumption that the probability of observing a word at a given position, only depends on the $k-1$ previous words. It is thus easier to estimate the probability of a new sequence, than when considering the entire context. For example, in a trigram model, the conditional probability $P(w_i|w_{i-2}, w_{i-1})$ can be estimated as follows:

$$P(w_i|w_{i-2}, w_{i-1}) = \frac{count(w_{i-2}, w_{i-1}, w_i)}{count(w_{i-2}, w_{i-1})} \tag{1.10}$$

$count(w_i, \ldots, w_k)$ is the number of occurrences of the sequence $w_i, \ldots, w_k$ in the training data. In practice, more complex models are used (like back-off models), which attribute non-null probabilities to unseen n-grams. Unseen n-grams are very likely at test time, because of Zipf's law (even more so with high n-gram orders).

IBM models The reason why SMT needs parallel data, is for training the translation model. Brown et al. (1993) propose 5 different word-based translation models. The so-called IBM models are based on a word-level alignment of the sentences.

In word-based SMT, each source word may generate several target words. A word-level alignment is a one-to-many relation $j \rightarrow a_j$ between the source sentence and the target sentence, which defines by which source word, each target word was generated. To allow the insertion of target words which are not actually aligned to any source word, a dummy `NULL` token is added at position 0 in the source sentence.

Remember that our system is translating from French to English, but the noisy channel model reverses the translation direction. In the translation model, the source language is English and the target language is French. Figure 1.1 shows an example of word-alignment of a French sentence with an English sentence.

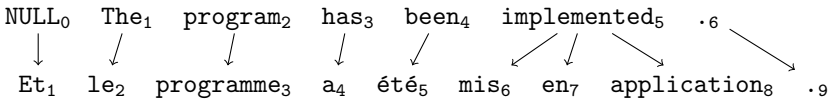


FIGURE 1.1: Example of word-alignment from English to French (from Michael Collins' lecture on IBM models)

The probability of a translation, is the sum of the probabilities of all possible alignments:

$$P(f|e) = \sum_a P(f, a|e) \tag{1.11}$$

The probability of an alignment depends on the IBM model that is used. In the simplest model (IBM 1), the probability of aligning two words f_j and e_{a_j} depends only on their translation probability $t(f_j|e_{a_j})$.

$$P(f, a|e) = \frac{\epsilon}{(l_f + 1)^{l_e}} \prod_{j=1}^{|e|} t(f_j|e_{a_j}) \tag{1.12}$$

Where l_e is the length of the English sentence e, l_f the length of the French sentence f, and ϵ is a normalization constant.

However, this model does not take into account the position of the words that are aligned. The first French word is as likely to be aligned with the first English word as with the last. IBM Model 2 includes an alignment probability, which models the probability $P(j|i, l_f, l_e)$ of aligning an English word at position i with a French word at position j. In IBM Models 1 and 2, the alignment of a French word does not take into account the alignment of the other words in the sentence. All French words may be aligned to the same English word (and all other English words aligned to nothing). IBM Model 3 includes a fertility model, which models the number of French words each English word is aligned to. IBM Model 4 and 5 bring other improvements, such as improving the reordering model, by taking into account the movement of several consecutive words together.

The translation probabilities $t(f_j|e_{a_j})$ and alignment probabilities $P(j|i, l_f, l_e)$ are taken from probability tables that are built using the training data. However, the training data consists of sentence-aligned texts, which need to be aligned at the word-level in order to produce these tables. This is a chicken-and-egg problem, and it is solved by the *Expectation-Maximization* algorithm, which updates the parameters of the model iteratively, so as to maximize the alignment probability of the training data. Here is a quick outline of how the EM algorithm works:

1. Initialize the parameters of the model (e.g., with uniform probabilities).

2. Align the training data with the model.
3. Estimate model parameters from the aligned data.
4. Repeat steps 2 and 3 until convergence.

Initially, all alignments are as likely. But actual translations generally co-occur more often in the training data than non-translations. For example, "kangaroo" and "kangourou" will probably be aligned more often than "squirrel" and "kangourou". Thus, at the parameters estimation step, the probability $t(kangourou|kangaroo)$ will be greater than $t(kangourou|squirrel)$, which will result in a better alignment at the next iteration.

Since the estimation of the model parameters is purely based on co-occurrence, the larger the parallel corpus used as training data, the better the translation model (greater vocabulary coverage and more reliable alignments). A popular implementation of the IBM models is GIZA++ (Och et al. 2003).

Phrase-based model While the IBM models are still used for word-level alignment, word-based translation has been supplanted by other methods. Word-based translation has some shortcomings. In particular, it is unable to deal with many-to-many translations. For instance, the French phrase "casser sa pipe", would translate literally as "break his pipe" (a more correct translation would be "buy the farm" or "kick the bucket").

Phrase-based translation, by Koehn et al. (2003), is the state-of-the-art method in SMT. It is very similar to word-based translation, except that instead of independent words, small units of consecutive words, called phrases, are translated. A phrase translation table is built by merging word-based alignments in both directions (English to French and French to English). These alignments are produced by a word aligner like GIZA++ (Och et al. 2003). Table 1.1 shows examples of entries in a phrase translation table, for the French phrase "bien sûr". Such phrase tables can be extremely large (several gigabytes when stored as text).

Translation $\bar{e}_i$	Probability $\phi(\bar{f}_i\|\bar{e}_i)$
of course	0.5
naturally	0.3
of course ,	0.15
, of course ,	0.05

TABLE 1.1: Examples of phrase translation table entries for French phrase "bien sûr".

The first version of the model, by Koehn et al. (2003), is defined as usual with a noisy channel model:

$$e = arg\max_{e \in e*} P(f|e)P(e) \tag{1.13}$$

$P(e)$ is a standard language model, augmented with a length factor to fix the bias of language models toward short translations. $P(f|e)$ is the phrase-based translation model. The English sentence e is decomposed into I phrases $\bar{e}_i$, which are translated into French phrases $\bar{f}_i$. The English phrases are reordered to form the output sequence e:

$$P(f|e) = \prod_{i=1}^{I} \phi(\bar{f}_i|\bar{e}_i)d(a_i - b_{i-1}) \tag{1.14}$$

$d(a_i - b_{i-1}) = \alpha^{|a_i - b_{i-1} - 1|}$ (with $\alpha \leq 1$) is the distortion probability, which penalizes the phrase reorderings in the English sentence. a_i is the start position (in terms of words) of phrase $\bar{e}_i$, and b_i the end position of $\bar{e}_{i-1}$. $\phi(\bar{f}|\bar{e})$ is the translation probability of phrase $\bar{e}$ into phrase $\bar{f}$, which comes from the phrase translation table.

State-of-the-art phrase-based MT is implemented by the open source framework *Moses* (Koehn 2010). A log-linear model is used to include more features:

$$p(e|f) = \exp \sum_{i=1}^{n} \lambda_i h_i(e, a, f) \tag{1.15}$$

h_i are the features (e.g., language model, phrase model, distortion model, word penalty), and λ_i are their respective weights. The weights are usually automatically tuned, so as to maximize the BLEU score on the development set, with tools like MERT (Och 2003).

1.1.4 Neural Machine Translation

Statistical Machine Translation, and in particular Phrase-Based MT was the state-of-the-art method for MT a couple of years ago. Recently, a new category of methods called "Neural Machine Translation" has started to appear, and has progressively replaced SMT. The main difference with SMT is that these models are trained end-to-end. A single deep neural network reads the input sequence, and produces an output sequence. The parameters of this model are optimized so as to maximize the likelihood of the training data.

Introduction Schwenk (2012) trains feed-forward networks to compute continuous representation of phrases, which are used to rescore pairs in a PBMT system's phrase table. This additional score can be used as a feature in the log-linear model (in addition to the standard phrase table scores). Cho et al. (2014b) propose a similar approach based on Recurrent Neural Networks (RNNs).

The first end-to-end neural-based approach is (Sutskever et al. 2014), which applies not only to MT but to any sequence to sequence task. It consists in a first Recurrent Neural Network (the encoder) which reads the input sequence and encodes it into a fixed-size vector, and a second RNN (the decoder) which does target language modeling conditioned on this representation. The model is trained end-to-end on parallel data, without monolingual data.

Bahdanau et al. (2015) improve this method, by adding an attention mechanism. They observe that the encoder-decoder approach has trouble translating long sequences, because the encoder is forced to encode arbitrarily long sequences as a fixed-size representation. To deal with this problem, the target language model is conditioned on an attention model, which can look anywhere in the input sequence at each time step. We detail both these approaches, which are the basis of our work, in Chapter 3.

Contrary to PBMT, these methods work at the word-level. Phrases in SMT are a way to take local context into account. In NMT, the entire context is summarized into a single attention vector. Furthermore, NMT gives decent results with a greedy decoding approach, where the target sequence is generated word by word from left to right (by conditioning only on the current decoder state, and not on the future). PBMT cannot do without a more complex search algorithm (generally beam search).

Since then, Neural Machine Translation has supplanted SMT as the state-of-the-art (Junczys-Dowmunt et al. 2016a; Luong et al. 2015c), now beating PBMT in MT competitions (Bojar et al. 2016; Sennrich et al. 2016b). As a result, NMT models have been deployed in the industry, for example at Google (Wu et al. 2016) and at SYSTRAN (Crego et al. 2016).

Out-of-vocabulary words In (Bahdanau et al. 2015) and (Sutskever et al. 2014), training and decoding complexity is linear with respect to the number of possible target tokens (vocabulary size). For efficiency reasons, Bahdanau et al. (2015) limit the source and target vocabularies to a shortlist of the most frequent 30k words. Out-of-vocabulary words are replaced by a special `UNK` symbol. This can strongly degrade translation quality, as the encoder may have trouble producing a proper representation of the input sequence, and many `UNK` tokens may be generated at test time.

A partial solution is to use a larger vocabulary size, but even with large vocabularies many unknown tokens can be encountered at test time (e.g., proper nouns, numbers, spelling mistakes, etc.) Jean et al. (2015a) propose a "sampled softmax" technique, which makes the model's complexity constant with respect to vocabulary size.

Other (complementary) solutions consist in replacing the generated `UNK` symbols, with the corresponding (if any) unknown word in the input, or by a translation of this word obtained with a dictionary. A solution is to use the word-level alignment produced by the attention mechanism (Jean et al. 2015a). However, this is difficult as a single target word may be aligned to multiple source words (soft alignment). Another solution, by Luong et al. (2015c) is to output special `UNK` tokens (`UNK_-1`, `UNK_0`, etc.) that encode the relative position of the aligned source token (which is available at training time with a GIZA++ word alignment).

However, the best results to date are obtained with subword units (Gehring et al. 2017b; Shazeer et al. 2017; Wu et al. 2016). Any word can be read or generated as a combination of several subword units. Those are automatically extracted from the training set using data-driven techniques like Byte-Pair Encoding (Sennrich et al. 2016c) or Google's Word Piece Model (Wu et al. 2016).

Other works, like Lee et al. (2016) or Kalchbrenner et al. (2016) go even further and do character-level translation. This completely removes the need of unknown word replacement techniques as any word can be created as a combination of its constituent characters. An advantage of character-level translation is that there is less data sparsity due to having capital letters: in word-level or subword-level MT, "The" and "the" account for two different vocabulary tokens. Also, this removes the need of doing word tokenization as pre-processing and de-tokenization as post-processing (whitespaces are considered as characters).

SOTA results The original model proposed by Bahdanau et al. (2015) is small compared to current state-of-the-art NMT models. The best results on large training sets (like WMT14 English-French or English-German) are obtained with much larger models.

Luong et al. (2015c) get similar results as the best performing SMT model (Durrani et al. 2014) on WMT14 En→Fr by using a deep model (6 LSTM layers) without attention. They ensemble 8 independently trained models, and use a strategy to replace unknown words in the output. Each model takes about ten days to train on 8 high-end GPUs.[2]

[2] Bahdanau et al. (2015) takes about ten days on a single GPU.

Zhou et al. (2016) use an even deeper encoder and decoder, with 16 LSTM layers in total. They use residual connections, which makes training easier in multi-layer recurrent models. They get a +3 BLEU increase over Luong et al. (2015c). Wu et al. (2016) train a similarly sized model that uses subword units instead of words, and refine training with reinforcement learning (optimization directly w.r.t. the evaluation metric). This latter model takes 6 days to train on 96 K40 GPUs. With an ensemble of 8 models, they get an increase of +0.8 BLEU compared to Zhou et al. (2016). They also train models on Google production data (which are now deployed inside Google Translate), and report almost human-level performance (in particular with French to English and English to Spanish).

A number of other neural network architectures have also been proposed that strongly depart from the recurrent models of Sutskever et al. (2014) and Bahdanau et al. (2015).

Vaswani et al. (2017) present the *Transformer*, a neural network which does not make use of RNNs at all. It consists in many layers that have several *self-attention* heads. This means that each layer can look anywhere in the previous layer. They manage to obtain state-of-the-art results on WMT14, with a single model (no ensemble) which takes one order of magnitude less computation time that its competitors: 3.5 days of training on 8 P100 GPUs.

Gehring et al. (2017b) propose a convolutional sequence to sequence model, which is much faster at test time (about 10 times) than Wu et al. (2016). It also gets the best results to date on WMT14 En→Fr, with a BLEU score of 41.6 (with an ensemble of 10 models). This is 0.5 BLEU above Wu et al. (2016). However, a single model takes 37 days to train on 8 GPUs.

Kalchbrenner et al. (2016) stack the encoder and decoder on top of each other, and allow longer outputs by padding the input sequence with dummy symbols. The encoder and decoder use several convolutional (non-recurrent) layers. The ByteNet model uses dilation between the convolutional layers, which results in an exponentially increasing receptive field.[3] Thanks to this large receptive field, it can do without RNNs and does not use an attention mechanism. Because the model's complexity is linear w.r.t. the input length (contrary to attention-based NMT), it can translate at the character level in reasonable time.

Multilingual NMT An advantage of deep NMT models is that the encoder produces an abstract representation of the input sequence. This hints at the possibility of an *interlingua*, i.e., some sort of universal language (our encoder's state) in which a sentence in any language can be encoded, before being translated to another language.

In SMT, one model needs to be trained for each language pair, which gives $n \times (n-1)$ models where n is the number of languages. This is not possible for all language pairs, as parallel data is not available in sufficient amounts for most of them. In this case, a pivot-based approach is used, where a pivot language acts as a bridge between two other languages. For example, to translate from English to Ukrainian, one can translate from English to Russian (large amounts of parallel data), and then from Russian to Ukrainian (linguistic proximity). A problem with this approach is that most often English is chosen as this pivot language (because training data is scarce for other language pairs), even though it is very distant linguistically-speaking from the source and target language. This can lead to a large loss of information and translations with a low adequacy.

[3] In a given encoder (or decoder) layer, each state looks at several states from the previous layer (using convolutions). While the first encoder layer looks at n consecutive states, the next layers look at states that are exponentially further apart (as the dilation rate increases).

Firat et al. (2017) propose an approach for multilingual machine translation, where the number of parameters is linear with respect to the number of languages (and not to the number of language pairs). The basic idea is that a single encoder can be trained for each language, and shared across many language pairs (and similarly for the decoder). A single attention mechanism is shared across all language pairs. The scores of the multilingual model are close to those of the single models, and even better in low-resource settings (where only small amounts of parallel data are available).

Johnson et al. (2016) propose to use the same model as Wu et al. (2016), but for multilingual MT. The same encoder and the same decoder are shared between many language pairs. The idea is very simple: a single word-piece vocabulary is created and shared between all languages. All the training data is concatenated, and an extra token is added to the beginning of the source sequence to choose the target language. The authors observe promising results on Google production data: slightly degraded results on high-resource language pairs (e.g., English-French), but improved results on low-resource pairs. Furthermore, this model is able to translate in language pairs which had no parallel data during training (zero-shot translation).

Comparison with SMT Isabelle et al. (2017) released a challenge set for evaluating MT models, containing a number of hand-selected difficult translation cases between English and French. They observe that the best performing NMT systems (Wu et al. 2016) still make some systematic errors. In particular, NMT systems are much worse than PBMT systems at translating idioms. They also always fail to get the order of arguments right with French verbs like “manquer” (where the subject and the object are reversed compared to its English translation “miss”).

Koehn et al. (2017) show that NMT performance is much more sensitive to the amount of training data than SMT. They train many English→Spanish NMT and SMT (Phrase-Based) systems with varying amounts of parallel data (from 0.4M to 356M words). They observe (on a news test set) that NMT starts outperforming SMT when there are more than 15M words in the training set. When using all the parallel data available, NMT even beats an SMT system that uses a huge (2 billion words) language model. However, with small amounts of data (less than 1M words) NMT performance is catastrophic (BLEU scores close to zero), while SMT obtains decent BLEU scores ($\approx$ 18 BLEU). Furthermore, SMT can easily harness larger volumes of monolingual data via its language model, which gives significant improvements, especially in low-resource settings. The authors also observe that NMT is less robust to out-of-domain translation: model trained on parallel data from one specific domain (e.g., movie subtitles) and evaluated on another domain (e.g., medical domain).

Östling et al. (2017) do similar observations. They train several models (German, Czech, French or Spanish to English) on small amounts of data: a subset of the Bible with 130k words, or Watchtower (a religious magazine) with 70k words. Compared to SMT, NMT scores are disastrous, in particular with the smaller Watchtower training set. A common observation of Koehn et al. (2017) and Östling et al. (2017) is that NMT tends to sacrifice adequacy for fluency: it is frequent to observe fluent, understandable NMT output that is completely unrelated to the input. SMT tends to do the reverse, defaulting to word-by-word translation or even recopying its input when too little information is available.

Neural Machine Translation in low-resource settings is still an open problem.[4] A common approach to mitigate this problem is to do transfer learning. In (Zoph et al. 2016b), a French to

[4]This was the main topic of the 2017 JSALT summer workshop: https://duyvuleo.github.io/ws17mt/

English model is trained on large amounts of training data. This model is then used to initialize other models, which have very small amounts of training data, but where the target language is also English (e.g., Urdu to English). With this pre-training approach, they obtain similar performance on these low-resource language pairs as a robust SMT system (trained on the same data). The multilingual approach presented earlier is also a solution, as it allows translation in language pairs that have no parallel data. Johnson et al. (2016) also show that finetuning this multilingual model with small amounts of parallel data, in an unseen language pair, greatly improves the translation quality for this language pair.

Artetxe et al. (2018) and Lample et al. (2017) propose unsupervised NMT models, i.e., models that do not use any parallel data. Both approaches are very similar: they use pre-trained word representations in the source and target language that share the same vector space (Artetxe et al. 2017; Conneau et al. 2017a). They also do back-translation, where the model is used to translate target language (monolingual text) to the source language, and then trained to reconstruct this target language sequence by reading the automatically generated source language sequence.

1.2 Automatic Post-Editing

In addition to Machine Translation, we are interested in the related field of Automatic Post-Editing (APE). Until recently, the techniques to solve this problem were mostly based on Statistical Machine Translation (SMT). Neural-based techniques have started to appear that achieve promising results. A challenge for such methods is the relatively small amount of training data compared to MT (generally in the order of 10k sentences).

1.2.1 Definition

Manual Post-Editing consists in taking the output of a Machine Translation system, and reworking it so that it meets some translation quality standards.[5] It has become a common practice for human translators (e.g., in translation agencies) to use an MT system to obtain a first, often imperfect, translation of the sequence they wish to translate, and then to manually modify this translation to correct its imperfections. Green et al. (2013) do an analysis of post-editing on three language pairs (English to Arabic, German and French), and observe that doing manual post-editing using an SMT system (2012 version of Google Translate) is cost effective compared to translating from scratch. According to (Green et al. 2013), post-editing reduces translation time and even increases the quality of translations.

Alternatives to this approach are translation memories, where the human translator looks for similar examples in a large database of previous translations; or computer-aided translation, where the human translator is interactively assisted by an MT system (e.g., with word translation suggestions).

Translation work-flows that include manual post-editing may result in the generation of potentially large amounts of new data, in the form of sentence triples: source sentence, MT hypothesis, and human post-edited version of this hypothesis. It can be desirable to use this new data to improve the MT system. There are several ways of doing so:

[5] Post-Editing can be done on any machine-generated output, e.g., handwriting recognition or speech recognition. In the context of this thesis, we are interested in Post-Editing of Machine Translation output.

- Re-train the MT system from scratch by adding the new data to the training corpus.
- Incrementally train the MT system with the new data (Hardt et al. 2010). This requires an MT system which can be incrementally trained (which is not the case with vanilla PBMT).
- Train a second system which takes as input the outputs of the MT system, and outputs improved translations.

The first two options require having access to the inner workings of the original MT system, which is not always possible, in particular when using commercial MT systems like Google Translate or SYSTRAN.

The third option is called *Automatic Post-Editing* (APE), because it basically consists in learning to imitate what a human post-editor does. It is the only way of automatically correcting the errors of an MT system in a black-box scenario, i.e., when we do not have access to its decoding process. Analogously to Machine Translation, there are three main types of APE approaches: Rule-based Post-Editing, Statistical Post-Editing (SPE), and Neural Post-Editing (NPE).

1.2.2 Evaluation

Like Machine Translation, Automatic Post-Editing quality can be automatically measured with the BLEU metric. However, an automatic metric which makes better sense is the Translation Edit Rate (TER). When applied in the context of post-editing, i.e., when the reference translation is a post-edited version of the MT hypothesis (and not an unrelated translation of the source sentence), TER is called HTER (for Human TER). Snover et al. (2006) show that the correlation of HTER with Human Judgments exceeds all other metrics: BLEU, METEOR, Human BLEU or Human METEOR. Intuitively, HTER can be seen as a way to extract the post-editing operations that the human translator did on the translation hypothesis, in order to reach a gold-standard post-edited version. It basically measures the post-editing effort by counting the number of words that the post-editor had to shift, delete, insert or substitute (it is a lower bound of the number of basic operations he had to execute).

In the automatic post-editing literature, the preferred evaluation metric is HTER (or TER when comparing to standard translation references), and sometimes BLEU is also shown for indication.

1.2.3 Statistical Post-Editing

Statistical Post-Editing (SPE) was the most popular approach until recently. The basic idea is that automatic post-editing can be seen as a translation task, where the source language is “bad English” and the target language is “good English”. As such, it can be implemented using the same techniques as in Statistical Machine Translation.

However, the results with this approach are mixed. The general consensus is that SPE is useful in the two following scenarios:

- Post-Editing of Rule-based MT output (Simard et al. 2007);
- Domain adaptation: porting a general-domain MT system to a more specific domain (Potet et al. 2012a).

A common problem in academic research for Automatic Post-Editing is the unavailability of corpora of sufficient size. Some existing post-editing corpora are: Potet et al. (2012b) and Turchi et al. (2016, 2017). They are often in the order of ten to twenty thousand sentence triples, several orders of magnitude smaller than the parallel corpora used in MT. The post-edited side is sometimes obtained through crowd-sourcing (Potet et al. 2012b), as the intervention of professional translators is prohibitively expensive.

There are two main approaches in SPE: monolingual and multi-source APE.

Monolingual APE Simard et al. (2007) train a phrase-based machine translation system to translate from MT output to human post-edited output. This is an instance of monolingual post-editing, where the source-language sentence is not used. They study statistical post-editing of a Rule-based MT (RBMT) system, on two datasets: English to French and French to English translation of job ads. This data comes from a website called Job Bank, maintained by the Canadian government, where potential employers can post job ads. Because of the Canadian law, all these ads need to be available both in English and French. For this reason, a Rule-based MT system is used, followed by manual post-editing. Simard et al. (2007) use the translations produced by these post-editors to train two SPE systems, whose input is the imperfect translation produced by the website's RBMT system. These (non-public) datasets contain about 31k triples (source, RBMT output, PE reference) for English-to-French MT, and 39k triples for French-to-English, with on average 11-14 words per segment.

Simard et al. (2007) make the following observations:

- SPE when applied to the outputs of the baseline RBMT system (RBMT+SPE) strongly improves translation quality (as measured by BLEU and TER).
- Training a PBMT system from scratch on the Post-Editing data (source language to post-editing reference) also improves translation quality compared to the RBMT baseline, even though the RBMT+SPE configuration is still better. This suggests that the baseline RBMT system is not very strong.
- Cascading the PBMT and SPE models does not improve translation quality. Possible explanations for this are that the PBMT system is much stronger than the RBMT baseline (thus harder to improve); or that SPE is not adequate for recovering SMT errors. It is important to note that, while the SPE system is trained with outputs of the PBMT system, the target remains the same RBMT post-edited output. This setting is called simulated Post-Editing: the post-editing target is not a true post-editing of the input, but a proxy reference translation. This latter translation is potentially very different from the input, which makes the APE task more difficult.
- When applied to a PBMT system trained on out-of-domain data, domain-specific SPE actually improves translation quality, and by a large margin, even though we are far from the specialized PBMT and the RBMT+SPE results.

The subsequent literature on Statistical Post-Editing confirms the hypothesis that SPE is good for domain adaptation and post-editing of RBMT output; but not so good for post-editing of general-domain SMT outputs (Chatterjee et al. 2015a; Potet et al. 2012a; Wisniewski et al. 2015).

Potet et al. (2012a) perform a study of Statistical Post-Editing of PBMT French to English outputs, with a PE corpus of news data (Potet et al. 2012b). They explore two scenarios: same domain post-editing, and post-editing to a new domain (domain adaptation). Their conclusions are that SPE does not improve the results in a general domain setting, no matter the amounts of PE data (even when adding large amounts of simulated PE data). In a domain adaptation setting however, SPE brings large improvements. Yet, domain adaptation techniques, like PBMT retraining with the new data, or training an additional phrase table on the new data bring larger improvements. These latter approaches of course assume that we have access to the parameters of the baseline PBMT system (not a "black-box" APE scenario).

Another approach for SPE which differs from (Simard et al. 2007), is the multi-source approach of (Béchara et al. 2011).

Multi-source SPE The previously proposed approaches for SPE were monolingual: they only used the MT output and tried to improve it. However, in many cases, this information might not be enough to get a correct translation. If the MT output is too messy, there might be missing words, or mistranslated words that are impossible to recover, or any other loss of information.

In human post-editing, the translator often has access to the source language sentence, which allows her to recover translation adequacy errors, in addition to fluency errors. It can be desirable for an APE system that it also be able to make use of this information (which is always available at training and test time).

Simard et al. (2007) hint at this possibility, by proposing to either incorporate the source language sentence as a new feature in the log-linear model; or to do a combination of an MT system with an APE system at test time (e.g., with re-ranking).

Béchara et al. (2011) propose a simple solution, which merges the source and MT sequence into a single sequence of word pairs, and translates this sequence to the post-editing reference.

To do so, they first align the source side and the MT side of the training set at the word-level using GIZA++. This alignment model can be used to align future sentence pairs (at test time). Then the MT side of the training corpus is pre-processed to replace each word $\texttt{f'}$ by $\texttt{f'\#e}$, where $\texttt{e}$ is the aligned source word. An SPE model is trained to translate from this new input (which combines both MT and source) to the post-editing reference ($\texttt{f'\#e} \rightarrow \texttt{f}$). A problem with this approach, is that it leads to extreme data scarcity: the vocabulary size goes from 9k to 71k, very large compared to the total number of words in the training corpus. Many "words" appear only once in the training set, and many out-of-vocabulary words can be encountered at test time. To reduce this problem, the authors replace the non-translated $\texttt{f'\#e}$ words inside the SPE output with $\texttt{f'}$ (which is already a word in the target language). They also propose a thresholding method, which replaces $\texttt{f'\#e}$ with $\texttt{f'}$ (reverts to monolingual SPE), at training and test time, when the alignment score between $\texttt{f'}$ and $\texttt{e}$ is below some threshold.

Béchara et al. (2011) apply this technique on simulated PE data of software user help. They observe a large improvement over the SMT and SMT + monolingual SPE baselines on French-to-English translation. On English-to-French however, the SPE systems (monolingual and bilingual) always degrade the SMT baseline. A limitation of their experiments is that they do simulated post-editing: the PE reference is actually a reference translation, which is potentially very far from the MT hypothesis. Also, the baseline SMT system is trained by the authors themselves on the same data (a small corpus of software help) using cross-validation. This SMT system is

very small by most standards (also, it does not use any external language model). This may make their results non-transferable to larger state-of-the-art SMT systems.

Béchara et al. (2011) do a detailed analysis of the outputs of their SPE systems, in order to identify where the improvements/degradations come from. To do so, they compute TER statistics between the SPE outputs and the PE reference: average number of insertions, deletions, substitutions and shifts. They observe that the successful English SPE system obtains fewer shifts, insertions and substitutions than the baseline, suggesting that it is good at improving reordering (shifts) and lexical choices (substitutions and insertions). However, it tends to produce longer translations (which results in more deletions by TER). The unsuccessful French SPE system on the other hand tends to reduce the length of the MT candidates by deleting too many words.

Recent work Chatterjee et al. (2015b) do a comparison of both approaches on several language pairs. For this purpose, they use Post-Editing data from Autodesk (software user help). The data contains 14k PE triples for six language pairs having English as source (Czech, French, Polish, Italian, Spanish, German). The authors also use additional data to train similarly sized language models for all target languages (2.5M sentences per language). In this setting, both the monolingual approach (Simard et al. 2007) and the bilingual approach (Béchara et al. 2011) significantly improve translation quality compared to the MT baseline (between 3 and 5 TER points). The bilingual approach brings statistically significant ($p < 0.05$) improvements over the monolingual approach for 5 out of 6 language pairs. Furthermore, the oracle which consists of picking the best of the two is even better (-2 TER points on average), which suggests that both approaches are complementary and could be combined. An interesting and rather counter-intuitive observation is that the relative TER improvements are higher for the best-performing language pairs.

Since the 2015 edition, the Workshop on Machine Translation (WMT) includes an Automatic Post-Editing task.

Chatterjee et al. (2015a) apply SPE to an English-to-Spanish corpus of news with 11k triples (WMT15 shared task) and do not manage to beat the MT "do-nothing" baseline. They try several techniques, including (Simard et al. 2007), (Béchara et al. 2011), new phrase-based features (phrase similarity, reliability and usefulness), a phrase-table pruning strategy (which removes non-reliable phrase pairs) and language models of different sizes (small PE, medium-size in-domain, and large out-of-domain). None of these techniques lead to a significant improvement in scores compared to the MT baseline. Their best configuration shows however a 0.6 TER improvement over the SPE baseline of Simard et al. (2007). They also propose a new corpus-level evaluation metric for APE:

$$\text{Precision} = \frac{\text{Improved}}{\text{Improved} + \text{Deteriorated}} \tag{1.16}$$

A sentence is considered as improved if its TER (w.r.t. PE reference) has decreased (compared to the MT baseline), and it is considered as deteriorated if the TER has increased.

Possible explanations as to why APE is harder in this setting, compared to the Autodesk setting are that the data is of lower quality (obtained by crowd-sourcing, instead of professional translations) and that the news domain is harder than the specific domain of software user help (Autodesk). This latter explanation would support the idea that SPE is especially good in specific domain settings.

Wisniewski et al. (2015) also fail to get improvements over the MT baseline on this corpus. Pal et al. (2015) manage to get a very slight improvement over the baseline, by using a monolingual SPE system, with careful data pre-processing (stemming) and building the phrase table from several word alignments: GIZA++, TER-based alignment, and METEOR-based alignment.

Most subsequent work in automatic post-editing, in particular in the 2016 and 2017 editions of the Workshop in Machine Translation, is based on Neural Machine Translation techniques.

1.2.4 Neural Post-Editing

Junczys-Dowmunt et al. (2016b) and Pal et al. (2016) are the first two contributions that propose to use Neural Machine Translation techniques for Automatic Post-Editing.

Pal et al. (2016) train a basic attention-based NMT model (Bahdanau et al. 2015) on a large post-editing corpus. The corpus contains 312k sentence triples of English-to-Italian translation of news and Europarl data by Google Translate (strong baseline) followed by manual post-editing (non-public data). Their system is monolingual, in the sense that it does not use the source (English) side. Yet, they achieve significant improvements compared to the Google Translate baseline and compared to an SPE (phrase-based) baseline. These are very promising results. However, the training corpus is extremely large by post-editing standards.

Junczys-Dowmunt et al. (2016a) do bilingual neural-based post-editing, in the context of the shared APE task of WMT 2016. The training corpus contains 12k triples of post-edited English-to-German translations in the IT domain. They obtained the best results on the task, by a large margin, beating the MT baseline by more than 3 TER points (5.5 BLEU). Their approach consists in training two NMT models: one model to translate from English to post-edited German (MT from scratch), and one to translate from imperfect German to post-edited German (PE model). Both models are combined at test time by averaging their log-probabilities. The key point for the success of their approach is the use a very large synthetic PE corpus, which is obtained by "round-trip translation". The building process of this corpus is illustrated by Figure 1.2.

- They first get a huge corpus of monolingual German crawled from the web (Common Crawl), with more than one billion sentences.
- This corpus is filtered to keep only well-formed sentences: starting with capital letter, ending with punctuation, minimum 30 letters.
- Then a language model is estimated on the target side of the PE data. This language model is used to compute probabilities for each sentence in the German corpus. The 10M sentences with the highest scores are kept: this filters the corpus to keep only in-domain data.
- Large phrase-based German-to-English and English-to-German systems are trained on all the available parallel data from the WMT IT-domain translation task. The 10M filtered German sentences are translated to English and then back to German using the SMT models. This gives a synthetic corpus of 10M triples, where the original German sentence is used as PE reference, the English translation as source-language sentence, and the German round-trip translation as MT hypothesis.

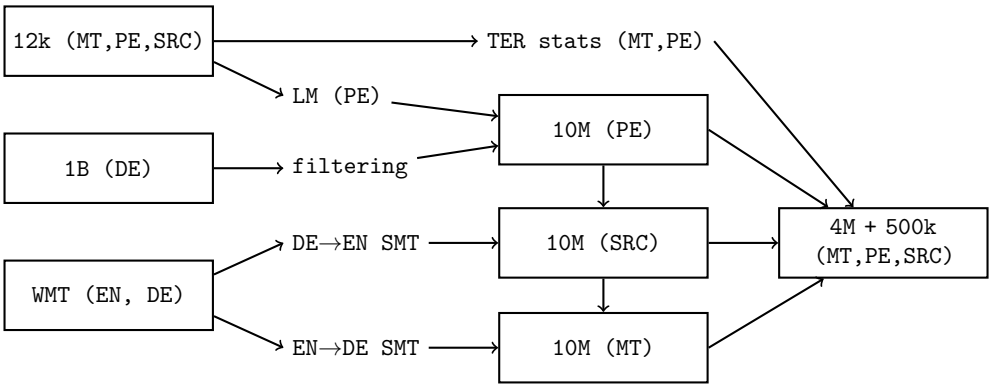


FIGURE 1.2: Building process of the large "4M + 500k" synthetic corpus for automatic post-editing (Junczys-Dowmunt et al. 2016b). "1B" is the German Common Crawl corpus, "WMT" is a large parallel corpus available for the translation task of WMT 2016. "12k" is the real post-editing corpus. "10M (SRC)" is obtained by translating "10M (PE)", and "10M (MT)" is obtained by translating "10M (SRC)" (round-trip translation).

- To get post-editing statistics close to those of the training corpus (same number of deletions, insertions, etc.), the authors get the 500k closest triples in terms of TER statistics as a medium-size synthetic PE corpus, and the 4M best triples as a large size (but noisier) PE corpus.

Their models are trained first on the large 4M corpus. Then, they finetune the models on the smaller and cleaner 500K corpus, to which they concatenate the real PE corpus (oversampled 20 times). Then, four instances of the MT and APE models are trained, and used in an ensemble of 8 models. Another feature is added to the log-linear model that counts the number of new words introduced in the APE output. This Post-Editing-Penalty (PEP) discourages candidates that are too different from the MT hypothesis. The weights of the 9 features are tuned using MIRA (Hasler et al. 2011), which looks for a set of weights that maximizes the BLEU score of the dev set. Thanks to this large model, Junczys-Dowmunt et al. (2016b) obtain unprecedented results on the APE task.

However, so large amounts of monolingual and parallel data are rarely available. One could wonder how the trained PBMT system on English to German would perform if applied to the source side of the APE corpus (translation from scratch). Also, this approach is tedious and requires large computation power: round-trip translation of 10M sentences with PBMT systems, and training 8 large NMT models.

Op-based APE Libovický et al. (2016) train a neural-based APE system, whose target is not a sequence of words, but a sequence of edits operations. These operations are extracted by computing the shortest edit path between the MT hypothesis and the PE reference. The available operations are: keep current word, delete it, or insert a new word. An advantage of this approach is that it is easy, even with little training data, to learn the identity, i.e., an APE system that does nothing and keeps the MT hypothesis as it is. Word-based Neural Post-Editing, on the other hand, can easily degrade the MT hypothesis if trained with too little data. Libovický et al. (2016) manage to slightly improve over the MT baseline, but get much lower scores than Junczys-Dowmunt et al. (2016b).

Multi-source APE Chatterjee et al. (2017) submitted a multi-source Neural Post-Editing system to the shared APE task of WMT 2017, which obtained the best score on this task (Bojar et al. 2017). This approach is very similar to Junczys-Dowmunt et al. (2016b), because it uses the same synthetic training data. However, instead of using an MT system and an APE system in a log-linear model (separate training, joint evaluation), they do joint training with a multi-encoder model (Zoph et al. 2016a). The decoder takes as input attention vectors over the source sequence (SRC encoder) and attention vectors over the MT hypothesis (MT encoder).

When training an NMT model to translate the source sentences from scratch (SRC to PE), they observe that:

1. The NMT model performs worse than the baseline MT model, with BLEU evaluation w.r.t. PE reference.

2. It actually performs better when the reference translation is used as target (and not the post-edited MT hypothesis, which favors the MT baseline). This evaluation tends to indicate that such large amounts of parallel data[6] are enough to retrain an MT model that performs much better than the baseline (thus making APE unnecessary).

State-of-the-art results on the WMT 2016 data,[7] are obtained by Hokamp (2017). They use the same models as Junczys-Dowmunt et al. (2016b), but with several other features as part of a log-linear model: a factored model (with part-of-speech tags, and dependency parsing), and a multi-source model which just concatenates the source sequence with the MT hypothesis (and uses a single encoder). They also train an "aligned-MT" model which translates from MT hypothesis to PE reference, where each MT word is replaced by the corresponding SRC word, using the word-level alignment produced by the NMT model. This model also obtains state-of-the-art results on the Quality Estimation task (predicting for each word in the MT hypothesis whether it is good or bad) (Bojar et al. 2016).

1.2.5 Tasks and Resources

Several Post-Editing resources are publicly available. A characteristic of APE is that successful models are usually trained on domain-specific data, and even more specifically, outputs from a precise MT system. The implication is that the APE resources are specific to a given APE task (post-editing system X in domain Y), and are rarely used for other tasks (contrary to MT).

Table 1.2 references some existing resources for automatic post-editing that are publicly available. The French-English corpus of Potet et al. 2012b was obtained by automatically translating news data with a state-of-the-art PBMT system (Potet et al. 2010), and post-editing the hypotheses with crowd-sourcing (Amazon Mechanical Turk). Potet et al. 2012a trained Statistical Post-Editing systems on this same data, and were unable to beat the "do-nothing" baseline. This is possibly due to the noisy nature of the post-edits, and to the too general domain (news).

Another popular resource is Autodesk (Zhechev 2012).[8] It consists in automatically translated software documentation (IT domain), with Autodesk's in-house PBMT system. The post-editing references were produced by professional translators. This is a large corpus by APE standards, with 30k to 410k triples in 14 language pairs (with English as source). Chatterjee et al. 2016

[6] The authors train an NMT model on the synthetic parallel data only. The scores could be even higher by training with the parallel data that Junczys-Dowmunt et al. (2016b) used to extract their parallel corpus.

[7] These results were not presented as part of the shared task of the workshop.

[8] https://autodesk.box.com/Autodesk-PostEditing

Corpus	Languages	Domain	Size	Post-Edits
Specia et al. 2010	En-Es	Legal	4k	Pro
Specia 2011	Fr-En/En-Es	News	2.5k/1k	Pro
Zhechev 2012	En-Ch/Cs/Fr/De...	IT	30k-410k	Pro
Potet et al. 2012b	Fr-En	News	11k	Crowd
Bojar et al. 2015	En-Es	News	12k	Crowd
Bojar et al. 2016	En-De	IT	13k	Pro
Bojar et al. 2017	En-De/De-En	IT/Medical	13k/26k	Pro
WMT 2018	En-De	IT	15k	Pro
Synthetic data				
Junczys-Dowmunt et al. 2016b	En-De	IT	4M + 500k	–
Negri et al. 2018	En-De/En-It	Multi	7M/3M	–

TABLE 1.2: Publicly available resources in Automatic Post-Editing. Table extracted from (Negri et al. 2018).

used a subset of this resource to perform a thorough evaluation of existing SPE techniques in several language pairs.

More recently, several APE datasets have been released in the context of the shared APE tasks of the Workshop on Machine Translation (Bojar et al. 2017, 2016, 2015). The English-Spanish data from the 2015 edition was in a general domain, and the post-edits were obtained from non-professional translators. None of the systems submitted to the task were able to beat the "do-nothing" baseline. The 2016 and 2017 editions were more successful, as the data was in a specific domain (IT), and neural-based techniques have started to appear.

A new trend is to use large amount of synthetic post-editing data to train large neural post-editing systems. The synthetic corpus by Junczys-Dowmunt et al. 2016b was obtained by round-trip translating monolingual German data (translating to English and back to German). A particularity of this corpus is that it was filtered to be close the true data distribution of the 2016 APE task (Bojar et al. 2016). More recently, Negri et al. 2018 released a large synthetic corpus for English-to-German and English-to-Italian post-editing. This is a "simulated PE" setting, where parallel corpora are used. The source-language sequences were automatically translated to the target language, creating the MT segments. The reference translations from the parallel corpus are used as post-editing references.

This corpus is in a general domain (actually, it contains data from multiple domains) and it targets two different MT systems (a PBMT and an NMT system). It will be interesting to see whether general domain data can be harnessed to improve domain specific Automatic Post-Editing.

The upcoming APE shared task (WMT 2018) will feature a new challenge: automatic post-editing of two different MT systems. There are two test sets: one that was obtained with the historical PBMT system, and one that was obtained with an NMT system. The domain remains the same, and the source-language data is sampled from the same distribution. In addition to the training data that was available in the two previous editions, 15k new triples are made available whose MT hypotheses were produced with NMT.

Chapter 2

Neural Networks

In this chapter, we are going to delve into a class of statistical models called "Artificial Neural Networks", by giving a tiny bit of theory as well as practical considerations. As of recently, deep neural networks (often containing many more parameters than training points) have become a particularly viable solution to many natural language processing problems, including machine translation. This chapter constitutes a necessary basis to understand the neural machine translation models that are used in this thesis.

2.1 Fundamentals

2.1.1 Definition

Artificial Neural Networks are biologically inspired computational models. Similarly to the brain, they are composed of many cells (neurons), connected to each other with modifiable weights (analogous to brain synapses).[1] They are useful in machine learning, because there exist optimization algorithms that can adapt these weights, in such a way that the neural network approximates a useful function.

Artificial Neuron An artificial neuron has several inputs and a single output. Its input either comes from other neurons, or is given by its environment. The neuron (sometimes called neural unit, or unit) computes a weighted sum of these inputs using modifiable weights (also called parameters). A modifiable bias can also be added to this input.

This linear function of the inputs is then processed by the unit, through an activation function g.

$$z = (\sum_{i=1}^{n} x_i \times w_i) + b = w^\top x + b \tag{2.1}$$

$$a = g(z) \tag{2.2}$$

where $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ are the parameters of the model, x is the input of the neuron (generally presented as a vector of size n), and a its output.

[1] The analogy stops here. Neurons and synapses in the human brain are much more complex than this simple computational model.

Perceptron The perceptron (Rosenblatt 1957) is a very simple artificial neural network. Figure 2.1 gives a graphical representation of the perceptron model. It is composed of a single neural unit whose activation function is the *unit step function*, defined as follows:

$$g(z) = \mathbb{1}(z > 0) \tag{2.3}$$

$$g(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{2.4}$$

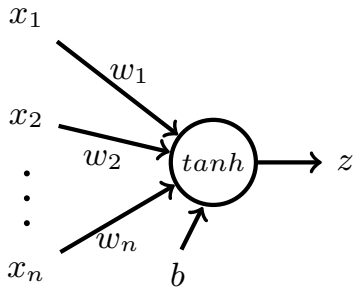


FIGURE 2.1: Perceptron

Intuitively, the perceptron fires a signal ($g(z) = 1$) if its input is above some threshold, and no signal ($g(z) = 0$) otherwise. A non-zero bias b can be interpreted as having a non-zero threshold: $z > 0 \iff w^\top x + b > 0 \iff w^\top x > -b$.

Feed-forward Neural Network One strong limitation of the perceptron model is that it can only compute linear functions. There is a more powerful family of models called "multi-layer perceptrons" or more commonly "feed-forward neural networks."

A feed-forward neural network is composed of several *layers* of neurons, and operates in a directed fashion: each layer reads from the previous layer and sends its output to the next layer. The input layer (first layer) takes its input from the environment (vector $x_1, \ldots, x_n$). Then, there can be any number of *hidden* layers, and a last layer whose output values are the final outputs of the network. The perceptron is a single-layer feed-forward network.

In a network with L layers, the input $z_i^{[k]}$ of the i$^{\text{th}}$ unit in the k$^{\text{th}}$ layer, and its activation $a_i^{[k]}$ are computed as follows:

$$z_i^{[0]} = x_i \tag{2.5}$$

$$\forall k \geq 1 \quad z_i^{[k]} = \sum_{j=1}^{n^{[k-1]}} W_{ij}^{[k]} z_j^{[k-1]} + b_i^{[k]} \tag{2.6}$$

$$a_i^{[k]} = g^{[k]}(z_i^{[k]}) \tag{2.7}$$

$$y_i = a_i^{[L]} \tag{2.8}$$

$n^{[k]}$ is the size of the k$^{\text{th}}$ layer, $g^{[k]}$ its activation function. $W_{ij}^{[k]}$ is the weight that connects the i$^{\text{th}}$ unit of the k$^{\text{th}}$ layer with the j$^{\text{th}}$ unit of the $(k-1)^{th}$ layer. $x_1, \ldots, x_{n_x}$ are the inputs of the network, and $y_1, \ldots, y_{n_y}$ its outputs ($n_x = n^{[0]}$ and $n_y = n^{[L]}$).

Figure 2.2 shows a two-layer feed-forward neural network (i.e., a model with a single hidden layer), with a single output value.

Equations 2.6 and 2.7 can be written in a more compact way with products of matrices. If we write $\mathbf{x} = x_1, \ldots, x_{n_x}$ as the input vector, the computation in a feed-forward network with L layers can be done as follows:

$$z^{[0]} = \mathbf{x} \tag{2.9}$$

$$\forall k \geq 1 \quad z^{[k]} = W^{[k]} z^{[k-1]} + b^{[k]} \tag{2.10}$$

$$a^{[k]} = g^{[k]}(z^{[k]}) \tag{2.11}$$

$$\mathbf{y} = a^{[L]} \tag{2.12}$$

$n^{[k]}$ is the size (number of units) of the k$^{\text{th}}$ layer, $n^{[0]} = n_x$ is the size of the input vector, and $n^{[L]} = n_y$ the size of the output vector. $W^{[k]} \in \mathbb{R}^{n^{[k]} \times n^{[k-1]}}$ and $b \in \mathbb{R}^{n^{[k]}}$ are the parameters (weight matrix and bias vector) of the k^{th} layer.

The vectorized version is generally the preferred way. It is a more compact way of defining and programming a large neural network. Vectorized computation is extremely efficient on specialized hardware compared to explicit sums.

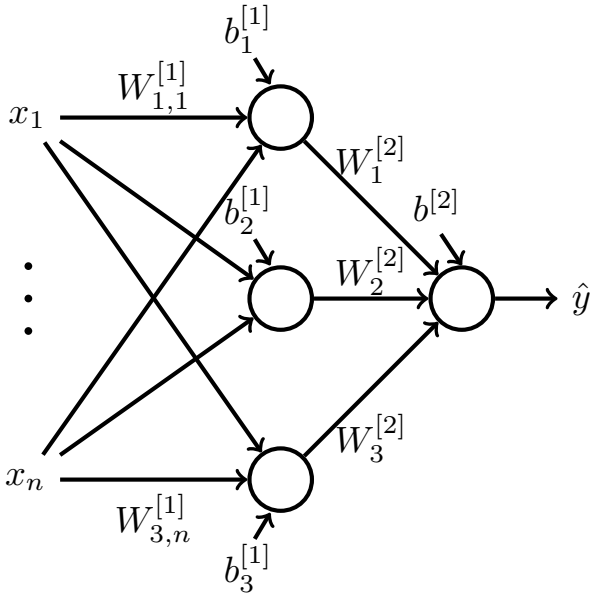


FIGURE 2.2: Example of feed-forward neural network with a single hidden layer of size three, and a single output unit.

We have seen that the perceptron can only compute linear functions. Feed-forward networks are not more expressive, unless using *non-linear* activation functions.

The Universal Approximation Theorem (Haykin 1994; Hornik et al. 1989) states that for any continuous function defined in a compact subset of $\mathbb{R}^m$, there exists a feed-forward network with one hidden layer and a finite number of hidden units that can approximate this function up to the desired precision. This is under the assumption that the feed-forward network uses an activation function which is *non-constant*, *continuous*, *bounded* and *monotonically-increasing* (e.g., *tanh* or *sigmoid*).

For this reason, feed-forward neural networks are said to be *universal approximators*. However, the universal approximation theorem does not state anything about the *learnability* of such network, which can be challenging.[2]

Activation function The activation function used in the perceptron model is the *unit step function* (or *Heaviside step function*): $f(z) = \mathbb{1}(z > 0) \in \{0, 1\}$.

Other candidates, which are most often used in neural networks, are:

1. Identity function (linear activation): $f(z) = z \in (-\infty, \infty)$
2. *Sigmoid* or *logistic* function (soft step) is often used for the last layer as an alternative to the unit step function, to provide a probabilistic output:

$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \in (0, 1) \tag{2.13}$$

3. Hyperbolic Tangent:

$$f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \in (-1, 1) \tag{2.14}$$

4. Rectifier (used in Rectified Linear Units, or ReLU):

$$f(z) = z^{+} = \max(0, z) \in [0, \infty) \tag{2.15}$$

The sigmoid is generally used for the output layer in binary classification problems. The hyperbolic tangent and the rectifier are preferred choices for the hidden layers. The last three activations are non-linear, which makes those good candidates to approximate complex functions.

2.1.2 Machine Learning Basics

As explained by Mitchell (1997): "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

Machine learning is the study of all techniques that allow computer programs (like neural networks) to learn from experience (typically real-world data), so as to better perform on a given task (e.g., machine translation). Neural networks are good candidates for machine learning models, as they are very expressive and modular, and their weights can be learned thanks to optimization algorithms that we shall see later.

Machine Learning Tasks There are three broad categories of Machine Learning tasks:

- Supervised Learning: there is an input (an object, generally a vector, or sequence of vectors), and a clear identified output (often called *label*).

[2] The theorem states the *existence* of such network. Finding this network is an entirely different problem.

  - Classification consists in finding the correct category for some input object. Examples of classification tasks are recognizing the main object in an image, or predicting the next word in a sentence. In binary classification, we choose between two classes, e.g., *cat* or *not cat*. In multi-class classification, we pick a label in a set of classes, e.g., *cat*, *dog*, *bunny*, or *mouse*.
  - Regression is similar to classification, but instead of predicting a categorical output, we predict a continuous value (e.g., movie rating).
  - Structured prediction is a generalization of classification to sequential outputs, i.e., instead of predicting a single label, we predict a sequence of labels. Machine Translation can be modeled as a structured prediction problem.

- Unsupervised Learning: we only get unlabeled inputs, and look for a hidden structure within these objects, i.e., some ways of explaining the data distribution.
  - Representation Learning / Dimensionality Reduction. Here the output we are interested in is an object of the same type as the input, but compressed in some way (i.e., of smaller size). Often, this involves some sort of transfer learning: the learned representation is used as a semantically richer or more concise representation of the input, in a classification or regression task.

- Reinforcement Learning: an agent evolves in an environment (e.g., a game of chess) and has to take actions (chess moves) that lead him to new states (new configurations of the chess board) and potentially give him rewards (e.g., victory or defeat). The goal is to find a policy (a mapping from states to actions) which maximizes the expected cumulative reward of the agent.

In this thesis, we are essentially interested in supervised learning settings, in particular classification or structured prediction tasks. We will therefore delve into these particular aspects of machine learning.

Train, dev and test sets The purpose of machine learning algorithms is to learn from data. More specifically, in supervised learning, the data consists of many *examples*, which are pairs of inputs and their corresponding labels.[3] We assume that this data is not simply noise but that it has some structure, that it has been generated by some natural process. The goal is to find a function that maps the input objects to their label (an approximation of the natural process responsible for generating the data). With this learned function (which we call *model*), we are able to make predictions about future (unlabeled) events that follow the same data distribution.

While learning by heart (by storing all the examples) may seem like a viable solution, we actually want our model to perform well on new examples, i.e., to be able to generalize. When given a point that it has never seen before, the model should be able to predict its label correctly.

That is why it is a common practice to split the available data into three data sets:

- The *training set* is used to train the model (i.e., find the best set of weights and biases).
- The *dev set* (or 'development set', or 'validation set') can be used for model selection (choosing the best model in a set of candidates), deciding when to stop training, or hyper-parameter tuning (e.g., choosing the number of layers and units).

[3] Depending on the task (classification, regression, structured prediction), the label can be categorical, continuous or sequential. Likewise, the input can take all sorts of forms.

- The *test set* is used to evaluate the final performance of a model (e.g., to compare with other state-of-art results). It should not be used to further improve this model.

The dev and test sets generally contain a few thousand examples each, i.e., enough examples to obtain statistically significant estimates of the performance of the model. The train set contains as much data as possible, as the generalization capacity of a model strongly depends on how many training examples it has seen.

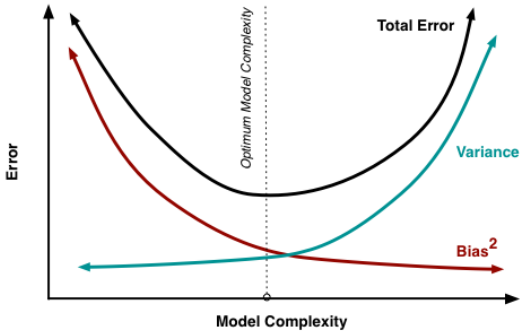
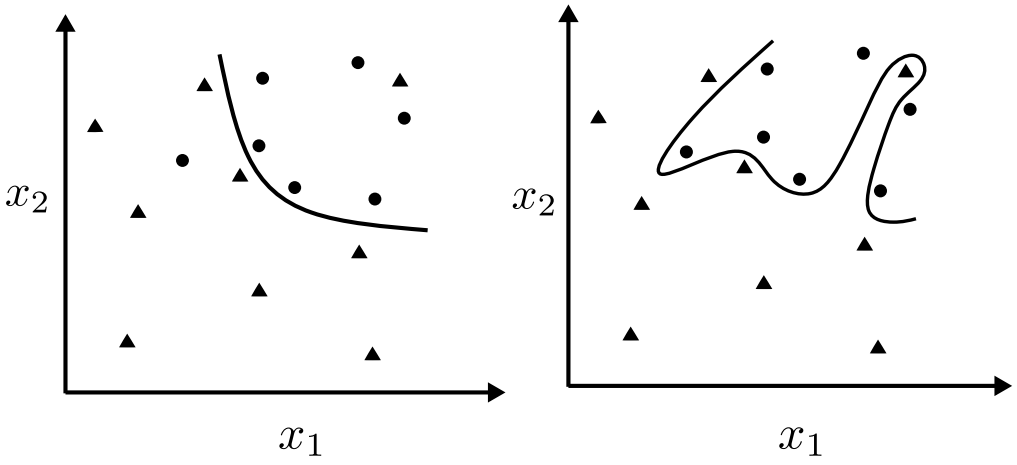


FIGURE 2.3: Test error with respect to model complexity. Too complex models overfit, while too simple models underfit. This figure comes from http://scott.fortmann-roe.com/docs/BiasVariance.html



(A) "Ideal case": the model fits the data, but not too much (it does not try to fit outliers, or noisy points)

(B) Overfitting: the model fits exactly the training points, at a risk of poor generalization to new unseen points.

FIGURE 2.4: Decision boundaries of the triangle and circle classes with different bias/variance trade-offs

Bias and variance a.k.a. underfitting and overfitting When evaluating the performance of a statistical model, the error can be decomposed in two terms: a bias term, which is due to the inability of the model to approximate the true distribution of the data (underfitting, e.g., trying to fit a non-linear dataset with a linear model); and a variance term, which is due to the model being too sensitive and fitting the noise in the training set (overfitting). As illustrated by Figure 2.3, more complex models (e.g., models with more hidden layers) tend to have a lower bias but a larger variance.

Figure 2.4 shows the decision boundaries of two models in a binary classification problem (triangle and circle classes). The first model is unable to correctly classify the entire training set. However, it is more robust to noise and can potentially achieve better scores on unseen data. The model on the right fits exactly all training points but the decision surface is very convoluted, and will probably perform poorly on new points (overfitting).

When designing a machine learning model, it is important to find a good trade-off between bias and variance. Depending on the problem at hand: difficulty of the problem, size of the dataset, amount of noise, different neural model sizes may be appropriate.

The usual way to diagnose an overfitting problem, is to compare the performance of the model on the training set and on the dev set. If the performance on the dev set is much worse than on the train set (and they both come from the same distribution), then it might be overfitting. In this case, the model is probably too large or there is not enough data. There also exist techniques that can reduce model variance without increasing model bias too much, like explicit regularization, artificial data augmentation, or early stopping.

On the other hand, if the performance on the training set is bad, the model is probably underfitting, and should either be made larger or trained longer.

2.1.3 Optimization

Objective function In supervised learning, in order to train a neural network, we need to define an *objective function*. The objective function, or *cost function* gives an estimate on how good the model is at doing the desired prediction task.

A necessary condition of the objective function is that it should be differentiable with respect to the model parameters, so that we can use a family of optimization methods called *gradient descent*, which we shall describe shortly.

The most commonly used objective functions in neural networks are the *Negative Log-Likelihood*, and the *Mean Squared Error*.

Mean Squared Error (MSE) is most common when doing regression, i.e., when the labels are continuous. It is the average of the squares of the differences between the predictions of the model and the true labels:

$$J_{\text{MSE}}(\theta) = \frac{1}{m}\sum_{j=1}^{m}(y^{(j)} - \hat{y}^{(j)})^2 \tag{2.16}$$

where m is the size of the training set, $y^{(j)}$ is the label of the j$^{\text{th}}$ training example, and $\hat{y}^{(j)}$ is the output of the model when taking $x^{(j)}$ as input.

θ corresponds to the entire set of model parameters (weights and biases). The cost depends on θ because different parameter values will lead to different outputs $\hat{y}$ (we omit the θ from $\hat{y}_\theta$ for sake of conciseness). The goal of optimization is to find the set of parameters $\theta^\star$ that minimizes this cost:

$$\theta^\star = \arg\min_\theta J(\theta) \tag{2.17}$$

When doing binary classification (labels belong to one of two classes), we often maximize the *Likelihood* of the training set, which is the estimated probability of its labels according to the model:

$$J(\theta) = \prod_{j=1}^{m} p_\theta(y^{(j)}) \tag{2.18}$$

$$p_\theta(y) = \hat{y}^y(1-\hat{y})^{(1-y)} = \begin{cases} \hat{y} & \text{if y = 1} \\ 1-\hat{y} & \text{if y = 0} \end{cases} \tag{2.19}$$

We dropped the superscript (j) for sake of clarity. $p_\theta(y)$ is the probability of label y according to the model. $\hat{y}$ is the probability of label 1, and because the probabilities sum to one, $1-\hat{y}$ is the probability of label 0.

Because we want the model to give high probabilities to the correct labels (and as a by-product low probabilities to the wrong ones), it makes sense to look for the set of parameters θ that maximizes this quantity.

We generally prefer minimizing the *Negative Log-Likelihood*, defined in Equation 2.20. This is equivalent to maximizing the likelihood (because the natural logarithm is monotonically increasing), but more convenient: the gradient is easier to compute, and there are less numerical errors due to multiplying small numbers together. Furthermore, the negative log-likelihood is identical to a quantity know in information theory as *cross-entropy*:

$$J_{\text{NLL}}(\theta) = -\frac{1}{m}\sum_{j=1}^{m} y^{(j)} \log \hat{y}^{(j)} + (1-y^{(j)}) \log (1-\hat{y}^{(j)}) \tag{2.20}$$

When doing multi-class classification, $\hat{y}$ is not a scalar anymore, but a vector with as many elements as there are classes.

With neural networks, this is generally done by using an output layer of size N, where N is the number of classes, with the identity as activation function. This gives a vector of unnormalized scores for each class. To obtain a probability distribution (i.e., scores that sum to one), we add a *softmax* layer:

$$z = W^{[L]}a^{[L-1]} + b^{[L]} \tag{2.21}$$

$$\hat{y} = \text{softmax}(z) \tag{2.22}$$

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}} \tag{2.23}$$

The decision rule is to pick the class whose score is the highest:

$$c = \arg\max_{i=1}^{N} \hat{y}_i = \arg\max_{i=1}^{N} z_i \tag{2.24}$$

The negative log-likelihood objective can be extended to the multi-class setting.

There are two ways of defining the labels $y^{(i)}$: with one-hot vectors, i.e., vectors of size N with a one at a specific position, and zeros everywhere else. For example, with *(cat, mouse, bunny, dog)*, the label for *cat* would be $(1,0,0,0)$, the label for *mouse* $(0,1,0,0)$, etc.

In the "one-hot vector case", the negative log-likelihood is computed as follows:

$$J_{\text{NLL}}(\theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{N} y_i^{(j)} \log \hat{y}_i^{(j)} \tag{2.25}$$

$$J_{\text{NLL}}(\theta) = -\frac{1}{m}\sum_{j=1}^{m} y^{(j)\top} \log \hat{y}^{(j)} \tag{2.26}$$

The other solution is to assign a value in $\{1, N\}$ to each label. If we define $id(j)$ as the index of label $y^{(j)}$, the cost function is computed as follows:

$$J_{\text{NLL}}(\theta) = -\frac{1}{m}\sum_{j=1}^{m} \log \hat{y}_{id(j)}^{(j)} \tag{2.27}$$

We define the *loss function* $\mathcal{L}(\hat{y}, y)$ as the objective function applied on a single example with label y, when the model prediction is $\hat{y}$:

- Binary cross-entropy loss:

$$y \in \{0, 1\} \quad \hat{y} \in (0, 1) \tag{2.28}$$

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log (1 - \hat{y})) \tag{2.29}$$

- Multi-class cross-entropy loss:

$$y \in \{0, 1\}^N \quad \hat{y} \in (0, 1]^N \quad \sum_{i=1}^{N} y_i = \sum_{i=1}^{N} \hat{y}_i = 1 \tag{2.30}$$

$$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{N} y_i \log \hat{y}_i = -y^\top \log \hat{y} \tag{2.31}$$

- Square loss:

$$y \in \mathbb{R} \quad x \in \mathbb{R} \tag{2.32}$$

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2 \tag{2.33}$$

Gradient descent Once a cost function is properly defined, we need to optimize this function, i.e., automatically find the set of parameters θ with the lowest cost on the train set. To do this, we can use an optimization algorithm called *gradient descent*.

It consists in repeatedly running the following *update rule* over parameters θ:

$$\theta \leftarrow \theta - \alpha \frac{\partial J(\theta)}{\partial \theta} \tag{2.34}$$

$$J(\theta) = \frac{1}{m}\sum_{j=1}^{m} \mathcal{L}(\hat{y}^{(j)}, y^{(j)}) \tag{2.35}$$

$$\theta \leftarrow \theta - \alpha \frac{1}{m}\sum_{j=1}^{m} \frac{\partial \mathcal{L}(\hat{y}^{(j)}, y^{(j)})}{\partial \theta} \tag{2.36}$$

where α is a hyperparameter called *learning rate*, or *step size*. The negative gradient $-\frac{\partial J}{\partial \theta}$ gives the direction of the update, while α specifies how much we should move in this direction.

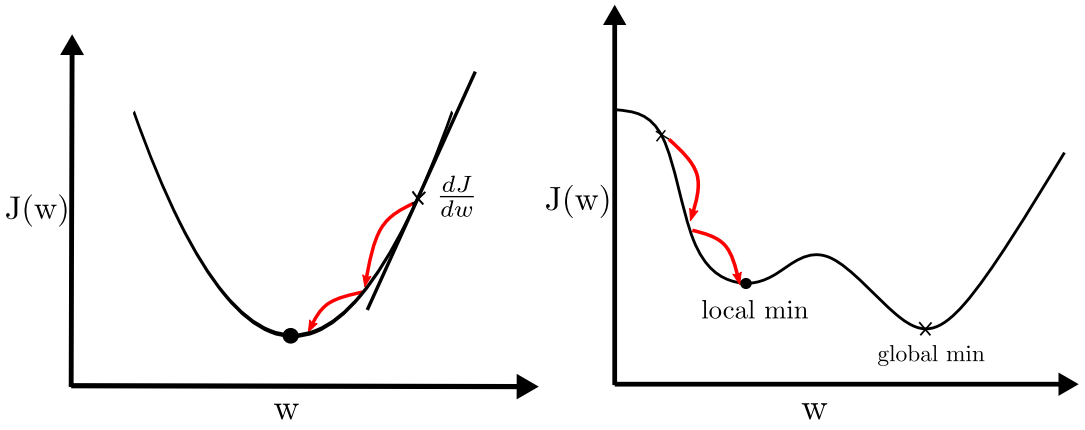
θ can be represented as a vector containing all the parameters of the model (concatenation of all flattened weight matrices and bias vectors). The gradient $\frac{\partial J(\theta)}{\partial \theta}$ has the same shape: it contains the partial derivative of the cost function with respect to each parameter. The update rule is applied element-wise.

In Figure 2.1 (A), the derivative of the cost function with respect to parameter w is positive (slope of the tangent line). When applying the gradient descent update rule (Equation 2.34), we decrease w, and get closer to the minimum value of the objective function. By applying the update rule several times, we eventually get to the global minimum.

When the cost function is convex, like Figure 2.1 (A), with the right step-size, gradient descent is guaranteed to converge to a global minimum.

However, with most neural networks the cost function is highly non-convex. Very often, it has numerous local minima (or saddle points), which are very difficult to localize. Figure 2.1 (B) shows a function with a local minimum: if the weight is initialized with a small value (left of the plot), gradient descent will probably get stuck there, and never reach the global minimum at the right of the plot.

In the figures, the models have a single parameter w. In practice, neural networks can have millions of parameters, which makes it impractical to analyze or visualize the cost function.



(A) Gradient descent of a convex cost function (B) Gradient descent of a non-convex function with a local minimum

TABLE 2.1: Gradient descent of unidimensional objective functions of different shapes

Stochastic gradient descent Batch gradient descent can be very costly, because it requires computing the gradients on the entire training set. With large neural networks and/or large training sets, this can be infeasible because of time or memory constraints.

An alternative is stochastic gradient descent, which consists in computing gradients of the loss on a single training example, and applying the update rule immediately:

$$\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}(\hat{y}^{(j)}, y^{(j)})}{\partial \theta} \tag{2.37}$$

This gradient is a noisy estimate of the true gradient, which can cause training to be quite unstable. A pass through the entire training set is called an "epoch". A single application of the update rule is called an "update", an "iteration" or a "step". Figure 1 describes the full SGD algorithm. The stopping condition can also be a pre-defined number of epochs or iterations. We can also evaluate the performance of the model on the dev set more frequently and stop training in the middle of an epoch if the model has stopped improving.

It is important to shuffle the training set before starting training, to avoid biasing the updates too much because of regularities in the training set (e.g., if we present only *cat* instances before showing dogs, the model will learn to predict cats regardless of its input). It is even recommended to re-shuffle the training set at each new epoch (Y. Bengio 2012).

Data: Shuffled training set D_{train} of size m, and dev set D_{test}
Result: Optimized model parameters θ
Initialize θ to random values;
while *dev error keeps decreasing* **do**
  foreach *example* $(x^{(j)}, y^{(j)}) \in D_{train}$ **do**
    Compute $\hat{y}^{(j)}$ using $x^{(j)}$;
    Compute $d\theta = \frac{\partial \mathcal{L}(\hat{y}^{(j)}, y^{(j)})}{\partial \theta}$;
    Apply $\theta \leftarrow \theta - \alpha \times d\theta$;
  end
  Compute error on dev set D_{test};
end

Algorithm 1: Stochastic Gradient Descent Algorithm

SGD has some interesting properties:

- Because of its noisy nature, SGD can sometimes escape local minima or saddle points, where batch gradient descent always gets stuck.
- It is less memory hungry, compared to *vectorized* batch gradient descent.
- It converges faster in most cases (it requires less epochs). This is mostly due to the redundant nature of real-world datasets. Say our training set D is extremely redundant, and contains three copies of the same dataset $\tilde{D}$. With batch gradient descent, a single epoch on D is equivalent to a single epoch on $\tilde{D}$ with a learning rate three times as large. With SGD, this is equivalent to three epochs on $\tilde{D}$.
- Incremental training: it is possible to train the model on the fly while receiving new training examples (online training), or continue training a previously trained model with new training examples (pre-training/finetuning). We can also stop training at any time and get a working model (e.g., in the middle of an epoch). This comes in handy with extremely large training sets, where we might want to stop training before the end of the first epoch.

Mini-batch gradient descent Stochastic gradient descent can be very slow, because it does not make use of parallelization, which is the main benefit of modern hardware. Moreover, with noisy training sets, SGD can be very unstable and give unpredictable results.

A variant of SGD is *mini-batch gradient descent*, where instead of doing an update for each training example, we group training examples in mini-batches of pre-defined size s:

$$\theta \leftarrow \theta - \alpha \frac{1}{s} \sum_{j=1}^{s} \frac{\partial \mathcal{L}(\hat{y}^{(j)}, y^{(j)})}{\partial \theta} \tag{2.38}$$

This combines the advantages of both batch and stochastic gradient descent: larger batch sizes are less unstable than SGD because the gradient estimate is more accurate, and we can control speed and memory usage by choosing the right batch-size. Mini-batch gradient descent is much faster, because we can compute the gradients for the entire mini-batch at once using parallelization.

There exist many strategies for creating the mini-batches. The simplest approach is to shuffle the training set at the beginning of an epoch (or only once at the beginning of training), and group consecutive examples in the same mini-batch.

In practice, this is the preferred method for training neural networks. In the literature, the term "SGD" often refers to mini-batch SGD. Common batch sizes range from 16 to 256 (in this work, we generally use 32 or 64). In Deep Neural Networks, it has been observed that large batch sizes tend to give models which do not generalize as well (Keskar et al. 2017).

Example: Logistic regression Logistic regression is like the perceptron model, except that it uses the *logistic function* (sigmoid) as activation function. This means that instead of an integer in $\{0, 1\}$, it predicts a real number in $(0, 1)$.

$$\text{input: } x \in \mathbb{R}^n \qquad \text{label: } y \in \{0, 1\} \qquad \text{output: } p \in (0, 1) \tag{2.39}$$

$$\text{parameters: } w \in \mathbb{R}^n, b \in \mathbb{R} \tag{2.40}$$

$$z = w^\top x + b \tag{2.41}$$

$$p = \sigma(z) \tag{2.42}$$

To use logistic regression as a classification model, we must couple it with a decision rule that says: if the output number p is greater than some threshold (generally 0.5), pick class 1, otherwise pick class 0. This output can be interpreted as a probability. Higher values mean higher confidence of the model that object x belongs to class 1.

In tasks where we only want to predict class 1 when absolutely certain (to reduce false positives), we can raise the threshold. Say class 1 means "individual x is cancer-free", a false positive can be catastrophic.

Another advantage of using a sigmoid instead of a hard step function like in the perceptron, is that it is differentiable, and we can use cross-entropy as the loss function:

$$\mathcal{L} = -(y \log p + (1 - y) \log (1 - p)) \tag{2.43}$$

If we want to train logistic regression with gradient descent, we must calculate the partial derivatives: $\frac{d\mathcal{L}}{db}$ and $\frac{\partial \mathcal{L}}{\partial w}$. To do so, we apply the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{d\mathcal{L}}{dz} \frac{\partial z}{\partial w} \qquad \frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{dz} \frac{dz}{db} \tag{2.44}$$

with,

$$\frac{\partial z}{\partial w} = x \qquad \frac{dz}{db} = 1 \qquad \frac{d\mathcal{L}}{dz} = p - y \tag{2.45}$$

This gives:

$$\frac{\partial \mathcal{L}}{\partial w} = (p - y) \times x \qquad \frac{d\mathcal{L}}{db} = (p - y) \tag{2.46}$$

The update rule is then straightforward:

$$w \leftarrow w - \alpha \frac{1}{m} \sum_{j=1}^{m} (p^{(j)} - y^{(j)}) \times x^{(j)} \tag{2.47}$$

$$b \leftarrow b - \alpha \frac{1}{m} \sum_{j=1}^{m} (p^{(j)} - y^{(j)}) \tag{2.48}$$

where m is either the size of the train set (batch gradient descent) or the size of the current mini-batch (SGD).

Logistic regression is a linear model. This means that it is only able to find classes that are linearly separable. To come back to the example in previous section. The decision surface is a line (or a hyperplane in higher dimensions), which gives a high bias when trying to fit datasets that are not linearly separable.

There are tricks which make possible the use of linear regression with more complex datasets. For instance, the *kernel trick* consists in doing a non-linear transformation of the input features x, resulting in a model which is not linear with respect to the initial features. Another trick is to make x very large by including many features, as a dataset may become linearly separable in higher dimensions.

Regularization Regularization is any method, which when applied during training can reduce the overfitting effects of the model, generally at the cost of increased model bias (larger train cost).

L2 regularization, also called "weight decay" in the field of neural networks is an example of regularization technique. It consists in adding a regularization term to the objective function:

$$\tilde{J}(\theta) = J(\theta) + \frac{\lambda}{2} ||\theta||_2 \tag{2.49}$$

$$||\theta||_2 = \theta^\top \theta = \sum_i \theta_i^2 \tag{2.50}$$

In the update rule, this gives:

$$\theta \leftarrow \theta - \alpha \frac{\partial J(\theta)}{\partial \theta} - \lambda \theta \tag{2.51}$$

$$\theta \leftarrow (1 - \lambda) \times \theta - \alpha \frac{\partial J(\theta)}{\partial \theta} \tag{2.52}$$

This regularization scheme tends to give model parameters of smaller magnitude. The hyperparameter λ controls the amount of regularization.

Another technique is L1 regularization:

$$\tilde{J}(\theta) = J(\theta) + \frac{\lambda}{2}||\theta||_1 \tag{2.53}$$

$$||\theta||_1 = \sum_i |\theta_i| \tag{2.54}$$

L1 regularization can lead to sparse solutions, i.e., weights with many zero values. It is often used with logistic regression to perform feature selection: when having many more features than needed, L1 regularization will drive "useless" weights to zero. At test time we can remove the corresponding features altogether, and get a more compact model.

There are also a number of techniques that result in *implicit regularization*.

A commonly used technique is early stopping. The complexity of a neural network increases during training. At the beginning of training, it generally computes very simple functions. The more training iterations, the more it learns complex patterns in the data. When plotting the training loss and dev loss with respect to training time (i.e., the number of iterations), we often observe that the training loss keeps decreasing, the dev loss decreases for a while and then starts increasing. A simple way to avoid overfitting, is to stop training at this time (when dev loss is at its lowest value).

Another way to perform implicit regularization is with data augmentation, i.e., adding more data. In some cases, we can also artificially increase the size of the training set. For example, when learning to recognize digits, we can perform random perturbations to the input images, like small rotations, or cropping. The model should be able to learn that an eight rotated by a small angle is still an eight.

2.1.4 Training Neural Networks

Backprop Back-propagation (LeCun et al. 2012; Rumelhart et al. 1988) is an algorithm for efficiently computing the gradients in a directed neural network (such as feed-forward networks).

It consists of a simple application of the *chain rule* in calculus, which breaks down the derivative of a composition of several functions as product of derivatives:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.55}$$

$$(f \circ g)' = (f' \circ g) \cdot g' \tag{2.56}$$

To apply the update rule for layer k, we want to compute the derivative of the loss function with respect to parameters $W^{[k]}$ and $b^{[k]}$. For sake of clarity, we will write these gradients as $dW^{[k]} = \frac{\partial \mathcal{L}}{\partial W^{[k]}}$, and $db^{[k]} = \frac{\partial \mathcal{L}}{\partial b^{[k]}}$

By applying the chain rule, we get:

$$dW^{[k]} = dz^{[k]} \otimes a^{[k-1]} \qquad db^{[k]} = dz^{[k]} \tag{2.57}$$

$$dz^{[k]} = \begin{cases} {g^{[k]}}'(z^{[k]}) \odot W^{[k+1]} dz^{[k+1]} & \text{if } k < L \\ \frac{\partial L}{\partial z^{[L]}} & \text{if } k = L \end{cases} \tag{2.58}$$

where $\otimes$ denotes an outer product $(x \otimes y)_{ij} = x_i \times y_j$, and $\odot$ denotes an element-wise product $(x \odot y)_i = x_i \times y_i$.
$dz^{[L]} = \frac{\partial \mathcal{L}}{\partial z^{[L]}}$ is the gradient of the loss with respect to the output of the last layer (before activation). For binary classification, where the output function is $g^{[L]} = \sigma$ and with a cross-entropy loss, we get:

$$dz^{[L]} = \hat{y} - y \tag{2.59}$$

The backpropagation algorithm consists in a forward pass, where we compute the outputs and activations of each layer, starting from the input layer, ending with the output layer. Then, in a backward pass, we start by computing the gradients of the loss with respect to the output of the last layer, and end with the input layer. Once all the gradients with respect to the model weights are computed, we can do an SGD update of the weights.

Algorithm 2 describes *backprop* for feed-forward neural networks in details. Figure 2.5 illustrates this algorithm as a computation graph, with a forward pass and a backward pass.

Data: $a^{[0]} = x$ and y, current $w^{[k]} \quad \forall k \in \{1, L\}$
Result: $dW^{[k]} \quad \forall k \in \{1, L\}$
`// forward pass`
for $k \leftarrow 1$ **to** L **do**
  Compute $z^{[k]} = W^{[k]} a^{[k-1]}$ and $a^{[k]} = g^{[k]}(z^{[k]})$;
end
`// backward pass`
Compute $dz^{[L]}$ using $\hat{y} = a^{[L]}$ and y;
Compute $dW^{[L]} = dz^{[L]} \otimes a^{[L-1]}$;
for $k \leftarrow L - 1$ **to** 1 **do**
  Compute $dz^{[k]} = g^{[k]\prime}(z^{[k]}) \odot W^{[k+1]} dz^{[k+1]}$;
  And $dW^{[k]} = dz^{[k]} \otimes a^{[k-1]}$;
end
Algorithm 2: Backpropagation algorithm applied to a feed-forward neural network. For sake of clarity, we omit the biases

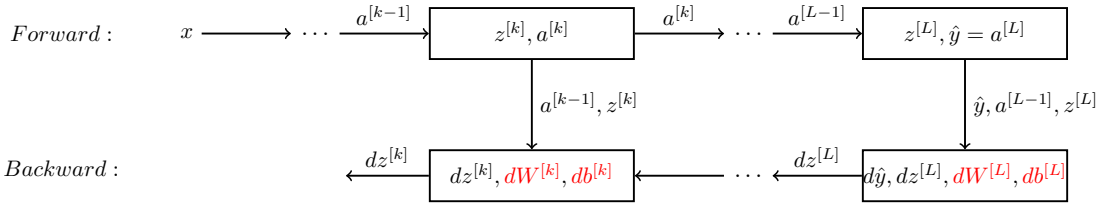


FIGURE 2.5: Backpropagation algorithm. In the forward pass, the outputs of each layer are computed one after another (they are needed for the evaluation of the loss function and of the gradients). In the backward pass, we start by computing the gradients of the loss function for the last layer of the network, and use these gradients to compute the gradients of the previous layer, and so on.

Adaptive learning rate In the gradient descent algorithm, the gradient of the loss function gives the direction in which we should move the weights to decrease the train loss. The amplitude of the update (how far in this direction we should move) is dictated in part by the magnitude of the gradient, but also by the learning rate α or *step size*.

A higher learning rate generally results in faster convergence. However, if the learning rate is too high, we can move too far in some directions and training can diverge altogether (exploding gradient). Also, with a high learning rate, when reaching the end of training (when getting close to a local minimum), the weights can bounce around the local minimum and never reach this point.

There are simple tricks with SGD: one can initialize the learning rate to the largest value possible which does not make training diverge (by trial and error), and then gradually decrease the learning rate to zero throughout training, with a method called *learning rate decay*:

$$\alpha_t = \frac{1}{1 + \gamma \times t} \times \alpha_0 \tag{2.60}$$

where γ is the *decay rate*, which controls how fast the learning rate should decay to zero. α_0 is the initial learning rate. t is the current time step, which can be the number of epochs or the number of SGD steps. Another method is exponential decay: $\alpha_t = \gamma^t \times \alpha_0$

This works quite well in practice, but still requires a lot of hand engineering to find the best initial value and the optimal decay strategy. It is also inefficient to have the same step size for all the weights, while the loss function may have a very different curvature depending on the dimension (e.g., very flat along some dimensions, and very steep along other dimensions).

Some training algorithms which are popular in neural networks are: Momentum (Sutskever et al. 2013), AdaDelta (Zeiler 2012), AdaGrad (Duchi et al. 2011), RMSProp and Adam (Kingma et al. 2015).

The idea of Momentum is the following: say the cost function that we are trying to minimize has a valley shape. It has a very flat downward slope along one dimension, and rises very abruptly on both sides along another dimension. If our initial learning rate is too high, we risk going over the cliffs and diverge. If it is too small, going down the slope can take forever. With the momentum algorithm, our updates gain "momentum" after each step, i.e., if the sign of the gradient does not change, the gradient along this dimension "accelerates" (Ng 2017; Zeiler 2012). This is often compared with a ball going down a slope: the more the ball rolls, the faster it goes, until reaching the end of the slope. This is accomplished with moving averages:

$$v_{dw} \leftarrow \beta_1 \times v_{dw} + (1 - \beta_1) dw \tag{2.61}$$

$$w \leftarrow w - \alpha v_{dw} \tag{2.62}$$

where $\beta_1 \in [0, 1)$ is a hyperparameter. Larger values correspond to more momentum (averaging over more updates), while $\beta_1 = 0$ corresponds to pure SGD (no momentum). A typical choice is $\beta_1 = 0.9$, which corresponds to averaging over ≈ 10 values. Going back to our valley example, we can now use a much larger learning rate, because the average of the gradient in the direction of the cliffs will be close to zero (alternating between large negative values and large positive values), while the average of the gradients in the direction of the slope will be positive.

RMSprop maintains a moving average of the squares of the gradients:

$$s_{dw} \leftarrow \beta_2 \times s_{dw} + (1 - \beta_2) dw^2 \tag{2.63}$$

$$w \leftarrow w - \alpha \frac{dw}{\sqrt{s_{dw}}} \tag{2.64}$$

This increases the amplitude of the update for dimensions where the gradient is consistently small (regardless of the sign), and decreases the amplitude for large gradients. Intuitively, this can be seen as some kind of normalization of the gradient, so that the same learning rate can be used in all directions.

Adam (which we often use in our work), combines both momentum and RMSprop in a single algorithm. It has four hyperparameters, α, β_1, β_2, and ϵ with default values (as specified by the authors) of $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. The default values are often fine (Adam is less sensitive to changes of hyperparameters than SGD), even though the learning rate α can sometimes be changed. It is sometimes combined with simple learning rate decay strategies, e.g., exponential decay by half every epoch.

Initialization Before starting training with SGD (or any other method), we need to initialize the parameters of the neural networks, i.e., decide of an initial value for each weight matrix and bias vector.

It is crucial to set the initial weights to non-zero values in order to break the symmetry of the neural network. If all weights are zero, then all the units in a given layer compute the same value (zero), and the gradient feedback is identical for all these units. This results in layers of n identical units, which is equivalent to layers with a single unit. That is why the weights are generally initialized to random values. The bias vectors can be (and usually are) initialized to zero.

A common practice is to initialize the weights to small values, centered around zero. This maximizes the chances of neurons firing values close to zero. This is desirable when using the hyperbolic tangent as an activation function, because we fall into the linear regime of the function (derivative close to 1), while too large values can give near-zero derivatives. On the other hand, the initial weights should not be too small, because the gradient updates are proportional to the neuron activations. A trade-off that is often used is a random initialization from a normal distribution with a zero mean and a standard deviation of 0.01.

A good initialization scheme can sometimes drastically improve convergence speed, and sometimes even leads to better local minima.

LeCun et al. (2012) recommend initializing the incoming weights of a neuron to a random distribution (uniform or normal) with mean zero, and a standard deviation of $\frac{1}{\sqrt{in}}$, where in is the number of input units for this neuron. For instance, we can initialize with a uniform distribution in $[-\sqrt{\frac{3}{in}}, \sqrt{\frac{3}{in}}]$. The advantage of this initialization scheme is that this ensures that the variance of the output of a unit is the same as the variance of its input. This helps avoiding the exploding gradient problem.

With Rectified Linear Units (ReLU) it is recommended to use a standard deviation of $\sqrt{\frac{2}{in}}$ instead (He et al. 2015; Ng 2017), because $\mathrm{Var}(\mathrm{relu}(x)) = \frac{\mathrm{Var}(x)}{2}$.

Glorot et al. (2010) propose a slightly more motivated initialization scheme, sometimes called "Glorot" or "Xavier" initialization. They show that to satisfy both constraints that the output variance be the same as the input variance (forward pass), and that the variance of the output gradient be the same as the variance of the input gradient (backward pass), we must have $\mathrm{Var}(w) = \frac{1}{in} = \frac{1}{out}$ (where in is the number of input connections to this unit, while out is the number of output connections). Because we do not always have $in = out$, we cannot always satisfy both constraints, so we meet halfway with $\mathrm{Var}(w) = \frac{2}{in+out}$. This is achieved

by uniform sampling in $[-\sqrt{\frac{6}{in+out}}, \sqrt{\frac{6}{in+out}}]$, or by sampling from a normal distribution with standard deviation $\sigma = \sqrt{\frac{2}{in+out}}$.

Another solution, which alleviates the need of a careful initialization, is batch normalization (Ioffe et al. 2015). It consists in actively normalizing the inputs of each layer in the network.

At a given layer with inputs $x_1, \cdots, x_m$, for each input x_i, batch normalization computes an estimate of its variance σ_i^2 and mean μ_i over the current mini-batch $(x_i^{(1)}, \ldots, x_i^{(D)})$. Then it normalizes each value in the batch: $\hat{x}_i^{(j)} = \frac{x_i^{(j)} - \mu_i}{\sigma_i^2}$. A problem is that by constraining the inputs to have a zero mean and a unit variance, we limit the representation power of the network: each unit is constrained to stay in its linear regime. The authors alleviate this problem by introducing new trainable parameters β_i and γ_i, where the new input value is $y_i = \gamma_i \hat{x}_i + \beta_i$. This restores the representation power of the model, because it can now learn γ_i and β_i so that batch norm computes the identity: $y_i = x_i$ (same as no batch norm). The default behavior though, with initial values of $\gamma_i = 1$ and $\beta_i = 0$ is to normalize the inputs, which increases training speed by falling into the linear regime of the (non-linear) activation functions.

When using this technique, the weight initialization scheme is less important, and a larger learning rate can be used with less risk of exploding gradient (divergence of the cost function). At test time we use statistics computed over the entire training set, by maintaining an exponentially weighted average of the mean and of the variance during training (Ioffe et al. 2015; Ng 2017).

Regularization: Dropout Dropout (Srivastava et al. 2014) is a regularization technique for neural networks, which consists in randomly dropping neurons at training time.

Each layer has a hyperparameter $p \in (0, 1]$, which is the *keep probability*, i.e., the probability of keeping each unit in this layer. We sometimes define $1 - p$ as the *dropout rate*.

Dropout can be implemented with a mask of zeros and ones of the same size as the layer. A new mask is sampled for each item in a mini-batch, from a Bernouilli distribution with probability p. The mask is then multiplied element-wise to the activations of the layer, effectively zeroing out some activations:

$$r \sim \text{Bernouilli}(p) \tag{2.65}$$

$$z = Wx + b \tag{2.66}$$

$$a = \frac{1}{p} \times r \odot g(z) \tag{2.67}$$

where $g(z) \in \mathbb{R}^n$ is the activation of this layer, $r \in \{0, 1\}^n$ is the current dropout mask, a is the resulting activation which is passed to the next layer. During back-propagation, the gradient is naturally not propagated to the canceled-out units. The activation is scaled by $\frac{1}{p}$, which ensures that the expected value of a is the same with or without dropout.

Dropout cannot be applied at test time, because it would require averaging over all the possible networks (exponentially many), which is intractable. Instead, we scale the activations at training time by $\frac{1}{p}$, and disable dropout at test time: $a = g(z)$.

Dropout is equivalent to training exponentially many "thinned" networks, with extensive weight sharing, and averaging them at test time. This has a strong regularization effect. By dropping units at random, dropout limits co-adaptation in the network. This means that a given unit in

a layer with dropout cannot rely too much on the outputs of the other units in the same layer, because they are likely to be dropped. Instead, it has to compute an interesting function on its own.

Common values for p are between 0.5 and 0.8. A rule of thumb is to apply less dropout to the first layers, and more dropout to the final layers.

2.1.5 Automatic Differentiation

Y. Bengio (2012) recommends structuring the computation in a neural network as a flow graph. We can define a neural network as a directed graph, with nodes corresponding to neurons or layers. Each node computes an operation (e.g., dot product with a weight matrix, or ReLU activation), and outputs a new value that can be passed to other nodes. Similarly, during the backward pass of back-propagation, information flows along the same edges but in the reverse direction. The advantage of defining a neural network as a graph, is that one can specify the gradient computation for each node, by defining a forward and a backward function for this node. It is then very easy to prototype new networks, by adding or removing layers, or modifying the size of a given layer. This method contrasts with a hard-coded hand computation of the gradient, which is much more tedious and prone to errors.

Deep Learning libraries like TensorFlow (Abadi et al. 2015) go one step further, by doing symbolic computation. The programmer defines a graph of symbolic expressions where each node corresponds to some operation. The graph can then be executed multiple times by substituting the input nodes with actual values. Here is how to define a simple graph in TensorFlow which takes a vector of size two as input, and computes a scalar by linear regression:

```
import tensorflow as tf
x = tf.placeholder(tf.float32, shape=[1, 2])  # input node
y = tf.placeholder(tf.int32, shape=[1])       # target for linear regression
w = tf.get_variable('w', shape=[2, 1])        # weight matrix of size [2, 1]
b = tf.get_variable('b', shape=[1])           # bias vector of size [1]
z = tf.matmul(x, w) + b                       # w x + b
cost = tf.losses.mean_squared_error(labels=y,
                                    predictions=tf.squeeze(z, axis=1))
```

Here we have defined an input node `x`, model variables `w` and `b`, and an output node `z` which is a symbolic expression with arguments `x`, `w` and `b`. The node `cost` computes a mean squared error between output node `z` and input node `y` (corresponding to a target value, which needs to be provided by the program).

The first dimension of `x`, `y` and `z` corresponds to the batch size (here we use 1, but this is usually larger).

Variables are special nodes in the sense that they are mutable: they have a value which survives between different executions of the graph. This comes in handy to implement model parameters (weights and biases) which need to be updated during training. On the other hand, at each execution of the graph new values for `x` and `y` have to be specified.

Once the graph has been defined, here is how to initialize it and run it:

```
tf.InteractiveSession()                  # create a new session
tf.global_variables_initializer().run()  # initialize w and b to rand values
```

```
x_value = [[0.81, -0.34]]                # some dummy input data
z_value = z.eval(feed_dict={x: x_value}) # compute y with given input data
```

`x`, `w`, `b` and `y` are symbolic nodes of the graph, and can be substituted with any value (of the right shape) in a call to `eval` via the parameter `feed_dict`. Here, `z_value` contains the output of node `z` (as a NumPy array) when running the graph and substituting `x` with `x_value`.

A big advantage of symbolic computation is that these frameworks can perform automatic differentiation. TensorFlow provides an operation `tf.gradients` which computes the gradients of some operation in the graph (e.g., `cost`) with respect to the model parameters (`w` and `b`). TensorFlow even provides high-level operations which directly apply the SGD update rule (or another training algorithm):

```
# define a new operation in the graph for the update rule of SGD
train_step = tf.train.GradientDescentOptimizer(learning_rate=0.5).minimize(cost)
# learn to predict 0.2 from [0.81, -0.34] (batch of size 1)
train_step.run(feed_dict={x: [[0.81, -0.34]], y: [0.2]})
```

This latter operation will update the model variables `w` and `b` by applying the SGD update rule with the defined cost function. Here we use a mini-batch of size one as a toy example, but typically the values for `x` and `y` would be multi-dimensional. Here we can see that all the complexity of gradient descent and back-propagation is hidden away by the TensorFlow library.

Thanks to this, anyone can prototype new neural network architectures, without requiring a deep knowledge of calculus, or the specifics of neural network optimization. One only has to pick a training algorithm which is known to work well (e.g., Adam) and try different hyperparameter values by trial and error.

Furthermore, such code can run very efficiently (in parallel) on specialized hardware like GPUs, with very little intervention from the user. Internally, the source code is compiled to low-level C and CUDA code by TensorFlow, which would be extremely tedious to do by hand.

To sum-up, the advantages of using a framework like TensorFlow are: automatic differentiation, fast prototyping thanks to access to high-level operations (e.g., dropout, Adam optimizer, etc.), and fast code which runs on GPUs with little to no modifications. Appendix A.2.1 gives a more detailed description of TensorFlow.

2.2 Text Embeddings

When training neural networks that take words or sequences of words as input, the first step is often to compute a continuous representation of these words. This is often done by mapping each word to a vector, using a lookup table (Y. Bengio et al. 2003). The vectors associated to words are called "word embeddings", or more rarely "distributed representations" or "word vectors".

This lookup table can be a trainable parameter of the model (i.e., initialized randomly and trained jointly with the rest of the model), or it can be pre-trained or pre-defined. The former solution is usually preferred with deep models that have access to large amounts of supervised data (e.g., Machine Translation of common language pairs). However, in many tasks, the amount of supervised training data is too small to learn so many parameters (without over-fitting). The preferred approach then is either to use hand-engineered features (or features from an engineered pipeline), as used to be very common in natural language processing; or pre-trained word embeddings on another task with more training data (transfer learning).

Word embeddings are often obtained by optimizing language modeling tasks, where the training data consists of monolingual text, available in huge amounts for many languages.

There exist many techniques for computing word embeddings, but the most popular is probably *Word2vec* (Mikolov et al. 2013c). We will focus on this particular set of techniques, while providing some context, and elaborating on a few variants and extensions. We will cover three types of embeddings: vanilla (monolingual) word embeddings as computed by Word2vec; crosslingual embeddings, i.e., word embeddings in different languages but sharing the same vector space; and continuous representations of entire sentences (sentence embeddings).

2.2.1 Word Embeddings

Distributional Semantics Most methods for computing word embeddings are based on the *Distributional Hypothesis*, which says "you shall know a word by the company it keeps" (Firth, 1957). This means that we can infer the meaning of a word by observing the words that often appear close to it. This hypothesis gave rise to a number of so-called "distributional methods" (or "count-based" methods).

These methods consist in representing words as vectors of co-occurrence counts. For example, in Latent Semantic Analysis, we construct a term-document matrix, where each entry is the number of occurrences of a given word in a given document. With this representation, each word is represented as a sparse vector (i.e., where most of the values are zero) whose size is the number of documents in the corpus. Then, matrix factorization methods like SVD can be applied on this large sparse matrix to perform dimensionality reduction, and obtain smaller (but dense) vectors. With such a representation, words can be compared by computing the cosine similarity between their vectors. Words that are similar (semantically or syntactically) will often get a high cosine similarity.

Other methods, like HAL (Hyperspace Analogue to Language) build a term-term co-occurrence matrix, where each value is the number of close occurrences of two words in a text corpus. This is obtained by sliding a fixed-size window in the text, and increasing the count for each pair of words that appears in the same window. Like LSA, the dimensionality of this representation can be reduced using SVD.

These "count-based" methods contrast with the so-called "prediction-based" methods, which consist in training a model to predict words in a text.

Neural Language Model Y. Bengio et al. (2003) introduce a "neural probabilistic language model". It consists in a neural network which, given some local context, can predict the next word in a sentence.

Given a window size n, the algorithm takes as input words $w_{t-n+1}, \dots, w_{t-1}$, and tries to predict word w_t. It computes the following functions:

$$x_t = (E(w_{t-n+1}), \dots, E(w_{t-1})) \tag{2.68}$$

$$y_t = Wx_t + U \tanh(d + Hx_t) + b \tag{2.69}$$

$$P(w_t | w_{t-1}, \cdots, w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_{i=1}^{|V|} e^{y_i}} \tag{2.70}$$

where $E \in \mathbb{R}^{|V| \times m}$ is a matrix that maps each word in the vocabulary V to a vector of size m (the lookup table introduced earlier). $x_t \in \mathbb{R}^{m(n-1)}$ is a concatenation of the input words' vectors. $W \in \mathbb{R}^{|V| \times m(n-1)}, U \in \mathbb{R}^{|V| \times h}, H \in \mathbb{R}^{h \times m(n-1)}$ and $b \in \mathbb{R}^{|V|}, d \in \mathbb{R}^h$ are trained parameters of the model, where h is the hidden layer size, and $|V|$ the vocabulary size.

An interesting thing to note, is that the matrix $E \in \mathbb{R}^{|V| \times m}$, sometimes called embedding matrix, contains a single real-valued vector of size m for each word in the vocabulary. These vectors are trained parameters of the model, which means that they are initialized randomly and then optimized to maximize the prediction accuracy of the model.

Intuitively, one can expect these vectors to contain interesting information about the words they embed, because this information is the only available data for the model to do its predictions. Y. Bengio et al. (2003) did not seem to be interested in these vectors, but rather in the predictions of the language model.

Word2vec Mikolov et al. (2013a) propose a language model which is similar to Y. Bengio et al. (2003), but much faster to train. This makes training possible on extremely large corpora of text and gives very good quality embeddings.

Contrary to Y. Bengio et al. (2003), the goal of Mikolov et al. (2013a) is not really to learn a model that can predict words, but rather the word representations that are produced as a by-product of this training. Mikolov et al. (2013a) show that the word embeddings that are learned by their models are universal enough to be used in many other NLP tasks, with little adaptation. This is an instance of *transfer learning*, as we take the parameters of a model on a given task (here language modeling), and reuse them in other (related) tasks.

The authors present two types of models: the Continuous Bag-of-Words model (CBOW), and the Skip-Gram model (SG). Figure 2.6 illustrates these two models.

CBOW takes as input a window of words (or context): $C_t = w_{t-S}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+S}$, and learns to predict the middle word w_t. Skip-gram does the reverse: it takes a single word w_t, and learns to predict each word in its context C_t.

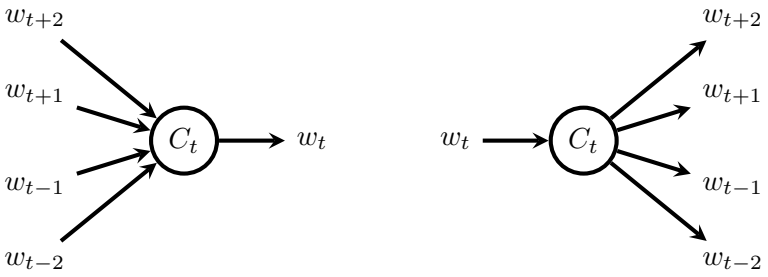


FIGURE 2.6: Illustration of the CBOW model (left), and Skip-Gram model (right). In CBOW, the input embeddings of the context words are averaged into a context vector, which is used to predict the middle word w_t. In Skip-Gram, the middle word w_t is used to predict all words in a window.

Here is how Skip-Gram predicts the probability of word j given word i (we refer to words by their index in the vocabulary):

$$z_{ik} = v_i^\top v'_k \tag{2.71}$$

$$p(j|i) = \text{softmax}(z_{ij}) = \frac{e^{z_{ij}}}{\sum_{k=1}^{|V|} e^{z_{ik}}} \tag{2.72}$$

where $v_i \in \mathbb{R}^n$ is the input embedding of word i, and $v'_k \in \mathbb{R}^n$ is the output embedding of word k. These are trained parameters of the model, which can be represented as embedding matrices $E \in \mathbb{R}^{|V| \times n}$ and $E' \in \mathbb{R}^{|V| \times n}$ where n is the embedding size, and $|V|$ is the vocabulary size. At the end-of-training, matrix E contains vector representations of each word in the vocabulary, i.e., word embeddings, which can be used in other NLP tasks.

The training loss for a given word w_t and its context window C_t is computed as follows:

$$\mathcal{L}(C_t|w_t) = -\sum_{w_k \in C_t} \log p(w_k|w_t) \tag{2.73}$$

The problem with Equation 2.72, is that a sum over the entire vocabulary $\sum_{k=1}^{|V|} e^{z_{ik}}$ needs to be computed for each word in the training corpus (times the number of training epochs). The complexity of the training algorithm is $O(|V| \times |D| \times S)$ where $|V|$ is the size of the vocabulary, $|D|$ is the number of words in the training set, and S the size of the context window. This quickly becomes infeasible with large vocabulary sizes and large training sets (which often come in pairs). Mikolov et al. (2013c) propose two different training objectives to reduce this complexity: Hierarchical Softmax, which uses a Huffman tree to encode the vocabulary, and has a complexity of $O(\log_2 |V| \times |D| \times S)$; and Negative Sampling with a complexity of $O(|D| \times S)$.

Negative sampling changes the task altogether: instead of predicting words as a multi-class classification problem (with a softmax function), we now train a logistic regression model (binary classification) to distinguish true target words (words from the context window) from randomly sampled words from a noise distribution. For each positive example (each word in the window), we draw k negative examples. The loss for a word w_t and its context C_t is computed as follows:

$$p(j|i) = \sigma(z_{ik}) = \frac{1}{1 + e^{-z_{ij}}} \tag{2.74}$$

$$\mathcal{L}(j|i) = -\log p(j|i) + \sum_{x \in neg} \log p(x|i) \tag{2.75}$$

$$\mathcal{L}(C_t|w_t) = \sum_{j \in C_t} \mathcal{L}(j|i) \tag{2.76}$$

where neg is a set of k negative examples, drawn randomly for each new word j. The probability distribution used for negative sampling is a smoothed unigram distribution estimated over the training set:

$$P(w) = \frac{\#w^{0.75}}{\sum_{w' \in V} \#w'^{0.75}} \tag{2.77}$$

where $\#w$ is the number of occurrence of w in the training set. The 0.75 exponents are arbitrary values chosen by the authors, which give a little more chance to rare words.

Algorithm 3 presents the Skip-Gram algorithm with Negative Sampling in details (SGNS). We omit the parallelism details, and invite the reader to look at the source code for more information.

Input: Subsampling rate γ, minimum vocabulary count, number of epochs, dimension of the embeddings n, initial learning rate α_0, number of negative examples k, maximum window size S_{max}
Data: Training set D, with one sentence per line ($|D|$ words in total)
Result: Word embeddings $E \in \mathbb{R}^{|V| \times n}$
Build vocabulary V and remove too infrequent tokens;
Compute smoothed unigram distribution over entire vocabulary;
`/* Repeat the following for each new epoch */`
foreach *sentence in training set* **do**
  Remove out-of-vocabulary words from sentence;
  Apply subsampling to remove random frequent tokens from sentence;
  foreach *word* w_t *in sentence* **do**
    Sample a new window size S between 1 and S_{max};
    foreach *word in context* $C_t = w_{t-S}, \ldots, w_{t+S}(\backslash w_t)$ **do**
      Sample k words from unigram distribution;
      Compute loss using Equation 2.76;
      Compute gradients with respect to E and E';
      Compute new learning rate;
      Update E and E' using SGD update rule;
    end
  end
end

Algorithm 3: Skip-Gram with Negative Sampling (SGNS). Some of the steps in this algorithm (subsampling, vocabulary filtering, dynamic window size) are detailed in Appendix A.1.1, in addition to other tricks that the authors used in their "Word2vec" implementation (e.g., asynchronous training).

Mikolov et al. (2013a) observed that word embeddings capture interesting linguistic regularities. For instance, we can observe that $v(cars) - v(car) \approx v(apples) - v(apple)$. Intuitively, the vector $v(cars) - v(car)$ embeds the concept of *plural* number. Similar observations can be

made with many other syntactic relationships (e.g., tenses, adjective/noun relationship, etc.), but even more interestingly, with many *semantic* relationships. For example, $v(king) - v(man) \approx v(queen) - v(woman)$, which seems to show that the vectors embed the concepts of $royalty$, and gender.

Following this observation, the authors propose a new benchmark for evaluating the quality of word embeddings. This task is called "analogical reasoning". Given a number of quadruplets (a, b, c, d), where "a is to b what c is to d" (e.g., king, man, queen, woman), the task consists in finding d from (a, b, c).

To find d, a solution is to compute the vector $y = v(b) - v(a) + v(c)$, and to look for the word whose vector is the closest, according to cosine similarity to y:

$$d^\star = \arg\max_{w \in V} cos(v(w), y) \tag{2.78}$$

$$cos(x, y) = \frac{x^\top y}{\|x\|\|y\|} \tag{2.79}$$

For this task, Mikolov et al. (2013d) provide a dataset with 19 558 such quadruplets, with five types of semantic relationships, and nine types of syntactic relationships. Algorithms for computing word embeddings can be compared by measuring their precision on this dataset, i.e., the number of times where $d^\star = d$ divided by the number of quadruplets.

Levy et al. (2014) observe better results on this task when using a multiplicative combination instead:

$$d^\star = \arg\max_{w \in V} \frac{cos(w, c)cos(w, b)}{cos(w, a) + \epsilon} \tag{2.80}$$

where $\epsilon = 0.001$ prevents division by zero. Here $cos(x, y)$ actually means the cosine similarity between $v(x)$ and $v(y)$. This formula only works with cosine similarities between 0 and 1. To enforce this, the authors apply $x = \frac{x+1}{2}$ on the result of Equation 2.79.

GloVe Pennington et al. (2014) present GloVe, an algorithm for computing word embeddings. They maximize a similar objective as Word2vec's Skip-Gram, but where the global statistics of the corpus are used: instead of predicting the probability of having word j in the context of word i, they predict the global co-occurrence count X_{ij}. To do so, they define a least squares objective, which they optimize with SGD by iterating over the co-occurrence matrix:

$$J = \sum_{i,j \in V} f(X_{ij})(v_i^\top v'_j + b_i + b'_j - \log X_{ij})^2 \tag{2.81}$$

$$f(x) = \begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases} \tag{2.82}$$

where $X \in \mathbb{R}^{|V| \times |V|}$ is a co-occurrence matrix, obtained by sliding a window of size $2 \times S$ in the training set, and counting the number of times word i appears in the context of word j. The function f is introduced to reduce the impact of rare co-occurrences, which tend to be noisy and carry less information. The cut-off $x_{max} = 100$ is used to avoid giving too much weight to very

frequent occurrences. This term is analogous to the smoothed unigram distribution in SGNS, with the same smoothing value $\alpha = 0.75$.

This model obtains competitive results with Word2vec's SGNS. Pennington et al. (2014) report significantly better results with their approach, on word similarity, word analogy and named entity recognition tasks, with much faster training. However, Levy et al. (2015) show on similarity and analogy tasks that SGNS consistently outperforms GloVe (albeit by a small margin).

GloVe and Word2vec are currently the two main contestants in the literature for state-of-the-art word embeddings.

Mikolov et al. (2013c) provide pre-trained models for English that were trained on a large news corpus (100B tokens). Pennington et al. (2014) distribute pre-trained models on English Common Crawl (42B or 840B tokens), Wikipedia (6B tokens), or Twitter (27B tokens).

Distributional Methods Baroni et al. (2014) compare prediction-based methods (like Word2vec) with count-based methods (like PPMI + SVD), and observe better results with prediction-based methods on a variety of tasks.

Levy et al. (2015) contradict these results by tuning count-based methods (PPMI and PPMI + SVD) to incorporate the same tricks as Word2vec, and show that these methods obtain similar performance on several tasks.

PPMI (Positive Point-Wise Mutual Information), in the context of distributional semantics consists in building a co-occurrence matrix $M^{PMI} \in \mathbb{R}^{|V| \times |V|}$, where each entry in the matrix is computed as follows:

$$PPMI(w, c) = \max(0, \log \frac{\#(w, c)|D|}{\#w\#c}) \tag{2.83}$$

where $\#w$ is the total count of word w in the training set, $|D|$ is the total number of words in the training set, and $\#(w, c)$ is the number of co-occurrences of words w and c (within a certain range). Levy et al. (2015) emulate SGNS's k negative samples by using shifted PPMI. Also, similarly to the smoothed unigram distribution used in SGNS, they smooth the context distribution in PMI (with the same $\alpha = 0.75$).

$$SPPMI_\alpha(w, c) = \max(0, \log \frac{\#(w, c) \sum_c \#c^\alpha}{\#w\#c^\alpha} - \log k) \tag{2.84}$$

The dimension of this co-occurrence matrix can be reduced using SVD. This is not necessary, but useful for obtaining compact representations. Levy et al. (2015) observe that it is actually detrimental to use the "correct" version of SVD. They obtain better results when taking $E = U_d$ instead of $E^{SVD} = U_d \cdot \Sigma_d$, where $M_d = U_d \cdot \Sigma_d \cdot V_d^\top$ is the decomposition obtained with truncated SVD.

2.2.2 Crosslingual Embeddings

In multilingual tasks, like Machine Translation or crosslingual document classification, we can be interested in using pre-trained embeddings for two (or more) languages. However, techniques

that compute embeddings, like Word2vec or GloVe, are trained on monolingual data, and only output embeddings for a single language. A naive approach would be to train monolingual embeddings for two languages on monolingual text in each language. However, such embeddings would be in independent vector spaces. Because they are initialized randomly, there is absolutely no guarantee that similar words in both languages (e.g., "cat" in English and "chat" in French, or even "lion" in English and "lion" in French) will have similar representations.

There are two kinds of approaches for training crosslingual embeddings:

- Joint training: using supervised data, like bilingual lexicons or parallel data, we can train two monolingual models jointly, so that they share the same vector space.
- Mapping: we can train two models independently on monolingual data, and then find a linear mapping between the two models (using a bilingual lexicon for example).

Joint training Luong et al. (2015a) propose a method, called *Bivec*, for learning word embeddings in two languages jointly. It does so by using a large parallel corpus. For each pair of sentence in the corpus, the sentences are aligned at the word-level, either by a GIZA-like tool (Och et al. 2003), or with a monotonous alignment. Then, it trains two Skip-Gram or CBOW models (one for each language) that interact with each other.

With regular Skip-Gram, an SGD update corresponds to training the model to predict a context word $w_k \in C_t$ from an input word w_t. In Bivec, we have two models with their own embeddings and vocabularies V^1 and V^2, and each training step considers two (aligned) pairs of words $w_t^1 \in V^1, w_t^2 \in V^2$ and $w_k^1 \in V^1, w_k^2 \in V^2$.

Each training step corresponds to four SGD updates: one monolingual update and one crosslingual update for each model. Monolingual updates correspond to predicting w_k^1 from w_t^1 (or w_k^2 from w_t^2). Crosslingual updates correspond to predicting w_k^2 from w_t^1 (and updating each model accordingly), and similarly with w_k^1 from w_t^2.

Gouws et al. (2015) propose a similar method, which trains two monolingual models jointly, by summing their training loss. They add a crosslingual loss term to the training objective, which is a L2 loss between the bag-of-words sentence vectors of sentence pairs in a parallel corpus. Unlike (Luong et al. 2015a), this method can use large monolingual texts for training, with a smaller parallel corpus to enforce the alignment between the vector spaces.

Mapping Given two sets of independently trained word embeddings, in two different languages, Mikolov et al. (2013b) propose a technique to learn a mapping between the two vector spaces. They do so by using a bilingual lexicon D, and looking for (with SGD):

$$W^\star = \arg\min_W \sum_{i=1}^{|D|} \|Wx_i - z_i\|^2 \tag{2.85}$$

where $|D|$ is the size of the lexicon and (x_i, z_i) is the i$^{\text{th}}$ word pair from the lexicon. $W \in \mathbb{R}^{n \times n}$ maps x_i to the same vector space as z_i.

Artetxe et al. (2017) propose a technique which surpasses all previous methods, and which is almost entirely unsupervised. It consists in building an initial bilingual lexicon containing only numerals, then similarly to Mikolov et al. (2013b), learn a mapping from one vector space to

the other using this lexicon. Once this mapping is obtained, they build a larger bilingual lexicon (using the trained embeddings), and learn a better mapping. By iterating like this until convergence, Artetxe et al. (2017) obtain a good quality mapping between two embedding spaces, with very little supervision (only the assumption that both languages use the same numeral system).

Such embeddings are used in later work by the same authors (Artetxe et al. 2018), to do unsupervised Neural Machine Translation (i.e., MT with no parallel data), with very encouraging results.

Conneau et al. (2017b) propose a similar method, which does not need any supervision at all (not even digits), and gives better results than Artetxe et al. (2017) on a word translation task. It builds a first mapping by doing *adversarial training*, where a generator learns to map from one embedding space to the other, and a discriminator learns to predict from which embedding space a given vector originates. Then, this mapping is refined using an iterative method similar to Artetxe et al. (2017).

2.2.3 Sequence Embeddings

Methods like Word2vec or GloVe learn word embeddings, i.e., representations of individual words. However, many tasks in NLP do not only take single words as input, but entire sequences of words of variable length.

Bag of words The most basic approach for representing sequences of words like sentences or documents, is to sum the representations (one-hot vector, or word embedding) of each word in the sequence. This results in a fixed size representation. Given a sentence $s = w_1, \cdots, w_T$, and a word embedding matrix $v : w \rightarrow \mathbb{R}^n$, the representation of s is trivially computed as follows:

$$v_s = v(w_1) + v(w_2) + \ldots + v(w_T) \tag{2.86}$$

There are some problems with this approach. First, all words are weighed equally, i.e., stop words like "the" weigh as much as content words. Also, this representation is invariant to word order, e.g., $v_{w_1 w_2} = v_{w_2 w_1}$. This can be problematic as two sentences with a different word order can have a very different meaning.

Surprisingly, this simple method often obtains very good results on downstream NLP tasks, and more sophisticated methods have a lot of trouble doing better. This can be explained by the fact that the word embeddings that are used are generally trained on very large corpora of texts, and are thus of very good quality. Furthermore, in many NLP tasks, we can get decent results without actual language understanding.

Ferrero et al. (2017) apply a weighting scheme on word embeddings, which attributes constant pre-defined weights to words depending on their part-of-speech tags. More important words in a sentence get higher weights, resulting in a better representation overall. They get a significant improvement on a plagiarism detection task.

Paragraph vector Q. V. Le et al. (2014) propose a method called "paragraph vector", which extends Word2vec's CBOW model to produce representations of entire sentences or paragraphs.

For each sentence (or sequence of words) in the training set $s^i = w_1^i, \ldots, w_T^i$, the PV-DM (Paragraph Vector Distributed Memory) model learns to predict each word w_t^i using its context $C_t^i = w_{t-s}^i, \ldots, w_{t+s}^i \backslash w_t^i$, and an additional trained bias vector b^i which is unique to the current sentence:

$$z_t^i = (b^i + \sum_{w \in C_t^i} v(w))^\top v'(w_t^i) \tag{2.87}$$

z_t^i is the unnormalized score for word t in sentence i. Like Word2vec's CBOW and Skip-Gram model, this value is given as input to a softmax layer to estimate a probability distribution over the entire vocabulary, or to logistic regression to distinguish between positive and negative examples. In addition to CBOW's input and output embeddings (v and v'), this model has an additional bias parameter $b^i \in \mathbb{R}^n$, which is unique per sentence s^i. At the end of training, this vector can be used as a representation for sentence s^i. Intuitively, because the window C_t does not cover the entire sentence, we can expect b^i to provide contextual information about the entire sentence, which helps the model predict w_t.

There are two modes for obtaining the representation of a given sentence:

- Batch (offline) mode, where all sentences are already available at training time. After training, a representation b^i is available for each sentence in the training set.
- Online mode (which the authors call "inference stage"): given a new sentence that was not used in training, a pre-trained model can be used and finetuned on this new sentence. To do so, all model parameters except the bias parameter b^i are frozen (i.e., we do not update them with SGD), and we iterate using SGD on the words of the new sentence multiple times until convergence.

The authors report remarkably good results on a sentiment analysis task. However, it was reported later in Mesnil et al. (2014), when their source code was made available, that they were due to a mistake in the experiments: the datasets were not shuffled, which made it extremely easy for their model to distinguish between positive and negative examples.

Thought vectors Kiros et al. (2015) propose an unsupervised method called "Skip-Thought" for computing universal representations of sentences. The idea is similar to Skip-Gram, but instead of predicting single words, this model predicts entire sentences.

The model is trained on a large corpus of *ordered* sentences, made of 11,038 books (74M sentences in total). Given a sentence, the model learns to predict the previous sentence and the next sentence in an ordered text.

To do so, the model first encodes the input sentence using a Recurrent Neural Network (RNN), a type of neural network that we will see in the next chapter. This encoder reads the sentence word by word, encodes each word using a trained embedding matrix, and updates its recurrent state step-by-step until reaching the end of the sequence. The final state of the encoder is a fixed-size representation of the variable-size input sequence. This representation is used as input by two decoders (one RNN each) to predict the previous sentence (word by word) and the next sentence. The training objective is a sum of cross-entropy losses for each word in the previous

and next sentence. By optimizing this sentence-prediction task, the encoder learns to compute useful representations of the input sequence that can be used in downstream NLP tasks.

To validate the quality of the sentence representations, the authors perform a number of experiments on NLP transfer tasks, including Semantic Relatedness (figuring out whether two sentences are semantically related or not), paraphrase detection (given a pair of sentences, decide whether or not they are paraphrases), and several classifications tasks like sentiment analysis.

The Skip-Thought model is pre-trained on books, and its features are then used as input to a logistic regression model trained on the given task (with supervised data). These benchmarks confirm the universality of the skip-thought vectors, showing that they transfer to many different tasks with little supervision (only a linear model).

InferSent Conneau et al. (2017a) propose a sentence embedding model which is trained on a supervised task: Natural Language Inference. This task consists in predicting whether two sentences are entailments, contradictions, or "neutral".

The authors obtain better results than Skip-Thought on several NLP tasks, and with much faster training time and less training data (570k *supervised* sentences vs. 74M *unsupervised* ordered sentences).

This approach shows that it is possible with some supervised tasks to obtain sentence representations that are universal enough to apply to other NLP tasks. This contrasts with observations from Hill et al. (2016), who observed poor quality embeddings (non-transferable) with several supervised tasks, including Neural Machine Translation.

Chapter 3

Sequence to Sequence Models

Now that we have introduced feed-forward networks and how to train them, we are going to explore a particular class of artificial neural networks that is particularly useful for dealing with sequences (e.g., text sequences). We will first introduce these recurrent neural networks, and then describe the most popular neural network for machine translation at the time of writing: seq2seq. We will also present some of the improvements that have been proposed in the literature and which we use in this thesis, in particular the so-called *attention mechanisms*.

3.1 Recurrent Neural Networks

As we have seen in the previous chapter, feed-forward neural networks are limited to fixed-size inputs.

To deal with variable-length inputs, like sequences of words, we generally have to make strong assumptions: use only local context with fixed-size windows of text (e.g., Word2vec); limit the length of the input to some maximum size, and pad shorter inputs with dummy symbols; or assume that the order of words does not matter, and sum the embeddings of each word in the input sequence (bag-of-words).

The solution that is largely preferred for dealing with sequential inputs (or sequential outputs) is Recurrent Neural Networks (or RNNs). They naturally adapt to any input length, and can keep contextual information about their input (a sort of memory).

3.1.1 Vanilla RNN

Definition The Vanilla RNN, or Elman Network (Elman 1990), illustrated in Figure 3.1, takes as input a sequence of vectors $x_1, \dots, x_T \in \mathbb{R}^{T \times m}$. The RNN is composed of a state $s_t \in \mathbb{R}^n$, which is updated at each new input x_t, and encodes information about past inputs. The state of an RNN is updated as follows:

$$s_t = \tanh(W_{rec} s_{t-1} + W_{in} x_t + b) \tag{3.1}$$

where $W_{rec} \in \mathbb{R}^{n \times n}, W_{in} \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$ are trainable parameters of the RNN. Depending on the implementation, the length T of the input can be fixed or variable (static RNN vs.

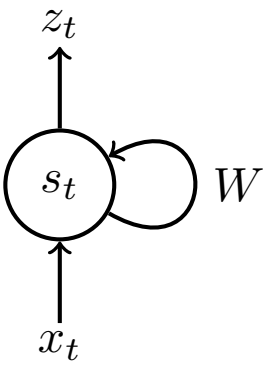


FIGURE 3.1: Vanilla RNN with state s_t, input x_t and output z_t.

dynamic RNN). When doing mini-batch SGD, we group several inputs in the same batch. To parallelize computation (vectorization), we pad all points in a batch to the maximum length in the batch, so that the entire batch can be stored as a tensor of shape (b, T, m), where b is the batch size.

The initial state s_0 can be set to zero, or it can be a trainable parameter of the model.

Use cases The states of the RNN can be understood as representations of the input sequence at each time step, (s_t encodes sequence $x_1, \ldots, x_t$). This representation can be used inside a feed-forward neural network, or another recurrent neural network to compute functions of the input.

Here are some examples of Natural Language Processing tasks where RNNs are useful:

- Language Modeling (LM), i.e., predicting the next symbol in a text sequence. It can be achieved as follows:

$$z_t = W_{voc} s_t + b_{voc} \tag{3.2}$$
$$p_t = \mathrm{softmax}(z_t) \tag{3.3}$$
$$\mathcal{E} = \sum_{t=1}^{T} \mathcal{L}(p_{t-1}, x_t) \tag{3.4}$$

  $W_{voc} \in \mathbb{R}^{|V| \times n}$ is a projection to the target vocabulary size. $p_t \in \mathbb{R}^{|V|}$ computes a probability for each word in the vocabulary. The error $\mathcal{E}$ is the sum of the losses for the prediction of each word x_t in the sequence (given history $x_1, \ldots, x_{t-1}$). $\mathcal{L}$ is typically a cross-entropy loss. Some examples of similar tasks (with a prediction at each time step) are Part-of-Speech tagging or Word Sense Disambiguation.

- Sentiment Analysis usually consists in predicting a rating (e.g., 1 or 0: positive or negative) for a text sequence (e.g., a movie review). Contrary to language modeling a single scalar is produced after reading the entire input sequence.

$$z = w_{out}^{\top} s_T + b_{out} \tag{3.5}$$
$$r = \sigma(z) \tag{3.6}$$
$$\mathcal{E} = \mathcal{L}(r, y) \tag{3.7}$$

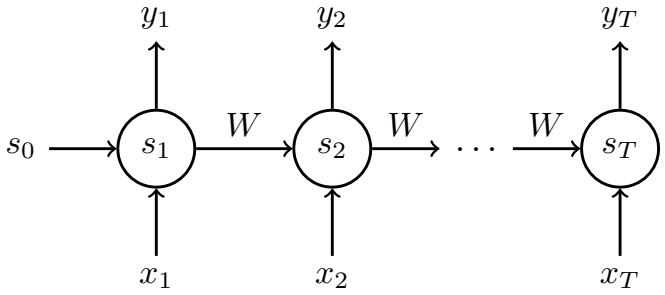


FIGURE 3.2: Unrolled RNN: similar to a feed-forward network, where each layer corresponds to a time step.

with $w_{out} \in \mathbb{R}^n$ and $b_{out} \in \mathbb{R}$. The error $\mathcal{E}$ is the loss between the predicted rating r and the target rating y. If the rating is categorical (positive or negative), $\mathcal{L}$ is a cross-entropy loss. If the rating is continuous (between 0 and 1), $\mathcal{L}$ is typically a squared loss. Some examples of similar tasks (where a single label is predicted) are document classification, spam detection or language identification.

- Machine Translation or Speech Recognition consist in reading a variable-length sequence and predicting another variable-length sequence (usually with a different length). This case can be addressed by using an encoder-decoder architecture, where one RNN is used to encode the input sequence, and another RNN uses this representation to output a new sequence. We will study this specific case in the next section.

The same optimization techniques as with feed-forward networks (SGD) apply to RNNs, which take the gradients of the error function with respect to the model parameters.

3.1.2 Backpropagation Through Time

Backpropagation Through Time (BPTT) is the application of backprop to recurrent neural networks. It consists in unrolling the RNN (illustrated in Figure 3.2) as if it were a feed-forward network with as many layers as there are time steps. Then, backprop can be applied to propagate the gradients from the last time step to the first time step.

The main difference with feed-forward networks is that all 'layers' share the same parameters (weights and bias). Figure 3.3 illustrates backpropagation through time. The error feedback $\frac{\partial \mathcal{E}_t}{\partial s_t}$ at a given time step is back-propagated to the earlier time steps, so that the contributions of all previous inputs $x_1, \ldots, x_t$ to the error term $\mathcal{E}_t$ are taken into account.

TensorFlow proposes two kinds of implementations of RNNs: static RNNs, which have a maximum number of time steps T and are statically unrolled when compiling the graph. This is implemented as a feed-forward network with T layers and hard weight sharing. The implementation is straightforward, however compilation time can be large, and this is not very flexible because the number of time steps is fixed. TensorFlow also provides dynamic RNNs, which are "dynamically" unrolled, i.e., the number of time steps does not need to be known until runtime.

Vanishing and Exploding gradient Vanilla RNNs suffer from some severe drawbacks:

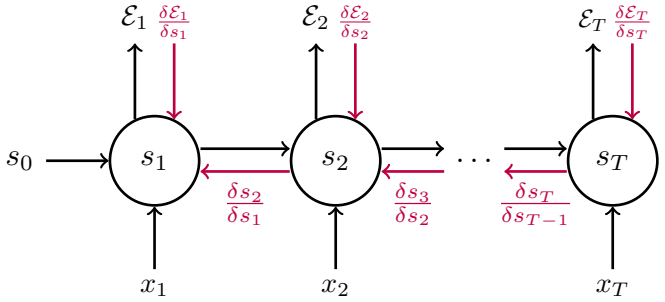


FIGURE 3.3: Backpropagation Through Time in a statically unrolled vanilla RNN. $\mathcal{E}_1, \dots, \mathcal{E}_T$ are the errors computed at each time step (in some tasks we only get $\mathcal{E} = \mathcal{E}_T$), and $\frac{\partial\mathcal{E}_1}{\partial s_1}, \dots, \frac{\partial\mathcal{E}_T}{\partial s_T}$ are the corresponding gradients.

- They are unable to store long-term information. Because the entire state s_t is updated at each time step, new information tends to quickly overwrite previously stored information.
- They are notoriously difficult to train (Hochreiter et al. 1997; Pascanu et al. 2013). Once unrolled, they are like extremely deep neural networks, which causes problems of vanishing and exploding gradient.

To illustrate the latter point, let's detail the computation of the gradients with BPTT.

The gradient of the error $\mathcal{E}$ with respect to the model parameters $\theta = (W_{rec}, W_{in}, b)$ can be decomposed as follows (Pascanu et al. 2013):

$$\mathcal{E} = \sum_{t=1}^{T} \mathcal{E}_t \tag{3.8}$$

$$\frac{\partial\mathcal{E}}{\partial\theta} = \sum \frac{\partial\mathcal{E}_t}{\partial\theta} \tag{3.9}$$

Let's consider only the error term $\mathcal{E}_t$. With the chain rule, we obtain:

$$\frac{\partial\mathcal{E}_t}{\partial\theta} = \sum_{1\leq k\leq t} \frac{\partial\mathcal{E}_t}{\partial s_t}\frac{\partial s_t}{\partial s_k}\frac{\partial^+ s_k}{\partial\theta} \tag{3.10}$$

where $\frac{\partial^+ s_k}{\partial\theta}$ is the partial derivative of s_k with respect to θ when keeping s_{k-1} constant:

$$\frac{\partial^+ s_k}{\partial\theta} = (\frac{\partial^+ s_k}{\partial W_{rec}}, \frac{\partial^+ s_k}{\partial W_{in}}, \frac{\partial^+ s_k}{\partial b}) = (1 - s_k^2)\begin{cases} s_{k-1} \\ x_k \\ 1 \end{cases} \tag{3.11}$$

The Jacobian matrix $\frac{\partial s_t}{\partial s_k}$ is computed as follows (Pascanu et al. 2013):

$$\frac{\partial s_t}{\partial s_k} = \prod_{k<i\leq t} \frac{\partial s_i}{\partial s_{i-1}} = \prod_{k<i\leq t} W_{rec} diag(1 - s_i^2) \tag{3.12}$$

where $diag$ converts its vector parameter (of size n) into a diagonal matrix (of size $n \times n$).

Figure 3.3 illustrates this product of gradients in the backward pass of BPTT. This is the main culprit for the vanishing and exploding gradient. Let's imagine that we are doing sentiment analysis of movie reviews. We read a sequence where the k$^{\text{th}}$ word is "boring" (very negative) and all other words are neutral. We get a single error signal $\mathcal{E}_T$ after reading the entire sequence. If our prediction is that this movie was excellent (predicted rating r close to 1), then the model parameters need to be updated in order to obtain better predictions (the movie was obviously not excellent). The contribution of word x_k ("boring") to the error $\mathcal{E}_T$ appears in the gradient as term $\frac{\partial \mathcal{E}_T}{\partial s_T}\frac{\partial s_T}{\partial s_k}\frac{\partial^+ s_k}{\partial \theta}$ (see eq. 3.10). The computation of $\frac{\partial s_T}{\partial s_k}$ involves a product of $T-k$ terms (see eq. 3.12). If W_{rec} has small values and $T-k$ is large, then $\frac{\partial \mathcal{E}_T}{\partial W_{rec}}$ will be close to zero. This means that W_{rec} does not change at the next SGD update, and we are stuck in an endless loop (vanishing gradient). On the other hand, if W_{rec} has large values, then $\frac{\partial \mathcal{E}_T}{\partial W_{rec}}$ can take exponentially large values, which makes W_{rec} diverge to infinity (exploding gradient).

More intuitively, in the mono-dimensional case, we get $\frac{\partial s_T}{\partial s_k} = w_{rec}^{T-k}\prod_{i=k}^{T}(1-s_i^2)$. If $w_{rec} < 1$, the exponential term w_{rec}^{T-k} quickly vanishes to zero as $T-k$ increases, and quickly explodes to infinity if $w_{rec} > 1$. Pascanu et al. (2013) show that this can also happen in the high-dimensional case.

Gradient clipping Pascanu et al. (2013) proposed a solution to the exploding gradient problem in recurrent neural networks. It consists in clipping the gradients so that their norm does not exceed some preset threshold δ:

$$g \leftarrow \frac{\partial \mathcal{E}}{\partial \theta} \tag{3.13}$$

$$\|g\| \leftarrow \sqrt{\sum_i g_i^2} \tag{3.14}$$

$$g \leftarrow g \times \frac{\delta}{\max(\|g\|, \delta)} \tag{3.15}$$

where g is a flattened list of all gradients (i.e., all gradient tensors in $\frac{\partial \mathcal{E}}{\partial \theta}$ are flattened and concatenated). This gradient clipping strategy works well in practice, and is often enough to negate the exploding gradient problem. The hyperparameter δ can be chosen by observing statistics of the average norm of the gradients. But Pascanu et al. (2013) have found that training is not very sensitive to choices of δ (provided that it is small enough to prevent the gradients from exploding).

3.1.3 Long-Short-Term Memory

Description We have seen that vanilla RNNs are particularly sensitive to the vanishing and exploding gradient problems, which makes them very difficult to train; even more so with long sequences, or when the time-lag between events and the corresponding error feedback is large. They also have trouble storing information over long periods of time.

The Long Short-Term Memory or LSTM (Hochreiter et al. 1997) has been shown to be more robust to longer sequences. Figure 3.4 illustrates the difference between LSTMs and vanilla

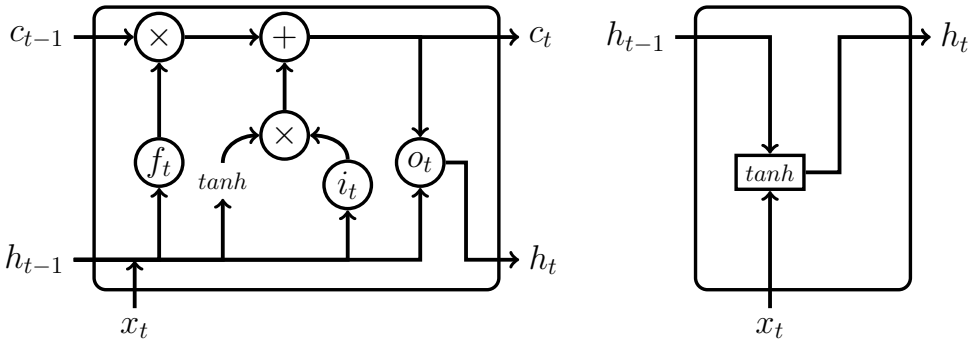


FIGURE 3.4: Illustration of an LSTM cell (left) and a Vanilla RNN cell (right). c_t is the state of the LSTM and h_t is its output. Vanilla RNNs use their state as output. This illustration originates from Christopher Olah's blog ("Understanding LSTM Networks")

RNNs. Instead of updating its internal state at each time step, and risking to overwrite important information, LSTMs control information flow with several *gates*:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{3.16}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{3.17}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{3.18}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{3.19}$$
$$h_t = o_t \odot \tanh(c_t) \tag{3.20}$$

where $U_f, U_i, U_o, U_c \in \mathbb{R}^{n \times n}$ and $W_f, W_i, W_o, W_c \in \mathbb{R}^{n \times m}$ and $b_f, b_i, b_o, b_c \in \mathbb{R}^n$ are trainable parameters of the model. The $\odot$ operator computes an element-wise product.

Equations 3.16, 3.17, 3.18 describe respectively the *forget gate*, the *input gate* and the *output gate*. Equation 3.19 describes how the internal state c_t of the LSTM is updated, and Equation 3.20 describes how the output of the LSTM is computed.

The forget vector $f_t \in (0, 1)^n$ controls how much information from the previous state c_{t-1} passes through to the new state. The input vector i_t controls how much new information (from input x_t) should be let through. And finally, the output vector o_t controls how much information from the current state should be made available as the LSTM's output. Contrary to RNNs, the internal state c_t is not available to the outside of the cell. Only the output of the LSTM h_t is used in other parts of the model (e.g., to make new predictions). Compared to vanilla RNNs, LSTMs have four times as many parameters, and their state is twice as large (we need to store c_t and h_t).

There are two main ingredients to the success of the LSTM (compared to RNNs):

- The so-called "Constant Error Carousel" (Hochreiter et al. 1997), which is illustrated as a straight line between c_{t-1} and c_t in Figure 3.4. We see in term $c_t = f_t \odot c_{t-1} + \cdots$ from eq. 3.19, that there is no non-linear transformation of the previous state. Contrary to RNNs, the gradient is not multiplied by W_{rec} at each time step (see eq. 3.12), which can make the gradient vanish or explode.[1] For this reason, LSTMs suffer much less from the vanishing gradient problem than vanilla RNNs. Thanks to the constant error carousel

[1] Even though it is multiplied by f_t, this is less problematic as f_t is often very close to 1 (except for dimensions we want to forget), and can take different values at each time step.

(not exactly constant since the forget gate was introduced, by Gers et al. (1999), but close enough), the gradient flows more easily to earlier time steps, and it becomes easier to learn long-term dependencies.

- The gates control information flow, and help the LSTM store information on longer periods of time. Thanks to the output gate, the LSTM can remember information in its state for later use, and only output it when the time is right. The forget gate makes it possible for the cell to forget information that it does not need anymore.

Gated Recurrent Units GRUs (Cho et al. 2014a) are a popular alternative to LSTMs. They are similar to LSTMs but the input and forget gate are merged into a single *update gate* (z_t). The output gate is removed, and replaced with a slightly different *reset gate* (r_t), which controls how much of the previous state should be used in the computation of the input. Contrary to LSTMs, GRUs directly output their state.

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{3.21}$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{3.22}$$
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{3.23}$$

Because it has fewer gates, and a smaller state (it uses its own internal state as output), the GRU has less parameters and is slightly faster than the LSTM. There is no general consensus as to which one is better.

3.2 Sequence to Sequence Model

Recurrent neural networks can read a sequence of vectors or symbols, and compute a representation of this sequence which is useful to a prediction task. An example of such task is language modeling: given the start of a sentence, predict the next word.

In this section, we are interested in sequence to sequence prediction tasks. This means that we want to predict variable-length sequences, *conditioned* on another variable-length sequence. Machine Translation, Speech Recognition/Translation, and Automatic Post-Editing all fall into this category.

Sutskever et al. (2014) propose a general framework for sequence to sequence prediction. It consists of two RNNs: an encoder, which reads the input sequence; and a decoder which predicts an output sequence. This framework, illustrated by Figure 3.5, is the basis for most of the subsequent contributions in Neural Machine Translation. This section presents this basic model, then we describe some extensions. We focus on the Machine Translation task, even though this kind of model can be used for any "sequence to sequence" task.

3.2.1 Description

Notations The training set consists in a list of $(\mathbf{x}, \mathbf{z})$ pairs, where $\mathbf{x} = x_1, \ldots, x_T$ is a source sequence of length T; and $\mathbf{z} = z_1, \ldots, z_{T'}$ is a target sequence of length T'. Given $\mathbf{x}$ we learn to predict $\mathbf{z}$.

V and V' are the source and target vocabularies, and $|V|$ and $|V'|$ their respective size. $E \in \mathbb{R}^{|V| \times m}$ and $E' \in \mathbb{R}^{|V'| \times m}$ are the source and target embedding matrices. These matrices are initialized randomly and trained jointly with the other parameters of the model.

Source and target symbols shall be identified by their index in the vocabulary, i.e., $x_i = j$ means that x_i is the j$^{\text{th}}$ word in V. $E(j) \in \mathbb{R}^m$ is the embedding vector of symbol j (the j$^{\text{th}}$ row in matrix E).

Encoder The encoder is an RNN, which reads the entire input sequence $\mathbf{x}$ and updates its state:

$$h_i = \text{update}_{enc}(h_{i-1}, x_i) \tag{3.24}$$

h_i is the state of the RNN cell at time step i. The initial state h_0 is initialized at random, or it can be a trained parameter of the model. update_{enc} is the transition function of the RNN (which can be any type of cell, e.g., LSTM or GRU). With a vanilla RNN we get:

$$h_i = \tanh(W_{rec} h_{i-1} + W_{in} E(x_i) + b) \tag{3.25}$$

where $W_{rec} \in \mathbb{R}^{n \times n}$, $W_{in} \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$ are trained parameters of the model.

The final state of the encoder $h_T \in \mathbb{R}^n$ is a fixed-size representation of the input sequence. This representation hopefully contains enough information about $\mathbf{x}$ to help the decoder predict $\mathbf{z}$. The decoder's state is initialized with this representation: $s_0 = \tanh(W_{init} h_T + b_{init})$. Alternatively, if both the encoder and the decoder have the same cell size, the decoder's state can be directly initialized with the encoder's state: $s_0 = h_T$.

There is a subtlety when implementing the update function for LSTMs. LSTMs actually produce two vectors at each time step: a state c_i and an output h_i. Both the output and the state are used by the LSTM to update its state. But only the output h_i is visible from the outside of the LSTM. For sake of brevity, we omit the c_t parameter in the update formulae. The implied formulation for LSTMs is:

$$h_i, c_i = \text{update}(h_{i-1}, c_{i-1}, x_i) \tag{3.26}$$

Decoder The decoder is another RNN, which predicts a sequence of output symbols $\hat{z}_1, \ldots, \hat{z}_{T''}$ as follows:

$$s_t = \text{update}_{dec}(s_{t-1}, E'(\hat{z}_{t-1})) \tag{3.27}$$

$$y_t = \text{generate}(s_t \oplus E'(\hat{z}_{t-1})) \tag{3.28}$$

$$\hat{z}_t = \arg\max_{i=1}^{|V'|}(y_{ti}) \tag{3.29}$$

where $\oplus$ is the concatenation operator ($x \oplus y = [x_1, \ldots, x_n, y_1, \ldots, y_m]$). $\hat{z}_0 = \texttt{BOS}$ is a special beginning-of-sentence symbol. During training, the output length T'' is the same as target length T'. At evaluation time — where target sequence $\mathbf{z}$ is not available and T' is unknown — the decoder stops updating its state and predicting symbols once it has produced a special end-of-sentence symbol: $\hat{z}_{T''} = \texttt{EOS}$. Note that for the model to learn when to output this special symbol, the target sequences should always end with $z_{T'} = \texttt{EOS}$.

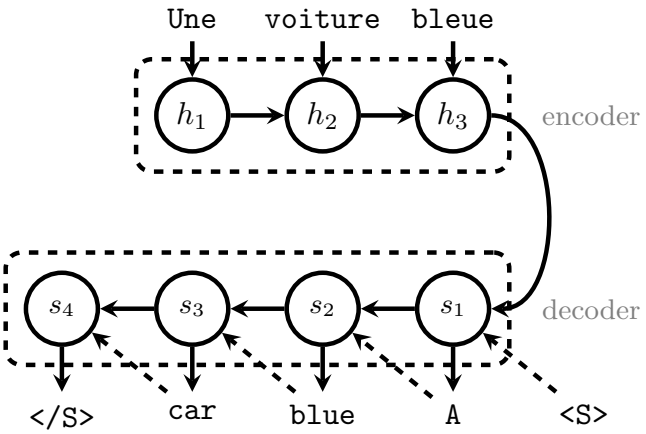


FIGURE 3.5: Illustration of a basic "sequence to sequence" model (Sutskever et al. 2014). A first RNN (the encoder) reads the input sequence, and a second RNN (the decoder) outputs a new sequence. The decoder is initialized with the final state of the encoder, which is a fixed size representation of the input sequence. The decoder stops when reaching a special end-of-sequence symbol.

$\mathrm{generate}$ takes as input the current decoder state s_t and maps it to a vector y_t of size $|V'|$, which contains a score for each symbol in the target vocabulary. The decoder outputs the symbol $\hat{z}_t$ with the highest score. update_{dec} can be the transition function of any RNN cell (e.g., GRU or LSTM). With a vanilla RNN and a simple linear projection, we get:

$$s_t = \tanh(W'_{rec} s_{t-1} \oplus W'_{in} E'(\hat{z}_{t-1}) + b') \tag{3.30}$$

$$y_t = W_{voc} s_t + b_{voc} \tag{3.31}$$

where $W'_{rec} \in \mathbb{R}^{n \times n}$, $W'_{in} \in \mathbb{R}^{n \times m}$, $b' \in \mathbb{R}^n$, $W_{voc} \in \mathbb{R}^{|V'| \times n}$, and $b_{voc} \in \mathbb{R}^{|V'|}$ are trained parameters of the model. See Figure 3.5 for an illustration of this encoder-decoder architectures.

3.2.2 Loss Function

For the model to make useful predictions, it has to encode useful information about the input sequence $\mathbf{x}$ into the encoder's hidden state $\mathbf{s}$. Indeed, the only information that the decoder has access to is the last hidden state of the encoder.

The probability of target sequence $\mathbf{z}$ given source sequence $\mathbf{x}$ according to the model is:

$$P(\mathbf{z}|\mathbf{x}) = \prod_{t}^{T'} p(\hat{z}_t = z_t|\mathbf{x}) \tag{3.32}$$

$$p(\hat{z}_t = j|\mathbf{x}) = \mathrm{softmax}(y_{tj}) = \frac{e^{y_{tj}}}{\sum_{k=1}^{|V'|} e^{y_{tk}}} \tag{3.33}$$

where y_{tj} is the j$^{\text{th}}$ element of vector y_t. We minimize a cross-entropy objective $\mathcal{L}$ over an entire batch $\mathcal{D}$ of sentence pairs, which is the average of the log-probabilities of each target sentence

given the source sentence:

$$loss(\mathbf{x}, \mathbf{z}) = -\log P(\mathbf{z}|\mathbf{x}) = -\sum_{t}^{T'} \log p(\hat{z}_t = z_t|\mathbf{x}) \tag{3.34}$$

$$\mathcal{L} = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},\mathbf{z})\in\mathcal{D}} loss(\mathbf{x}, \mathbf{z}) \tag{3.35}$$

3.2.3 More Details

Teacher forcing In practice, during training we always feed the target (ground truth) symbol z_{t-1} to the decoder instead of the previously generated symbol $\hat{z}_{t-1}$, i.e., Equation 3.27 becomes:

$$s_t = \text{update}_{dec}(s_{t-1}, E'(\widetilde{z}_{t-1})) \tag{3.36}$$

$$\widetilde{z}_{t-1} = \begin{cases} z_{t-1} & \text{if training} \\ \hat{z}_{t-1} & \text{if decoding} \end{cases} \tag{3.37}$$

where $\widetilde{z}_0 = \texttt{BOS}$. This technique is called *teacher forcing* (Williams et al. 1989).

Without teacher forcing, training is difficult. The outputs of the decoder are conditioned on its previous outputs. Once it does a mistake, it can quickly diverge from the target translation, which makes the later word-level feedback irrelevant.

Even though it helps with training, teacher forcing can cause problems at test time. The model is accustomed to being shown "perfect" inputs. Once it starts doing mistakes, it can quickly degenerate to rubbish output, because it is fed with sequences it has never seen during training and does not know how to handle. Some contributions propose alternative training methods that expose the decoder to noisy input (S. Bengio et al. 2015; Goyal 2016; Ranzato et al. 2016).

Reversed input Sutskever et al. (2014) found that reversing the source sequence, such that $\mathbf{x} = x_T, \ldots, x_1$ helped achieve better scores.

The authors explain this phenomenon by the fact that this shortens the minimum gradient path, by bringing the first symbols of the source sequence closer to the first output symbols.

The decoder acts as a language model: its current state gives a representation of the output sequence up to the current word. Once it has generated a few words, it becomes easier for the decoder to generate new words by looking at this representation. However, the first output words are harder to predict, because there is no context except the source sentence. Making the first source words "closer" (by reading them last, and keeping a fresh memory of them) may help with predicting these first few output words, which are crucial for generating the next words. This probably only works because there is generally some kind of monotonicity in the problems we consider (e.g., in English to French translation, the first target words are likely to correspond to the first source words).

Multi-layer encoder and decoder To achieve state-of-the-art results in a Neural Machine Translation application, Sutskever et al. (2014) use LSTM units — which are better at learning long-range dependencies than vanilla RNNs.

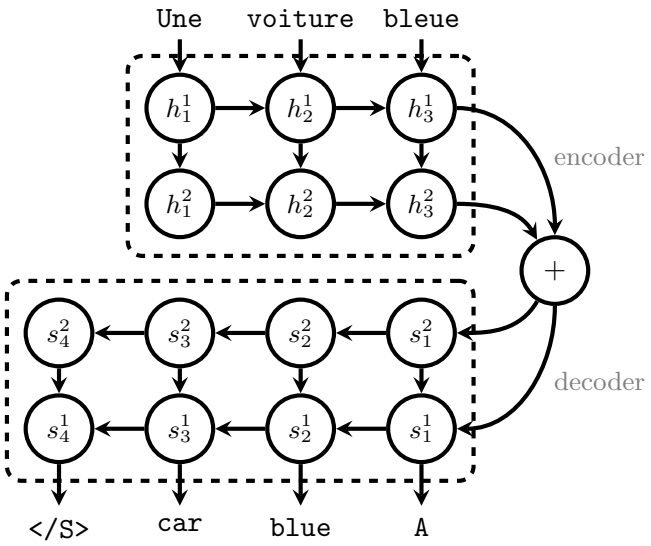


FIGURE 3.6: Illustration of a sequence to sequence model with a 2-layer encoder and a 2-layer decoder. The encoder's first RNN reads the input sequence, and a second RNN reads the outputs of the first RNN. The final states of the encoder are concatenated and used to initialize the decoder. The decoder's first layer reads the previously generated symbols. The second layer reads the outputs of the first layer and generates new symbols.

They achieve the best results by using a deep encoder and a deep decoder, which stack multiple layers of LSTMs. This is best illustrated by Figure 3.6. For example, a two-layer encoder can be defined as follows:

$$h_i^{[1]} = update_{enc}^{[1]}(h_{i-1}^{[1]}, E(x_i)) \tag{3.38}$$

$$h_i^{[2]} = update_{enc}^{[2]}(h_{i-1}^{[2]}, h_i^{[1]}) \tag{3.39}$$

The decoder can be initialized with the last state of the last layer $h_T^{[2]}$. It can also use a concatenation of the last states of all encoder layers:

$$s_0^{[1]} = \tanh(W_{init}^{[1]}(h_T^{[1]} \oplus h_T^{[2]}) + b_{init}^{[1]}) \tag{3.40}$$

3.3 Attention Models

In the vanilla sequence to sequence model, the encoder reads the entire source sequence and computes a fixed-size representation $h_T \in \mathbb{R}^n$. It is difficult to encode an arbitrarily large amount of information into a fixed-size vector. Bahdanau et al. (2015) show that this kind of model produces worse results as the input length increases.

Also, the decoder is only initialized with this fixed-size representation, and does not have access to it afterwards. This means that the computation path and the gradient path (between the first input word and the last output word) have a length of $T + T'$. Even though LSTMs are known for being good at handling long-range dependencies, and do not suffer much from the vanishing gradient problem, the length of the gradient path does have a non-negligible impact.

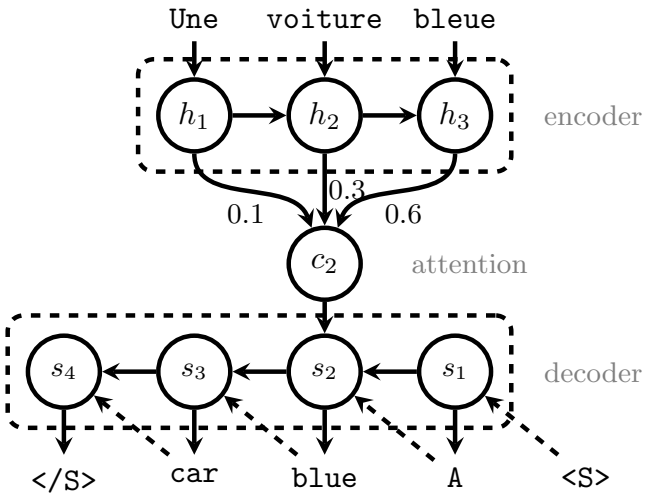


FIGURE 3.7: Illustration of a sequence to sequence model with attention. Instead of just initializing the decoder with the last state of the encoder, the decoder can look anywhere in the sequence of hidden states using an attention mechanism. At each time step, this attention model generates a context vector which summarizes the input sequence, depending on the current state of the decoder. This context vector can be used to update the state of the decoder (input feeding) and to help generate a new symbol.

A solution to this problem is to use an *attention model*, which allows the decoder to look at the input sequence at any time.

3.3.1 Global Attention

Attention-based decoder Bahdanau et al. (2015) propose to enhance the decoder with an attention mechanism. This model is illustrated by Figure 3.7. Equations 3.27 and 3.28 are modified as follows:

$$c_t = \text{look}(s_{t-1}, (h_i)_{i=1}^{T}) \tag{3.41}$$

$$s_t = \text{update}_{dec}(s_{t-1}, E'(\widetilde{z}_{t-1}) \oplus c_t) \tag{3.42}$$

$$y_t = \text{generate}(s_t \oplus E'(\widetilde{z}_{t-1}) \oplus c_t) \tag{3.43}$$

$$\hat{z}_t = \arg\max_{i=1}^{|V'|}(y_{ti}) \tag{3.44}$$

The decoder uses its current state s_{t-1} to look at the sequence of encoder hidden states $h_1, \dots, h_T$ and compute a context vector $c_t \in \mathbb{R}^n$. This context vector is used by the decoder to update its state and to generate its next output symbol.

The c_t argument to the update_{dec} function (eq. 3.42) is optional. This method is called "input-feeding" by Luong et al. (2015b), and supposedly helps the decoder remember where it has already looked. Bahdanau et al. (2015) also use this input feeding approach in their RNNsearch model, even though they do not name it.

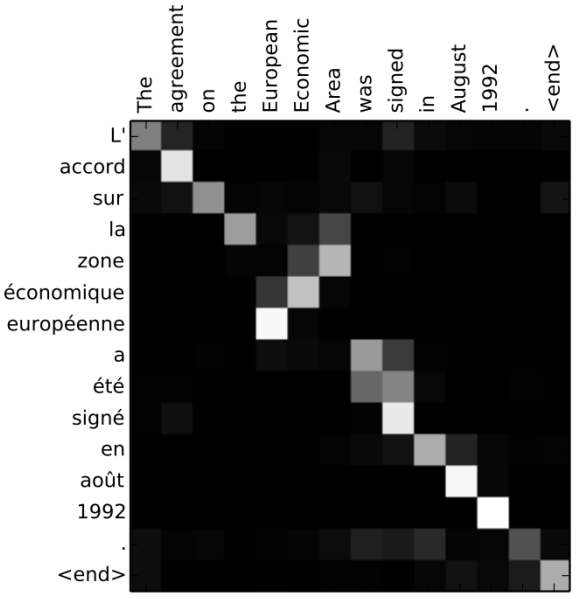


FIGURE 3.8: Example of alignment from (Bahdanau et al. 2015) obtained with their RNNsearch model. The x-axis and y-axis correspond to the source and target sequence (respectively). Each square shows the attention weight α_{ti} (black $= 0$, white $= 1$) between output word t and input word i (or rather the corresponding hidden state h_i).

Attention function The look function consists in a feed-forward network that takes as input the current state of the decoder s_{t-1} and predicts a weight for each of the encoder's hidden states h_i:

$$c_t = \mathrm{look}(s_{t-1}, (h_i)_{i=1}^T) = \sum_{i=1}^{T} \alpha_{ti} h_i \tag{3.45}$$

$$\alpha_{ti} = \mathrm{softmax}(r_{ti}) = \frac{e^{r_{ti}}}{\sum_{k=1}^{T} e^{r_{tk}}} \tag{3.46}$$

$$r_{ti} = v_{att}^{\top} \tanh(W_{att}(h_i \oplus s_{t-1}) + b_{att}) \tag{3.47}$$

where $v_{att} \in \mathbb{R}^k$, $W_{att} \in \mathbb{R}^{k \times 2n}$ and $b_{att} \in \mathbb{R}^k$ are trained parameters of the model. The $\mathrm{softmax}$ function ensures that the weights α_{ti} are a probability distribution over the input length, i.e., $\sum_i^T \alpha_{ti} = 1$ and $\forall i, \alpha_{ti} \in (0, 1]$. Other attention models differ as to how they compute the scores r_{ti}. Some take additional information as input (Cohn et al. 2016); or use a different aggregation function, like multiplicative attention (Luong et al. 2015b). The additive attention model described here is the most commonly used one, and we call it "global attention" (as it can look anywhere), or "vanilla attention".

The parameters of the attention model are trained jointly with the rest of the model, so as to minimize the translation loss. Very often, this results in an informative attention model, which will put more weight on the input words (or more precisely their hidden state h_i) that are useful for predicting the next word $\hat{z}_t$. Intuitively, this results in a soft-alignment between the input sequence and the output sequence. Figure 3.8 shows an example of such alignment from (Bahdanau et al. 2015).

Multiplicative attention Luong et al. (2015b) propose a similar attention mechanism, which uses a different aggregation method (product instead of concatenation) for computing scores r_{ti}:

$$r_{ti} = h_i W_{att} s_{t-1} \qquad W_{att} \in \mathbb{R}^{n \times n} \tag{3.48}$$

Bidirectional encoder Bahdanau et al. (2015) use a bidirectional encoder. Instead of a single RNN, two RNNs which read the input sequence in both directions (from left to right, and from right to left) are stacked on top of each other. The encoder hidden states h_i are a concatenation of the outputs of both RNNs:

$$\vec{h}_i = \text{update}_{fwd}(\vec{h}_{i-1}, E(x_i)) \tag{3.49}$$

$$\overleftarrow{h}_i = \text{update}_{bwd}(\overleftarrow{h}_{i+1}, E(x_i)) \tag{3.50}$$

$$h_i = \vec{h}_i \oplus \overleftarrow{h}_i \tag{3.51}$$

where $\vec{h}_0$ and $\overleftarrow{h}_{T+1}$ are initialized with zeros, or with $\vec{h}_{init} \in \mathbb{R}^n$ and $\overleftarrow{h}_{init} \in \mathbb{R}^n$, which are trained parameters of the model.

This removes all concern about the direction in which the input sequence should be read. Another advantage is that all encoder states contain context information about the entire sequence: $\vec{h}_i$ encodes information about all words up to x_i, and $\overleftarrow{h}_i$ encodes information about all words from x_i to x_T. It should be noted that while each encoder state h_i encodes the entire sentence, there is a stronger focus on current word x_i and its immediate context, due to the fact that RNNs remember short-term information better.

3.3.2 Local Attention

Global attention models can be computationally expensive. At each time step t, the decoder has to compute a score over all T encoder hidden states. This results in a decoding complexity of $O(T \times T')$, while the vanilla (attention-less) sequence to sequence approach has a complexity of $O(T + T')$.

Luong et al. (2015b) propose a different attention mechanism, which only looks into a fixed-size window of encoder states, thus achieving a linear complexity w.r.t. output size T'.

It first uses current decoder state s_{t-1} to compute a position p_t in the input sequence where it should look:

$$p_t = T \times \sigma(v_{att}^\top \tanh(W_{att} s_{t-1})) \tag{3.52}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.53}$$

The logistic function σ returns a value between 0 and 1, which tells how close to the start (0) or to the end (1) of the input sequence the decoder should look. This value is multiplied by the length of the input sequence to get the absolute position p_t.

Now, instead of considering all the hidden states, the attention mechanism only looks at the hidden states within the window $\mathcal{W} = [p_t - d, p_t + d]$, where d is a hyperparameter of the model.[2] This reduces the complexity of the attention model from $O(T \times T)$ to $O(2d \times T)$.

For any position k outside of the window, $\alpha_{tk} = 0$. The other weights α_{ti} are computed as follows:

$$\alpha'_{ti} = \frac{e^{r_{ti}}}{\sum_{k \in \mathcal{W}} e^{r_{tk}}} \tag{3.54}$$

$$\alpha_{ti} = \alpha'_{ti} \times \exp(-\frac{(i - p_t)^2}{2\sigma^2}) \tag{3.55}$$

Similarly to Equation 3.45, a probability distribution is estimated over the hidden states by using the $\mathrm{softmax}$ function. The main difference is that we do not consider the entire sequence of hidden states, but only the states within the attention window. Then, these weights are multiplied by a Gaussian distribution centered around p_t. This gives more weight to the central hidden state h_{p_t}, and less weight to the states which are further away from position p_t.[3] The standard deviation σ is set empirically to $d/2$.

3.4 Various Improvements

In addition to the attention mechanism, a number of other improvements have been made to the vanilla sequence to sequence model.

3.4.1 The Unknown Word Problem

One weakness of Neural Machine Translation models, which was highlighted by Bahdanau et al. (2015), is the so-called *unknown word problem*.

Table 3.1 shows results obtained with an NMT system compared to SMT. We see that the NMT system is far behind, except when we remove all test sentences with unknown words in them (in either source or target side), in which case it gets slightly better than SMT. This shows that NMT is far more sensitive to unknown symbols than SMT.

There are two reasons for this. First, there are much more unknown words in the first place. In SMT, the phrase table stores probabilities for each phrase that was seen at least once (there is no limit to the number of such entries). In NMT, for computational efficiency, we use a shortlist of the most frequent tokens (generally between 30k and 100k), and transform any other symbol to `UNK` (special unknown word token).

But more importantly, NMT models are unable to rewrite the input words like SMT does. At all times, an SMT system knows which word(s) it is currently translating. If such word(s) do not appear in the phrase table, it just copies them in the output. It is very useful in the case of proper nouns or numbers for example.

[2] In (Luong et al. 2015b), for English-German word-based MT, d is empirically set to 10.

[3] This term is important as it is the only way for the model to learn to predict the position p_t (i.e., learn the parameters W_{att} and b_{att}). Otherwise, $\frac{\partial \mathcal{L}}{\partial W_{att}}$ is zero.

Model	All	No UNK
RNNsearch (NMT)	28.45	36.15
Moses (SMT)	33.30	35.63

TABLE 3.1: Table from Bahdanau et al. (2015). BLEU score of their NMT approach on WMT 2014 $en \rightarrow fr$ test set, compared to the SMT baseline. The "No UNK" column corresponds to the BLEU scores when removing all test sentences with unknown words.

Furthermore, NMT learns a representation of the input sequence. If the input is polluted by to many `UNK` symbols, the model won't be able to learn a useful representation.

Sampled softmax One of the reasons why there is a lot of `UNK` symbols in the training and evaluation data, is the size of the vocabulary. Indeed, for performance reasons, a vocabulary shortlist is often used — of 30k source words and 30k target words in Bahdanau et al. (2015). Using the entire training vocabulary (i.e., all the words that appear at least once in the train set), is generally infeasible. Jean et al. (2015b) propose a modification of the softmax layer in the decoder and of the loss function, so that the training complexity does not depend on the vocabulary size. This sampled softmax method trains using a fixed-size subset of the entire vocabulary at each time step. As a result, only the parameters corresponding to the target word, and a set of *negative* words are updated.

Replacement of unknown words Another solution, which seems obvious when one knows how SMT works, is to provide the NMT system with a similar mechanism for recopying unknown words. Instead of producing an `UNK` symbol, we could look in the input sequence and find out which word we are translating, and then just copy this word.

Jean et al. (2015b) propose to use the global attention model to identify which word is being translated (the word with the highest attention weight), and either recopy this word, or find a translation in a large dictionary.

This technique can be used conjointly with the *sampled softmax* method. However, as Jean et al. (2015b) noted, the gain for doing so is reduced, because the larger the vocabulary, the less the model sees `UNK` symbols at training time, and the less likely it is to output them at test time.

Luong et al. (2015c) propose techniques which do not rely on the attention mechanism. Their most successful technique is called *PosUnk*. It starts by aligning the source and target side of the training set with an unsupervised aligner (e.g., GIZA++). Then, instead of replacing unknown target words with a single `UNK` symbol, we find the relative position of the source word to which they are aligned. There is a special unknown word token for each such relative position: `UNK(-1)` if the aligned word is one word to the left, `UNK(0)` if it is at the exact same position, etc. By pre-processing the training data as such, the decoder learns to output special unknown word symbols which carry positional information. Then, as a post-processing step, we can replace each symbol by the source word at the relative position.

As a side note, while these methods are able to remove `UNK` symbols from the output; they do not solve the problem of having too many `UNK` symbols in the input. Having many such symbols in the input sequence hinders the model, as it may be unable to learn a correct representation of the source sequence. This often results in gibberish output, or entire sequences of `UNK` symbols, in which case a post-processing solution is not enough.

Subword units Another way of solving this problem is to use smaller units than words. Sennrich et al. (2015) propose an elegant solution based on *Byte Pair Encoding*. The vocabulary is comprised of the most frequent subword units in the train set. A subword unit can be anything from a single character to an entire word. A special symbol is appended to the subwords which do not correspond to word suffixes (e.g., _), so that we know to concatenate them with the next subword (e.g., `hog_` and `warts` form `hogwarts`).

This allows the model to read and generate any word. This is useful with agglutinative languages like Finnish or Turkish; or with inflected languages, like Spanish or French. In the case of inflections, a subword corresponding to the word stem, and a subword corresponding to the inflection are likely to be in the vocabulary. The inflected word can be formed by concatenating those (e.g., `hav_` and `ing`). In case of agglutination, words can be formed by a concatenation of affixes. For example, `neun_`, `hundert_`, `und_` and `elf` give `neunhundertundelf` (nine hundred eleven in German). The subword vocabulary also contains the most frequent syllables in the vocabulary (e.g., `ing`, `in`, `the`, etc.), which makes it easy to form any word in this language rather efficiently. In the worst case scenario, words can be formed by a concatenation of their letters (e.g., numbers, foreign words or misspelled words).

The algorithm for finding these subword units is iterative. It starts with a list of all single characters in the train set (which allow by themselves to reconstruct any unknown word). Then, it looks for the pair of existing character n-grams whose concatenation is the most frequent (for example `e + d -> ed` is very frequent). It continues to do so until it has reached a number of n-gram pairs (e.g., 30k), or when it cannot find new pairs. The result is a list of merge operations, which take two frequent n-grams and form a longer n-gram.

Once this list is obtained, words can be transformed into subwords by first splitting them into a sequence of their constituent characters (initial n-grams), and then successively, and greedily, applying merge operations (the most common ones first) on pairs of n-grams, until no such merge is possible.

Say our sorted list of merge operations is as follows:

```
1. t </w>
2. s t</w>
3. l u
4. e st</w>
5. b lu
```

Here is an example with `bluest` (`</w>` marks the word boundary):

```
b l u e s t </w>
b l u e s t</w>
b l u e st</w>
b lu e st</w>
b lu est</w>
blu est</w> = blu_ est
```

The result is a segmentation of `bluest` into `blu_` and `est`. Another advantage of this method is that the generated vocabulary also contains entire words, at least the most common ones. As a result, the average sentence length does not change much, which can be desirable as the model's time and memory complexity depends on the length of the input and output sequences.

3.4.2 Improve Decoding

Beam search decoder At test time, we would like to find the output sequence $\mathbf{w}^\star$ with the highest probability according to the model:

$$\mathbf{w}^\star = \arg\max p(\mathbf{w}|\mathbf{x}) \tag{3.56}$$

$$p(\mathbf{w}|\mathbf{x}) = \prod_t^{|\mathbf{w}|} p(w_t|w_1, \dots, w_{t-1}, \mathbf{x}) \tag{3.57}$$

$$p(w_t = j|w_1, \dots, w_{t-1}, \mathbf{x}) = \text{softmax}(y_{tj}) = \frac{e^{y_{tj}}}{\sum_{k=1}^{|V'|} e^{y_{tk}}} \tag{3.58}$$

However, exploring the entire space of hypotheses is infeasible (since there are $|V'|^T$ possible hypotheses). Surprisingly, the greedy solution often gives very good results. It consists in decoding from left to right, and picking at each time step the token with the highest score:

$$\hat{\mathbf{w}} = \hat{w}_1, \dots, \hat{w}_T \tag{3.59}$$

$$\hat{w}_t = \arg\max p(\cdot|\hat{w}_1, \dots, \hat{w}_{t-1}, \mathbf{x}) = \arg\max_{i=1}^{|V'|} y_{ti} \tag{3.60}$$

Another approach, which gives slightly improved results is to use a beam search algorithm. At each time step, we keep the n best hypotheses, and expand these hypotheses at the next time step.

There are several variants of the beam search decoder. For efficiency reasons, we may want to stop looking for new hypotheses once enough finished hypotheses have been found. A hypothesis is considered as finished if it contains the end-of-sentence token. After the pruning step, if the current beam contains a finished hypothesis, we do not need to continue expanding this hypothesis. It is removed from the set of running hypotheses (and added to a set of finished hypotheses), and the maximum beam size is reduced by one.

One common problem with beam search decoding is that it tends to favor shorter sentences. Because the score of a hypothesis is the sum of the log probabilities of all its words, a longer sentence will often have a lower score. A popular solution is to do length normalization, i.e., normalize the scores of the hypotheses by a value which depends on their length (Wu et al. 2016):

$$\text{lp}(\mathbf{w}) = (\frac{1+\mu}{|\mathbf{w}|+\mu})^\eta \tag{3.61}$$

$$\text{score}(\mathbf{w}) = \text{lp}(\mathbf{w}) \log p(\mathbf{w}) \tag{3.62}$$

where η and μ are hyperparameters of the model, and $|\mathbf{w}|$ is the length of sequence $\mathbf{w}$.

When $\eta = 0$, then $\text{lp} = 1$ this corresponds to no length normalization. In the simplest case with length normalization, $\eta = 1$ and $\mu = 0$, which corresponds to $\text{lp} = 1/|\mathbf{w}|$. Parameter μ is often set to 0.

These normalized scores are only used at the end of beam search decoding, for rescoring the finished hypotheses (generally n of them).

A coverage penalty can also be used (Wu et al. 2016):

$$\text{cp}(\mathbf{x}, \mathbf{w}) = \beta \times \sum_{i}^{T} \log(\min(\sum_{t}^{|\mathbf{w}|} \alpha_{ti}, 1.0)) \tag{3.63}$$

$$\text{score}(\mathbf{w}) = \text{lp}(\mathbf{w}) \log p(\mathbf{w}) + \text{cp}(\mathbf{x}, \mathbf{w}) \tag{3.64}$$

where α_{ti} is the alignment probability between source word x_i and output word w_t, as computed by the attention model. β is a hyperparameter of the model, which controls the strength of the coverage penalty factor.

While the attention mechanism ensures that an output word's total probability mass is 1 (because of the softmax normalization), some source words may very well align to no output word at all, i.e., $\exists i, \sum_t^{|\mathbf{w}|} \alpha_{ti} \approx 0$. This coverage penalty favors hypotheses where each source word's total probability mass is high, i.e., hypotheses where most source words are covered by the attention model.

Chorowski et al. (2016) propose a similar coverage penalty term, and hypothesize that this may help the decoder avoid looping endlessly over the same outputs. It is a frequent failure mode in NMT. When the decoder has a high confidence in a group of tokens, it sometimes outputs these tokens again and again. By favoring hypotheses where each source word has a translation, we may avoid such edge cases.

Wu et al. (2016) test different values for η and β. They empirically set μ to 5, and use a beam size of 8. They found that on a large WMT14 $en \rightarrow fr$ task, coverage penalty and length normalization are important to obtaining the best results. They settle with $\eta = 0.2$ and $\beta = 0.2$, which gives the best BLEU score on the dev set (1 point above the baseline without length normalization and coverage penalty). Note that $(\eta, \beta) = (0, 1)$ achieves the same BLEU score (no length normalization, and maximum coverage penalty); and that $(\eta, \beta) = (1, 0)$ achieves almost the same score (full length normalization, no coverage penalty).

In SMT, because the language model and translation model consider only short range dependencies, the beam search decoder is crucial to obtaining decent translations. The chosen beam size is generally the highest possible, while keeping a good decoding speed. In Moses for instance, the default beam size is 100. In NMT, the beam search decoder improves over a greedy decoder, but only slightly (by at most one or two BLEU points). The size of the beam is most often between 2 and 12. Larger beam sizes are almost always detrimental.

The state of the decoder carries information about the entire past output, and it can look anywhere in the input sequence. This gives enough context to the decoder to make an informed decision about which word to generate, without worrying about future words. Also, the model is trained with greedy decoding in mind, i.e., it is optimized at the word level; and during training we feed the decoder with its past greedy outputs. This may be the reason why beam search decoding does not perform so well, and can even be detrimental with higher beam sizes.

Language model In SMT, the noisy-channel model allowed us to combine a translation model with a language model (Brown et al. 1993). Later, a more general log-linear model was proposed, which combined any number of features whose weights could be tuned (Koehn 2010).

The translation model focuses on maximizing translation adequacy, but could alone result in very crude output (with low fluency). On the other hand, the language model ensures that the

output is fluent in the target language, without having to worry about the source sentence. Both models specialize in two different things, and combining them can help recover from the errors of individual models. One other advantage of this approach is that the language model can be trained on larger amounts of monolingual data, which can be very helpful when little parallel data is available to train the translation model.

It is very tempting to use a similar approach in NMT. We can combine the translation model that we described in the previous sections, with a language model trained on monolingual data. However, the benefit is less obvious than in SMT. Indeed, the decoder itself acts as language model, because it learns a probability distribution over sequences in the target language (the probability of a word is conditioned on the previously generated words). Still, the encoder-decoder is trained with parallel data, and an external language model is a way to incorporate more (monolingual) data.

The beam search decoder proceeds as before, but estimates the log probability of a word given the hypothesis' past words as follows:

$$\log p(w_t|w_1, \dots, w_{t-1}, \mathbf{x}) = \log p_{MT}(w_t| \dots) + \gamma \times \log p_{LM}(w_t|w_{t-2}, w_{t-1}) \tag{3.65}$$

where p_{LM} is the conditional probability of a word given the previous two words as estimated by a trigram language model. γ is a hyperparameter which puts more or less emphasis on the language model.

This technique is described as "shallow fusion" by Gulcehre et al. (2015). In this work, an external neural language model is used.

In Bahdanau et al. (2016) an n-gram language model is integrated in the same way. However, the decoder is at the character-level. They transform the word-level language model into a character-level one by using a weighted finite state transducer. In this instance (speech recognition), the external language model was essential to obtaining good results.

Ensembles Most state-of-the-art results are obtained with ensembles of models (Luong et al. 2015c; Sutskever et al. 2014; Wu et al. 2016; Zhou et al. 2016). This generally consists in training a number of instances of the same model (typically between 4 and 8), and averaging their log-probabilities.

In a beam search decoder, the log-probability of a word is computed as follows:

$$\log p(w) = \sum_{i}^{N} \lambda_i \log p_i(w) \tag{3.66}$$

where $p_i(w)$ is the softmax probability of word w according to the i$^{\text{th}}$ model in the ensemble, and N is the size of the ensemble. By default, $\lambda_i = \frac{1}{N}$. However, these weights can be tuned on the dev set, with tools like MERT (Och 2003), so as to give more weight to models which are the most beneficial to the ensemble. In the context of NMT, ensembles generally bring a large improvement to evaluation scores (often several BLEU points). Because the training algorithm is stochastic (different weight initialization, train set shuffling), different models can have very different weaknesses and strengths. Averaging these models will likely average out the errors of single models.

Sometimes, when it is too expensive to re-train new models from scratch, one may use different checkpoints of the same training instance (provided they are not too close); or take the same checkpoint and finetune it in N new training instances (Jean et al. 2015b).

Another slightly different method — which may be use conjointly with ensembles — is to take different checkpoints of the same training instance, and average all their parameters element-wise. Of course, this does not work with checkpoints from different training instances (like ensembles), as training is stochastic and averaging parameters won't result in meaningful values. The gain from using this method is consistent, but often smaller than using ensembles (Junczys-Dowmunt et al. 2016b; Sennrich et al. 2017).

The log-linear formulation from Equation 3.66 can be used with any combinations of models — provided that they have the same target vocabulary. One can combine several instances of the same model with different models altogether (e.g., language models).

3.5 New NMT models

At the time I started writing this thesis, RNNsearch (Bahdanau et al. 2015) was arguably the most popular neural machine translation model. A variant of this model, namely GNMT has been deployed in Google Translate (Wu et al. 2016), and now achieves considerably better results than their previous statistical machine translation model. All of our work is based on the RNNsearch model and variants of it.

Yet, new models have been proposed recently that perform even better than RNNsearch, with significantly lower training times. The most promising models for general machine translation are now ConvS2S (Gehring et al. 2017b) and Transformer (Vaswani et al. 2017).

These two models use the same "encoder-decoder" structure as seq2seq, but do away with recurrent neural networks. While RNNs are very expressive and could theoretically encode any type of sequential information, they are slow to train due to two main reasons: they do not parallelize very well because of their autoregressive nature,[4] and they suffer from the vanishing gradient problem.

ConvS2S uses many convolutional layers to encode the input and output sequence. Its encoder produces a sequence of states of the same length as the input sequence, where each state encodes information about most of the sequence (because at each layer, convolutions aggregate information from nearby positions). The decoder can look at any of these states thanks to an attention mechanism. Transformer uses many layers with self-attention, where each layer can look anywhere in the previous layer. Contrary to RNNs these two models are unable to compute positional information. For this reason, positional embeddings are also given as input (in addition to word embeddings). These two models also use careful initialization or layer normalization (Ba et al. 2016) to stabilize training, along with residual connections (He et al. 2015) to facilitate gradient flow.

Thanks to teacher forcing (because we do not have to wait for the model's previous prediction to train a particular decoder time step), these models can be trained in a non-autoregressive manner, which largely speeds up training. Furthermore, the gradient can flow very quickly from any position in the last decoder layer to any position in the first encoder layer. This makes it possible to increase the depth of the network, and hence improve its expressivity without

[4] Computation at a given time step depends on computations at the previous time steps.

suffering from the vanishing gradient problem. Thanks to recent advances by Ott et al. 2018, a state-of-the-art English to French model can be trained in about 20 hours on 8 GPUs.

Part II

Contributions

Chapter 4

Neural Machine Translation

This chapter details our contributions to the field of Neural Machine Translation (NMT). It contains essentially replication work. The first section is about our implementation of word embeddings methods, and a series of experiments to validate this implementation. The second section will describe our framework for Neural Machine Translation. And finally, we will detail our replication work on Neural Machine Translation along with some original results.

The next two chapters will describe our contributions in the related fields of Automatic Speech Translation and Automatic Post-Editing.

4.1 MultiVec

MultiVec[1] (Bérard et al. 2016a) is a toolkit written in C++, which can compute vector representations for words (word embeddings), and sequences of words (sentences, paragraphs). It can also learn word embeddings in a shared vector-space between two languages (bilingual embeddings), using a parallel corpus as training data.[2] It implements several techniques from the literature, including the CBOW and Skip-Gram models from Word2vec (Mikolov et al. 2013a), with the Hierarchical Softmax and Negative Sampling training objectives (Mikolov et al. 2013c). It also implements the Paragraph Vector algorithm that can compute embeddings for sequences (Q. V. Le et al. 2014); and the Bivec algorithm for learning bilingual embeddings from parallel data (Luong et al. 2015a).

Algorithms Here is a detailed list of the algorithms that are available inside MultiVec, along with their acronym and a short description. Some of these algorithms are described more in depth in the state of the art (see Section 2.2).

- Word2vec (Mikolov et al. 2013c) is a set of efficient algorithms to learn word embeddings from a large monolingual corpus. Words have an input embedding vector and an output

[1] MultiVec is available for download at this address: `https://github.com/eske/multivec`

[2] We initially developed MultiVec for own use for Automatic Post-Editing. We wanted a fast way of computing rich crosslingual representations of sequences, to represent states in a sequential post-editing process. However, our initial goal of using Reinforcement Learning techniques for Automatic Post-Editing did not lead to any positive result, and we turned our attention to the newly discovered Neural Machine Translation techniques.

embedding. The probability of some word given some context is estimated by computing the dot product between the input vector representation of this context, and the output embedding of this word. This score is then normalized, either by doing a softmax over the entire vocabulary, logistic regression (negative sampling), or hierarchical softmax. Mikolov et al. (2013a,c) describe several techniques, which we implemented in MultiVec:

  - CBOW (Continuous Bag-of-Words): the model is trained to predict each word in a sentence by using its surrounding words (context). The input embeddings of the context words are averaged, summed, or concatenated. This context is multiplied by the output embedding of the target word. This gives a score, which is used to estimate the probability (according to the model) of this word given the context words.
  - SG (Skip-Gram) is a reversed version of CBOW. From a single word, the model is trained to predict all the context words.
  - Negative Sampling: this is a training objective that can be used conjointly with the CBOW model (CB-NS), or with Skip-Gram (SG-NS). Instead of predicting the target word by computing a probability distribution over the entire vocabulary (softmax), we sample negative examples (random words), and train the model to identify positive examples (the actual word) from negative examples (the sampled words). Negative sampling does logistic regression, whose complexity does not depend on vocabulary size (contrary to a softmax classifier).
  - Hierarchical Softmax (HS) is another training objective where the vocabulary is structured as a binary Tree, and the probability of a word can be decomposed as the probability of the corresponding path in the tree. Hierarchical softmax has a logarithmic complexity w.r.t. vocabulary size (vs. linear growth for regular softmax).

- Paragraph Vector (Q. V. Le et al. 2014) is a set of algorithms for computing vector representations of text sequences.

  - PV-DM (Distributed Memory): this model is similar to CBOW, but also computes a unique vector for each sentence in the training corpus. When predicting words in a given sentence, the vector of this sentence is included into the context vectors (along with the input embeddings of the context words). Like in regular CBOW, the context vectors can be averaged, summed, or concatenated.
  - PV-DBOW (Distributed Bag-of-Words): this is a variant of Paragraph Vector, where the model is trained to predict each word in a sentence by using this sentence's vector only (similar to Skip-Gram).
  - Online/Batch PV: the above two methods can be used in two modes. In batch mode, all the sentences whose representation we want to compute are available at once. In online mode, a model is first trained on a text corpus, and can then be used to compute representations of new sentences as they come. This is done by freezing all the parameters of the model, except for the vector of the given sentence, and train with SGD for a fixed number of iterations, on this sentence only.

- Bivec (Luong et al. 2015a): this algorithm combines two monolingual models, and uses a parallel corpus to perform monolingual updates (e.g., a source word is used to predict another source word), and crosslingual updates (e.g., a source word is used to predict a target word). At the end of training, source language words and target language words share the same vector space. By default, Bivec does a uniform (monotonous) word-based alignment between each pair of sentences. Both the CBOW and Skip-Gram models can be used, as well as the Hierarchical Softmax and Negative Sampling training objectives.

- Bilingual Paragraph Vector: Bivec can be easily combined with Online Paragraph Vector. Once a bilingual model has been trained, its two monolingual components can be used to compute the representations of a pair of sentences.

Several other implementations of Word2vec[3] have been proposed since then, including Facebook's FastText library,[4] and Gensim.[5]

MultiVec has several other useful features:

- We provide the implementation of a couple of benchmarks for word embedding evaluation: analogical reasoning, sentiment analysis, and crosslingual document classification.
- Bilingual models can be saved and used as two monolingual models.
- A Cython wrapper makes it easy for Python programmers to call the MultiVec API from Python.
- Models can be trained easily thanks to an extensive command-line interface.
- The implementation is extremely fast, on par with Word2vec, but much less obfuscated.
- Word embeddings can be exported in the same format as Word2vec (text or binary).
- A number of similarity measures are made available: similarity between two words (cosine similarity or distance), or between two sequences (bag-of-words cosine similarity, soft word error rate, etc.)

In this section, we present the framework in more details, starting with an overview of its overall architecture and command-line interface. Then we give some non-trivial implementation details. Finally, we evaluate the performance of the toolkit on several tasks, and compare it against the official implementations of the considered techniques.

4.1.1 Description and Usage

Description The `multivec/` directory contains the main source code (in C++) of the framework. In `word2vec/`, we also provide the original implementation of Word2vec, which we modified to have the same command-line interface as MultiVec. We also provide an improved binary for evaluating embeddings on the analogical reasoning task.

The `benchmarks/` directory contains scripts for running the evaluation benchmarks: sentiment analysis, crosslingual document classification, and analogical reasoning.

Finally, the `cython/` directory contains the code for the Python wrapper. This is only a thin wrapper around the C++ library. This means that most functions run just as fast as if they were called from C++.

One property of MultiVec is that it is a standalone library (like Word2vec). The only software dependencies it has are a recent C++ compiler (GCC or Clang), CMake, and Cython and NumPy

[3] https://code.google.com/archive/p/word2vec/
[4] https://github.com/facebookresearch/fastText
[5] https://radimrehurek.com/gensim/

if you want to use the Python wrapper. Compilation is straightforward, and instructions can be found Appendix A.1 or on the project's web page. For better portability, we chose to implement our own rudimentary serialization features (for saving and retrieving models), and simple vector arithmetic, rather than depending on external libraries.

The source code of MultiVec is structured as two major classes: `MonolingualModel` and `BilingualModel`. A bilingual model contains two monolingual model instances, with their own vocabulary and weights. When training a bilingual model, we do monolingual updates (that modify the corresponding monolingual model), and crosslingual updates (that modify parameters in both models). More details about the architecture of the toolkit are given in the appendix.

There are several advantages of structuring our code this way. First, monolingual models do not depend on the bilingual models (we can easily strip all the "bilingual" features from the toolkit). Then, bilingual models can be exported as two separate monolingual models (e.g., for using monolingual distance features, or online paragraph vector). Finally, it would be relatively easy to implement other crosslingual embedding algorithms like (Artetxe et al. 2017; Lample et al. 2017; Mikolov et al. 2013b).

These two classes define a number of public methods (e.g., `train`, `load`, `save`, `sent_vec`, etc.), which are accessible both from the command-line interface and the Python wrapper. However, some features, like distance functions are only accessible from the Python wrapper.

Command line Like Word2vec, MultiVec can be used to train new models from the command line. We provide two binaries for this purpose: `bin/multivec` for monolingual models and `bin/multivec-bi` for bilingual models. We give here a summary of the command line features that are available. For a more complete account, you can look in the appendix, or on the project's main page.

Given a training file `data/news.en`, with one sentence per line, tokenized at the word level[6] (and optionally lowercased), a monolingual model can be trained as follows:

```
# train and save a monolingual model
bin/multivec --train data/news.en --save models/news.bin
```

This trains a CB-NS model with the default settings. A Skip-Gram model can be trained with the option `--sg`, and the Hierarchical Softmax objective can be used with `--hs`. To only save the vectors, use the `--save-vectors` option. This writes the vectors in a text file, following the Word2vec format.

To use the PV-DM algorithm in batch mode (i.e., compute embeddings for each sentence in the training set), use the `--sent-vector` flag:

```
# train and save paragraph vectors
bin/multivec --train data/news.en --sent-vector \
--save-sent-vectors models/news.sent.vec
```

[6]The scripts `scripts/prepare-data.py` or `scripts/tokenizer.perl` can be used for this purpose.

Bilingual models are trained with the `bin/multivec-bi` program. Instead of a single training file, it takes a pair of files (a parallel corpus) with the `--train-src` and `--train-trg` options:

```
# train and save bilingual model
bin/multivec-bi --train-src data/news.en \
--train-trg data/news.de \
--save models/news.en-de.bin -v
```

By default, MultiVec does a uniform word-based alignment between parallel sentences. Optionally, a word alignment of the training corpus can be provided with the `--alignment` option. The `--beta` option controls the strength of crosslingual updates compared to monolingual ones. The source and target models can be saved individually as monolingual models with `--save-src` and `--save-trg`, and loaded later with the `bin/multivec` program or the Python wrapper.

4.1.2 Implementation Details

We detail some techniques that we used to obtain similar training speed as Word2vec. The latter is written in C and implements various tricks that make it extremely fast, at the cost of an obfuscated source code. We want to achieve a good trade-off between the clarity of higher-level programming (object-oriented programming), and the raw speed of the toolkit.

Vector library Word2vec stores the model parameters as basic `float[]` arrays. This makes the code verbose and hard to understand, as any operation on vectors (e.g., multiplying a vector by a scalar) requires writing an explicit `for` loop.

In the OOP paradigm of C++, it is easy to define a `Vector` class that supports basic vector arithmetic, thanks to operator overloading. However, our initial vector implementation turned out to be much slower than explicit `for` loops. Indeed, an expression like `x += a * y` compiles as two `for` loops: one for the multiplication by scalar `a`, and another for the addition/assignment.

A solution to this problem, which is often used in C++ linear algebra libraries, is the so called *expression templates*. Basically, it consists in using the C++ templates to build a static tree of expressions, which is compiled as a single `for` loop (loop fusion). We used this technique to improve our simple vector class, so that it performs on par with C-style `for` loops. Our implementation supports the following vector operations: Euclidean norm, multiplication/division by a scalar, element-wise addition/subtraction by another vector of the same size, and dot product between two vectors. We did not want to use an external linear algebra library, as we did not find any standalone library that was sufficiently small, easy to install and portable.

Asynchronous training Like Word2vec, MultiVec can make use of multi-core CPUs by spanning multiple threads, each taking care of its own part of the training file. Surprisingly, even though all threads share the same memory, Word2vec does not care about thread safety. Several threads can read and write at the same location in memory, resulting in some amount of noise as some SGD updates are lost or overlap with others. As observed by Y. Bengio et al. (2003) in

the context of a Neural Language Model, such an asynchronous implementation does not seem to impact training performance.

In MultiVec, this is implemented as a `chunkify` function, which reads the entire training file, and delimits it into "chunks", so that all chunks have the same number of lines. We do not actually split the file into several files, nor do we store the file into memory (training files are often too large to fit in memory), but just store the starting position of each chunk on disk. Then, as many threads as there are chunks are created. Each thread runs its own training procedure that reads the corresponding chunk from disk, line-by-line, and trains for a given number of epochs. All threads share the same memory, and can access and modify model parameters concurrently.

As a measure of comparison, we also tried to enforce thread safety by using mutexes.[7] Each parameter matrix (input weights and output weights) has its own instance of `std::mutex`. Whenever a thread needs to read or update parameters, it locks the corresponding mutex, effectively preventing other threads from accessing the same parameters. We measure the training speed of CB-NS using this approach, and the accuracy of the trained embeddings on the analogical reasoning task. Table 4.1 shows that enforcing thread safety introduces a substantial overhead, with a catastrophic slowdown when using 16 threads. Furthermore, it does not seem to improve the performance of the learned embeddings on the analogy task.

Fast random generator MultiVec (and Word2vec) need access to a random number source. Random numbers are needed for initializing the model parameters (input and output weights), and then during training for subsampling (random dropping of words depending on their frequency), for selecting the window size and for negative sampling (random selection of negative examples depending on their frequency). While random weight initialization is done only once and is not a bottleneck, the other functions are run at a higher frequency.

As a side note, there is another source of stochasticity, which is the asynchronous implementation of SGD. Threads do not always run at the same speed, and there is no guarantee as to the order of SGD updates. Overlapping updates because of our non thread-safe implementation can even add more random noise.[8]

There are $O(epochs \times |D|)$ calls to `rand` for subsampling and window size selection (where $|D|$ is the number of words in the training corpus). The real bottleneck is with negative sampling, with $O(epochs \times |D| \times neg)$ calls for CBOW and $O(epochs \times |D| \times C \times neg)$ for Skip-Gram (neg is the number of negative samples, and C is the average window size). This is only a factor n (dimension of the embeddings) away from the complexity of the training algorithm (in terms of basic additions or multiplications). For this reason, fast pseudo-random number generation is essential. We use the following function:

```
inline unsigned long long rand(unsigned long long max) {
    thread_local long long thread_id =
            std::hash<thread::id>()(this_thread::get_id());
    // unique seed for each thread
    thread_local unsigned long long next_random(time(NULL) + thread_id);
    next_random = next_random * 25214903917ull + 11;
    // with this generator, the most significant bits are bits 47...16
```

[7] This an object that can be in two states: locked or unlocked. Whenever a thread tries to lock a mutex that is already locked, it blocks until the mutex is unlocked by the thread that owns it.

[8] A consequence is that MultiVec gives different results across training instances, even when using the same seed for the pseudo-random generator.

```
    return (next_random >> 16) % max;
}
```

This function returns a value between 0 and $max - 1$. The `thread_local` specifier creates a single random generator per thread (which is shared across calls to `rand`). This ensures thread-safety, without the overhead of locking access to the random generator whenever we are using it. This is the same pseudo-random generator as used by Word2vec, which we wrap in a function for better readability.

We compare this implementation against a naive use of `std::rand`, and a more robust (and standard) implementation that uses a Mersenne Twister (`std::mt19937`), which we also wrap inside a similar function.

Table 4.1 shows that `std::rand` is extremely slow, in particular with multiple threads. This is due to the fact that its implementation in GCC enforces thread safety in a blocking way (with mutexes). On the other hand, the high-quality Mersenne Twister generator from the standard library is only slightly slower that the custom Linear Congruential Generator used by Word2vec. Our evaluation on the analogical reasoning task shows no difference, so we keep the faster version.

The negative sampling algorithm samples words from the vocabulary according to their unigram distribution. To efficiently sample from this distribution, we adopt the same trick as Word2vec. A large table of fixed size (`1e8`) is built, where the index of each word in the vocabulary is copied a number of times which is proportional to its frequency. It is then easy to sample from this distribution by doing uniform sampling over the entire table.

Time benchmark To compare the training speed of our framework with different implementation choices, we do the following benchmark: we train CB-NS models on the English side of News Commentary, with 10 iterations. For each method, we train 10 models, and average their training time (not counting initialization and saving), and accuracy on the analogy task. We repeat this on two different machines, one with a 4-core CPU, and one with a 16-core CPU. Table 4.1 shows the average training time of the methods we tried. In terms of accuracy, all the methods obtain similar performance.

We see that methods that enforce thread safety by using mutexes (Sync SGD and `std::rand`) are extremely slow, in particular with 16 threads. There does not seem to be an accuracy improvement when using a thread-safe implementation of SGD.

Discretizing the results of the exponential function inside a table, like Word2vec (to avoid having to repeatedly call `std::exp`) only reduces training time by a tiny fraction, at the cost of making the code less readable. Finally, the custom rand function (like Word2vec) is slightly faster than using the random generators from the standard library.

Thus we keep the "Baseline" row from Table 4.1 as our final implementation, as it gives the best trade-off between code clarity and training speed.

4.1.3 Experiments

We perform a set of experiments to benchmark the training speed of our toolkit in different conditions, and to compare its performance against its competitors. As it re-implements the

Model	4 Threads (i5)		16 Threads (Xeon)		Precision @1
	Time (s)	Relative	Time (s)	Relative	
Word2vec	33	-18%	22	-10%	17.9%
Baseline	**41**		**24**		**18.5%**
(1) Exp table	40	-3%	23	-3%	18.3%
(2) Proper rand	43	+5%	25	+4%	18.5%
(3) Vanilla vec	63	+54%	38	+58%	18.4%
(4) `std::rand`	106	×3	247	×10	18.3%
(5) Sync SGD	235	×6	854	×35	18.4%
(3) + (4) + (5)	363	×9	1257	×52	18.2%

TABLE 4.1: Training time of the CBOW-NS model, with different implementation choices. The models are trained on the English side of News Commentary (5M words, lowercased). The training times (in seconds) are averaged over ten runs. `dim=100, iter=10, negative=10, win_size=5, subsampling=1e-3, min_count=5`. The precision is over the analogical reasoning task (see next subsection for a description).

exact same techniques as other models in the literature, it should obtain the same results as the official implementations of these methods. The first task is the analogical reasoning task, to evaluate the CBOW and Skip-Gram models (against Word2vec). Then we do sentiment analysis to evaluate our implementation of Paragraph Vector. Finally, we evaluate our implementation of Bivec on the crosslingual document classification task.

Analogical Reasoning We evaluate our toolkit on the Analogical Reasoning Task as described in (Mikolov et al. 2013d). The authors provide a dataset containing five different types of semantic questions, and nine types of syntactic questions, with a total of 19 558 questions. A question is a tuple $(word_1, word_2, word_3, word_4)$ in which $word_4$ is related (semantically or syntactically) to $word_3$, in the same way that $word_2$ is related to $word_1$. A famous example is $(king, man, queen, woman)$. It has been observed that $C(king) - C(man) \approx C(queen) - C(woman)$. This task evaluates the ability of the model to capture several kinds of linguistic regularities. For instance, other types of questions include *state-city* relationships or *adjective-adverb* relationships.

The accuracy as measured in this task is the percentage of questions for which the closest word in the vocabulary to $word_3 - word_1 + word_2$ according to the cosine similarity is exactly $word_4$ (precision at one).

As shown in Table 4.2, Word2vec and MultiVec with the same settings get very similar results. Interestingly, bilingual models seem to perform significantly better, even on a monolingual task. The number of epochs was intentionally halved in the bilingual case, to make sure that this result is not simply due to a higher number of updates. Interestingly, the CBOW model seems to perform better on syntactic questions, while Skip-Gram is much better on semantic questions.

We also observe that MultiVec is almost as fast as Word2vec. All the models are trained on the same 16-core CPU (Intel Xeon).

There are some limits to this task: there are not many question types and the scores that we compute are unbalanced. Some question types have many more examples than others. For example, 80% of all semantic questions are about city-state relationships. There are also more syntactic questions than semantic ones. While our `analogy` program also prints a score for each

Method	Model	Dim	Syntactic (%)	Semantic (%)	Total (%)	Time (min)
Word2vec	CBOW	100	47.4	24.8	37.5	21
		300	**48.2**	25.6	38.3	66
	SG	100	44.8	33.1	39.7	88
		300	44.5	**38.6**	**41.9**	280
MultiVec	CBOW	100	47.7	27.4	38.8	22
		300	**48.7**	27.9	39.6	66
	SG	100	45.1	33.8	40.1	92
		300	44.8	**39.3**	**42.4**	278
Multivec-Bi	CBOW	300	**52.6**	39.1	**46.6**	161
	SG		47.5	**42.6**	45.3	478

TABLE 4.2: Results (precision @1) of the analogical reasoning task, on Word2vec's `questions-words.txt`. The models were trained on the English side of WMT14 (117M words) for 20 iterations, "Multivec-Bi" is our bilingual implementation, trained on English-German WMT14 for 10 iterations. Training time is given in minutes. The initial learning rate is 0.05 for Skip-Gram models, and 0.1 for CBOW models. The other parameters are: `negative=10, win_size=5, subsampling=1e-4, min_count=5`. The vocabulary, of size 176k, covers 18 936 questions. The evaluation is lowercased.

Toolkit	Method	Training data	Error rate (%)	Time (s)
Word2vec	batch PV-DM	~~train + test (non-shuf)~~	~~7.2±2.6~~	720
		train + test	**11.6±0.1**	429
MultiVec			**11.8±0.1**	473
	online PV-DM	train	12.0±0.1	381 + 700
		WMT14	20.4±0.3	614 + 701
	bag-of-words	train	13.0±0.1	358
		train + test	12.0±0.1	471
		WMT14	19.9±0.2	601

TABLE 4.3: Results of the sentiment analysis task on the IMDb dataset (averaged over three runs). The batch models were trained on the IMDb training and test data. The online models were trained on either the IMDb training data or on the English side of the WMT14 English-German parallel corpus (117M words). The settings are: PV-DM (CBOW) with 40 iterations, 15 negative samples, a dimension of 100, a window size of 5, a learning rate of 0.1 and a subsampling rate of 10^{-4}. The large WMT models are trained for 10 iterations only. The first row of the table is obtained on non-shuffled data (with hierarchical softmax, and a window size of 10), which makes it invalid (Mesnil et al. 2014).

question type, and balanced scores (average of all question type scores), we report unbalanced scores, for easier comparison with other results in the literature.

Also, the scores highly depend on the size of the vocabulary. Questions that are out-of-vocabulary are skipped (when any of the four words is absent from the vocabulary). Because we compute the cosine similarity between the query and all words in the vocabulary, and get the word with the highest score, smaller vocabularies have an advantage (it is more likely that we pick the wrong word in a larger set of words). This makes comparison difficult between models that were trained on different datasets.

Sentiment Analysis We evaluate our implementation of paragraph vector on the sentiment analysis task. The same experimental protocol as (Q. V. Le et al. 2014) is used.[9] The IMDb dataset contains 100 000 documents. 50 000 of those are labeled with a positive or negative label, and 50 000 are unlabeled. The representations of 25 000 labeled documents are used as training examples for a logistic regression classifier. The remaining 25 000 labeled documents are used as test examples.

Table 4.3 reports the results of the different models. We compare the batch Paragraph Vector implementation provided by Q. V. Le et al. (2014) with our batch and online implementations. We also report results obtained by simply averaging word embeddings (bag-of-words). We observe that the Paragraph Vector models are only slightly better than a simple bag-of-words. Models that are out-of-domain (trained on WMT data only) are much worse, with an error rate around 20%. The online version of Paragraph Vector is only slightly worse than the batch version, even though it is trained on less data (75 000 documents against 100 000).

As Mesnil et al. (2014) remarked, the excellent results in the original paper are not reproducible. The authors provided an implementation of their method, which obtains similarly good results as reported in the paper, but only when applied on a non-shuffled dataset (examples are sorted according to their label). The first row of Table 4.3 illustrates this. It seems like hierarchical softmax is able to encode some sort of positional information in the embeddings, which the classifier can make use of to cheat.

Crosslingual Document Classification To evaluate the quality of our bilingual word embeddings, we reproduce Klementiev et al. (2012)'s experiments on the crosslingual document classification task. This task consists in classifying documents in a language using a model that was trained with documents from another language. Like Klementiev et al. (2012) and Luong et al. (2015a), 1000 documents from the RCV corpus are used for training, and 5000 documents for testing. Each document belongs to one of 4 categories.

Like Luong et al. (2015a), we train bilingual word embeddings on the English-German Europarl, using the same settings. Document representations are then computed by doing a weighted sum of word embeddings, according to pre-defined word frequencies (TF-IDF). A perceptron classifier is then trained on the source-language documents and evaluated on target-language documents.[10]

We compare bilingual models trained with Bivec and MultiVec. We also show results obtained by computing document representations with online paragraph vector. To do so, we export the previously trained bilingual model to source and target models, which are then used to compute paragraph vector representations for source and target documents.

Table 4.4 shows similar results for both MultiVec and Bivec. Paragraph vector does no better than the bag-of-words representation, but the results confirm that our approach for computing bilingual paragraph vectors is sound.

[9] A training script and a modified version of Word2vec was provided by the authors on the Word2vec Google group.

[10] The data splits and training scripts were provided by the authors.

Dim	Toolkit	Method	Accuracy [%] en→de	Accuracy [%] de→en
40	Bivec	bag-of-words	86.1	74.4
	MultiVec-Bi	bag-of-words	88.1	75.3
		online PV-DM	88.4	77.6
128	Bivec	bag-of-words	89.0	78.6
	MultiVec-Bi	bag-of-words	88.9	76.4
		online PV-DM	88.2	79.1

TABLE 4.4: Results obtained within the framework of the CLDC task using the RCV corpus. $en \rightarrow de$ means that we train on English data and test on German data; $de \rightarrow en$ is the reverse. The settings are the same as those in (Luong et al. 2015a): skip-gram model, 30 negative samples, 10 epochs.

4.2 Seq2seq

This section presents our framework for Neural Machine Translation. The framework is implemented in Python on top of TensorFlow (Abadi et al. 2015), a symbolic math library, particularly useful for programming neural networks. Our code is mostly original, but reuses some components of TensorFlow's seq2seq example, and some scripts from other sources: like Moses pre-processing scripts (Koehn 2010), or BPE extraction scripts from (Sennrich et al. 2016c).

We will start with a quick description of our framework, then we will give some implementation details that we find interesting. And finally, we will give an example of model configuration, and instructions on how to train and use models. The next section will give a few experimental results that we obtained using this framework.

4.2.1 Description

Features Seq2seq implements encoder-decoder models for Neural Machine Translation, as well as models for other tasks that we tackled (e.g., Automatic Speech Translation and Automatic Post-Editing). Its features are, but are not restricted to:

- Encoder-decoder models with global attention (Bahdanau et al. 2015), local attention model (Luong et al. 2015b). Support for multiple layers of LSTM or GRUs, bidirectional encoders, and various decoder architecture (Sennrich et al. 2017).
- Beam search and ensemble decoding, also possibility to average the weights of several models (Junczys-Dowmunt et al. 2016b).
- Multi-Source training with multiple encoders and multiple attention mechanisms (Zoph et al. 2016a).
- Multi-Task training with models that share an encoder or a decoder (Luong et al. 2016).
- Pyramidal speech encoder (Chan et al. 2016) and convolutional speech encoder (Weiss et al. 2017).
- Character-level encoder and decoder (Lee et al. 2016).

The main program (`seq2seq.sh`) takes a YAML configuration file as first argument, which defines the values of the model hyperparameters (training data, model destination, architecture, model size, etc.) Then, the user specifies an action to take: train, decode, evaluate or align. Training mode can resume training if the model directory already exists, or starts training from scratch. When training a new model, a directory is created that contains all the information necessary for using the model and replicating the experiment. Decoding mode takes an existing model (it assumes a model already exists at the specified location), and translates the given text corpus. Evaluation mode is similar, but it also evaluates the outputs against a reference translation using automatic metrics (e.g., BLEU). Alignment mode uses the attention model to plot an alignment between the input and the output sequence.

Architecture The `translate/` directory contains the main source code of seq2seq. The `scripts/` directory contains scripts for tokenization, BPE segmentation, speech feature extraction, vocabulary creation, training monitoring, etc.

We split the logic of seq2seq into several Python files. There is a main program that is responsible for parsing the command line arguments and the configuration files, and for calling high-level routines depending on the user's wishes (train, decode, evaluate). The `TranslationModel` class defines the non-TensorFlow logic of a model. It holds the training and evaluation data, and defines high-level methods for training, decoding, evaluating, saving or loading a model. `MultiTaskModel` can have several instance of the former and follows the same public interface. Its `train` method picks a random model at each training step and does an update on the corresponding model. Another file (`models.py`) contains the functions for creating TensorFlow graph blocks (e.g., encoder, decoder, training loss, etc.) The `Seq2SeqModel` class is the bridge between `TranslationModel` and the TensorFlow graph. It is responsible for creating the graph, and it defines methods for getting the next batch, doing an SGD update, or decoding a batch. It is important to know that a static TensorFlow graph is first created by the constructors of `TranslationModel` and `Seq2SeqModel`. Then, it is initialized, either at random or by loading an existing checkpoint. Only then training or decoding can proceed by sending data into the graph.

Appendix A.2.1 gives a description of the TensorFlow features that are useful to understanding our code. We also give a more detailed overview of the architecture of the project and of the structure of the TensorFlow graph (in Appendix A.2.2).

Hyperparameters are passed around from the main program up to the functions responsible for creating the graph parts, using special dictionaries whose elements can be accessed as attributes (e.g., `config.cell_size`). The configuration file can define one or several encoders, and a decoder. Each of these can redefine their own value for each hyperparameter (e.g., cell size). Encoders and decoders are identified by a `name` attribute, which is used to name the model variables and defines the extension for the training files. For instance, if the encoder is named "en", then seq2seq will look for `train.en`, `dev.en` and `vocab.en`, and the encoder's embedding matrix will be named `embedding_en`. If a hyperparameter is not redefined in the experiment's config file, seq2seq looks for its default value inside `config/default.yaml`. Some hyperparameter can be modified thanks to command-line options (e.g., batch size and beam size), which have the highest precedence.

Reproducibility is made easier by our framework. It logs a lot of information during training, like timestamps, configuration, graph variables (names and dimension), and the results of a periodic evaluation on the dev set. The log file of a given experiment already tells a lot about the model, which may facilitate its replication. When training a model, seq2seq also saves the

configuration files of the experiment in the model directory, as well as the vocabulary files. It also makes an archive of the current version of the source code and saves it in the target directory.

4.2.2 Implementation Details

We now describe a few of the capabilities of seq2seq and give implementation details that we find interesting.

Padding For efficient training and decoding on GPUs, multiple sentences are grouped in a batch. Each of their symbols are first mapped to an integer id that corresponds to an index in the vocabulary. We group sentences of similar length together to improve training speed (Bahdanau et al. 2015). Because a batch is represented as a tensor, all sequences in a batch must have the same length. For this reason, we pad shorter sentences with dummy symbols. Then, in the TensorFlow graph, we use masks (tensors of zeros and ones) to inform the model about the true sequence length. For instance, we do not want the attention model to look at dummy hidden states (that are beyond the actual length of the input sequence). We also do not want the cross-entropy training loss to take dummy outputs (after the end-of-sentence marker) into account.

Attention decoder Contrary to most early implementations on TensorFlow, our attention decoder is dynamic. This means that the sequence length does not need to be defined at the graph creation, but can have a different value for each new batch (the length of the longest sequence in the batch). The graph is dynamically unrolled to the correct length thanks to the `tf.while_loop` function. This is much faster than having a single static graph with a large sequence length, and more convenient than having different static graphs for different sequence lengths ("bucketing" approach).

Pre-training trick Pre-training a model on some dataset and then finetuning on an another dataset is made easy by our framework. The user only has to change the path to the training files in the configuration file, and then resume training. We would recommend however to change the model's target directory, and load the pre-trained model(s) with the `--checkpoints` option.

It is also possible to do partial variable sharing. For instance, we may want to train a French-to-English machine translation model with large amounts of data. And then use this model to initialize a German-to-English machine translation model, which we train with smaller amounts of data. Pre-training can help converging faster, acts as an implicit regularizer, and sometimes converges to a better solution. In this case, only the decoder is shared between the two models. To run this scenario, the user only has to create two configuration files, which share the same decoder configuration (same name and same settings). When the first model has finished training, the user can initialize the second model with the first model using the `--checkpoints` option. Only the variables that share the same name will be loaded, and the other variables will be initialized at random. A model can be pre-trained with several models by providing several arguments to the `--checkpoints` option (e.g., one that shares the same encoder and one that shares the same decoder).

The implementation of this feature is actually straightforward. We named the graph variables carefully (using `tf.variable_scope`), to include the name of the encoder or the decoder to

which they belong to. This way, when loading a checkpoint from a model that had an encoder with the same name as the current model's encoder, the encoder variables are initialized with its values.

When finetuning, the user can choose to freeze some parameters (e.g., the embeddings). This can help to avoid overfitting when finetuning on a small dataset.
This is possible with a `freeze_variables` list in the configuration file. Any variable whose name appears in this list will not be updated by SGD.

Multi-task training We use the same variable naming strategy for multi-task training, combined with TensorFlow's variable 'reuse' capability. In TensorFlow, when trying to create a variable that already exists in the graph (which has the same name), we can seamlessly use the previous variable.

When several tasks are specified in the configuration file (see Appendix A.2.2 for examples), seq2seq creates one instance of `TranslationModel` for each task. Each task creates its own bricks in the graph (encoder, decoder, attention, loss, etc.) Thanks to automatic reuse, variables that were already created by a previous model are reused and automatically shared across tasks.

Multi-task training is then done by alternating updates between tasks (Luong et al. 2016). The `train` method of `MultiTaskModel` repetitively picks a random model, and calls its `train_step` method.

Beam search decoder Our beam search decoder is a modified version of a third-party implementation.[11] The advantage of this implementation is that it runs entirely in the TensorFlow graph, and that it can decode entire batches at once. However it is quite minimalist, and we added some features.

The running hypotheses are stored in a tensor of shape `[batch_size, beam_size, current_step]` that contain sequences of token ids. Because the decoder's state is conditioned on its past outputs, each hypothesis has a different decoder state. The current decoder states are stored in a tensor of shape `[batch_size, beam_size, state_size]`. Similarly, we store the current score (sum of log probabilities of all tokens) of each hypothesis, and a mask that keeps track of finished hypotheses (containing an `EOS` token).

```
state = tf.tile(tf.expand_dims(initial_state, axis=1), [1, beam_size, 1])

scores = tf.log([[1.] + [0.] * (beam_size - 1)])
scores = tf.tile(scores, [batch_size, 1])

ids = tf.tile([[utils.BOS_ID]], [batch_size, beam_size])
hypotheses = tf.expand_dims(ids, axis=2)    # current beam
mask = tf.ones([batch_size, beam_size], dtype=tf.float32)

for i in range(max_len):
    ids = tf.reshape(ids, [batch_size * beam_size])
    state = tf.reshape(state, [batch_size * beam_size, state_size])

    state, logits = time_step_fun(state, ids, i)

    state = tf.reshape(state, [batch_size, beam_size, state_size])
```

[11] https://github.com/vahidk/EffectiveTensorflow

```
    logits = tf.reshape(logits, [batch_size, beam_size, vocab_size])
    token_scores = log_softmax(logits, axis=2)

    mask1 = tf.expand_dims(mask, axis=2)
    mask2 = tf.one_hot(indices=[[utils.EOS_ID]], depth=vocab_size)
    token_scores = token_scores * mask1 + (1 - mask1) * (1 - mask2) * -1e30

    scores = tf.expand_dims(scores, axis=2) + token_scores
    scores = tf.reshape(scores, [batch_size, vocab_size * beam_size])

    # returns tensors of shape [batch_size, beam_size]
    scores, indices = tf.nn.top_k(scores, k=beam_size)

    # token id and beam id of each selected hypothesis
    beam_ids = indices // num_classes
    token_ids = indices % num_classes

    state = batch_gather(state, beam_ids)
    hypotheses = batch_gather(hypotheses, beam_ids)
    mask = batch_gather(mask, beam_ids)

    hypotheses = tf.concat([hypotheses, tf.expand_dims(token_ids, axis=2)],
                           axis=2)
    mask *= tf.to_float(tf.not_equal(token_ids, utils.EOS_ID))

hypotheses = hypotheses[:, :, 1:]  # remove BOS symbol
```

In this version, the graph is statically unrolled (python for-loop). At each time step, the `state` and `ids` tensors are reshaped before being passed to the decoder's time step function. The trick here is to treat all running hypotheses as elements in a batch. The decoder returns a new state and a `logits` tensor, which contains scores for each possible expansion of each hypothesis in the beam. We compute log softmax scores for these logits, and add them to their respective hypothesis scores. We get a set of beam size $\times$ vocab size candidates. Then, we use `tf.nn.top_k` to get the indices (and values) of the 'beam size' best scores. These will constitute a new set of hypotheses. These indices have a shape of `[batch_size, beam_size]` and range from 0 to $(\text{vocab size} \times \text{beam size} - 1)$. They can be decomposed into a beam id (the index in the beam of the hypothesis that was expanded), and a token id (the index of the new token in the vocabulary). Then we select the decoder states, mask and hypotheses that correspond to these beam ids. Finally, we append the new token ids to the current hypotheses, and update the mask to account for potentially newly finished hypotheses.

Compared to this implementation, we add several features:

- Instead of a static loop, we use a dynamic `tf.while_loop`. This is much faster to compile, and we can set the maximum length dynamically (e.g., depending on the input length). If all the hypotheses in the current beam are finished (i.e., they all contain an `EOS` token), we stop decoding early (thanks to the dynamic stopping condition of the while loop). This makes decoding much faster, in particular in an online setting where the batch size is 1.
- A huge problem with the default implementation is that it continues expanding the finished hypotheses. This can be a problem if a finished hypothesis has a very good score. It can expand into several identical hypotheses with good scores preventing more interesting hypotheses to expand. At the end, we potentially get a set of short identical hypotheses (except for the symbols after the first `EOS`). To solve this problem, we force the decoder to

expand the already finished hypotheses with `EOS` symbols only (by adding a large score penalty to the other expansions). This way, a finished hypothesis cannot be duplicated. Furthermore, these extra symbols are not taken into account into the total score of a hypothesis. This makes it impossible for finished hypotheses to be removed from the beam (which is equivalent to reducing the beam size by one each time a new finished hypothesis is selected).

- Length normalization: after the while loop, we rescore the translation hypotheses by dividing the log scores by hypothesis length (power some tunable weight). This gives more chance to longer hypotheses.
- Ensemble decoding: we call the time step function for each model in the ensemble, and average their log probabilities. We also store the decoder states of all models in the ensemble for the next time step.

Ensemble For ensemble decoding, we create an instance of `Seq2SeqModel` for each model in the ensemble, under a different variable scope. This means that all variables in a model will have a special prefix in their name (e.g., `model_1/`), which differentiates them from equivalent variables in other models from the ensemble. Then, the variables are initialized by loading the checkpoints (one checkpoint for each model in the ensemble) and mapping their variable names to the new names (with a prefix). The advantage of this implementation is that it is easy to combine the outputs of several models in a single graph operation, and decoding can run entirely on the GPU. This is also straightforward to integrate into our TensorFlow implementation of beam search.

Generate, update and look In attention-based sequence to sequence models, it is unclear in which order the *generate*, *update* and *look* operations should be run. The update operation takes the previous state, the last computed context and the previous symbol and outputs a new state. The look operation uses the current state to create a new context vector. And the generate operation takes the current state, the current context and the previous symbol and outputs a new symbol.

The two main approaches are: *look* then *update* then *generate* (Bahdanau et al. 2015; Jean et al. 2015a), or *look* then *generate* then *update* (Jean et al. 2015b). The latter one, though slightly more complex to implement makes a bit more sense in our opinion.

Both approaches have limitations. In the first one, because the next symbol is generated last, the state that is used to compute the next attention context is not up to date with this symbol. In the second one, the state that is used for generating the next symbol has not been updated with the latest context vector. Our general observation is that the second approach of generating a new symbol before updating the state performs better. Jean et al. 2015b do the same observation.

A solution that we find elegant is the conditional GRU by Sennrich et al. (2017). It uses two GRU cells whose states are intertwined. The first state is updated with the latest symbol, then this state is used to compute a new context vector, which is used to update the second state, and finally this state and the up-to-date context vector are used to generate a new symbol. This can be understood as doing: *update*, *look*, *update* again and then *generate*. Our framework implements these three approaches, which can be controlled thanks to parameters in the configuration files.

4.2.3 Use Example

We now give a basic example on how to use seq2seq. More examples and configuration files are available on the project's web page.[12] The first step is to download the training data and to pre-process it. The next step is to train a model with seq2seq (on a GPU). Once a model has finished training, it can be used for translating new examples.

How to use seq2seq Use `scripts/prepare-data.py` to pre-process the data into tokenized train, dev and test corpora, with optional lowercasing, length filtering, punctuation normalization, and byte-pair encoding. The script also creates vocabulary files that are necessary for mapping tokens (words, characters or subword units) to token ids and conversely. Once this data is pre-processed, a configuration file for the experiment needs to be created. The main parameters are the names of the encoder and decoder, the max sequence length, the data and model directories, the training algorithm, learning rate, batch size and saving/evaluation frequency.

Then, start training with `./seq2seq.sh config_file --train -v`, preferably on a machine with a GPU, and inside a `screen` session, so as to be able to go back to the process even after leaving the parent terminal. GPU usage can be monitored thanks to the `nvidia-smi` command. Training progression can be observed on the standard output, or into a log file inside the model directory.
Special scripts are useful to monitor training performance: `get-best-score.py` parses the log file and looks for the best score according to given metric. `plot-loss.py` can take several log files as parameter and plot their training loss, dev loss or other metrics like BLEU according to training steps. With the `--text` or `--auto` flags, it shows a table in text mode, which allows monitoring and model comparison even without a display server (e.g., over SSH).

Training won't stop on its own, unless a maximum number of steps or epochs have been configured. With large experiments, it can be more convenient to let the experiment run for an indefinite period of time, and then to manually interrupt training when the performance does not seem to improve anymore or starts getting worse. Seq2seq evaluates the model periodically on the dev set according to user-defined metrics (e.g., BLEU), and saves checkpoints (snapshot of the model variables at a given time) when it achieves the best performance.[13]

Once training is complete, the model can be used for decoding or evaluated on a test set, thanks to the `--decode` or `--evaluate` options. By default, these options will select the best checkpoint according to chosen evaluation metric. Several model instances can also be trained, and combined at decoding time with the `--ensemble` flag and the `--checkpoints` option.

BTEC example Here is an example of configuration on BTEC (Basic Travel Expression Corpus), which achieves very good results:

```
label: 'BTEC baseline'

cell_size: 256
attn_size: 256
embedding_size: 128
```

[12] https://github.com/eske/seq2seq

[13] By default, it keeps the latest checkpoint (so that we can resume training if interrupted), and 4 checkpoints that correspond to the 4 best scores to date.

```
bidir: True             # bididirectional encoder
cell_type: GRU          # GRU or LSTM
weight_scale: 0.1       # normal init with stddev=0.1

data_dir: data/BTEC
model_dir: models/BTEC
batch_size: 32

optimizer: adam         # training algorithm
learning_rate: 0.001

steps_per_checkpoint: 2000 # saving freq (number of batches)
steps_per_eval: 2000    # dev set evaluation frequency

batch_mode: standard    # epoch shuffle + sort batches
read_ahead: 10          # number of batches to sort
max_len: 25             # max number of input and output words
max_steps: 30000        # stop after this many SGD steps

encoders:
  - name: fr            # source files extension

decoders:
  - name: en            # target files extension
    conditional_rnn: True  # conditional GRU
    pred_deep_layer: True  # dense layer before vocab proj

use_dropout: True       # use dropout when training
rnn_input_dropout: 0.2
word_dropout: 0.2       # drop src and trg words at random
```

The `read_ahead` parameter reads this many batches ahead of time, sorts the sequences by target length and regroups sequences of similar length in the same batches. The advantage of this approach is that this reduces the amount of padding that is needed. Training speed is increased, because the average max length in a batch is smaller.

We pre-process the (already tokenized) BTEC data, by lowercasing and producing a word-based vocabulary with no size limit:

```
scripts/prepare-data.py raw_data/BTEC/train fr en data/BTEC \
    --no-tokenize --vocab-size 0 --lowercase \
    --dev-corpus raw_data/BTEC/dev \
    --test-corpus raw_data/BTEC/test
```

To train a model with this configuration, the user only has to run this command (preferably in a `screen` so that she can keep interacting with it):

```
./seq2seq.sh config/BTEC.yaml --train -v
```

This creates a `models/BTEC` directory, and starts logging training information on the standard output and in `models/BTEC/log.txt`. Training will automatically stop after 30 000 SGD steps, but the user can also manually stop the process if she's satisfied with the model's performance on the dev set. After 10 000 steps (about 17 full epochs and 20 minutes of training on a small GTX 750 Ti GPU), the dev BLEU score is 49.3. After this point the model starts overfitting: the training loss continues decreasing while the dev loss starts increasing.[14]

Then, to decode the test set (using the best checkpoint so far), the user can run the following command. This prints the translation hypothesis on the standard output.

```
./seq2seq.sh config/BTEC.yaml --decode data/BTEC/test.fr \
    --beam-size 8
```

Or to evaluate directly, and store the results of decoding inside `test.mt`:

```
./seq2seq.sh config/BTEC.yaml \
    --evaluate data/BTEC/test.{fr,en} \
    --beam-size 8 --output models/BTEC/test.mt
```

4.3 MT Experiments

This section presents experimental results (replicated or original) that we obtained on Neural Machine Translation with our framework. We worked on two different tasks: English-to-French translation of news with the large WMT14 corpus, and German-to-English translation of subtitles with the smaller IWSLT14 corpus.

4.3.1 News Translation (WMT14)

The WMT 2014 (Workshop on Machine Translation) English-to-French translation task is a popular benchmark in the Neural Machine Translation literature (Bahdanau et al. 2015; Cho et al. 2014b; Gehring et al. 2017b; Jean et al. 2015a; Johnson et al. 2016; Shazeer et al. 2017; Sutskever et al. 2014; Vaswani et al. 2017; Wu et al. 2016; Zhou et al. 2016).

The full training data contains 36M sentence pairs, with approximately 2B words for language modeling. However, most of this data was automatically crawled from the web and is quite noisy. Cho et al. (2014b) use a data selection technique (Axelrod et al. 2011) to select a subset of 12M parallel sentences (348M French words) and 418M French words for language modeling. The advantage of this training set is that it is smaller and cleaner, thus easier to train with. The first results on NMT were obtained with this training set. However, the best results to date on this task are obtained with the full 36M corpus (Gehring et al. 2017b; Shazeer et al. 2017; Vaswani et al. 2017).

The development set is a concatenation of the `newstest2012` and `newstest2013` datasets, and the test set is `newstest2014`. Table 4.5 shows data statistics about the filtered corpus. The evaluation is performed with *tokenized* BLEU, i.e., the target of the evaluation is the target

[14] The best BLEU score we have managed so far on BTEC dev with greedy decoding is 53.2 (with a different model).

Corpus	Total Lines	English		French	
		Words	Average	Words	Average
Train	12.1M	304M	25.2	348M	28.8
Dev	6003	138k	23.0	155k	25.9
Test	3003	71k	23.7	81k	27.0

TABLE 4.5: WMT14 English-French corpus statistics.

side of the test corpus tokenized at the word-level by Moses' `tokenizer.perl` script.[15] The evaluation is case sensitive, which means that a word starting with a lowercase letter, and the same word starting with a capital letter are not considered identical.

We replicate pioneer work from Bahdanau et al. (2015) and Jean et al. (2015a), and also try other techniques like Byte Pair Encoding (Sennrich et al. 2016c) and conditional GRU (Sennrich et al. 2017). The goal of this set of experiments is to validate our implementation and to show the reproducibility of the aforementioned results. We also wish to perform a more in-depth analysis of these results, in particular by looking at the training progression and the outputs of the models. However, training on this dataset is very expensive (about a week on a high-end GPU with the models we tried). Because of resource and time limitations, we are not able to present more results. We go more in depth in the next subsection with a smaller dataset.

Models We implement and train an exact copy of Bahdanau et al. (2015)'s RNNsearch model.[16] We also apply the same technique as Jean et al. (2015a) to replace the output `UNK` tokens.

We use a bidirectional encoder with GRU cells of size n and source embeddings of size m. This results in a sequence of hidden states $h_1, \dots, h_T \in \mathbb{R}^{2n}$. The last backward encoder state is used as initial state for the decoder, after a non-linear transformation: $s_0 = \tanh(W_{init}\overleftarrow{h}_1 + b_{init})$ where $W_{init} \in \mathbb{R}^{n \times n}$ and $b_{init} \in \mathbb{R}^n$.

$$c_t = \sum_i^T \alpha_{ti} h_i \qquad \alpha_{ti} = \frac{e^{r_{ti}}}{\sum_{k=1}^T e^{r_{tk}}} \qquad r_{ti} = v_{att}^\top \tanh(W_{att}(h_i \oplus s_{t-1}) + b_{att}) \tag{4.1}$$

$$s_t = \text{GRU}(s_{t-1}, E'(\widetilde{z}_{t-1}) \oplus c_t) \tag{4.2}$$

$$x'_t = s_t \oplus E'(\widetilde{z}_{t-1}) \oplus c_t \tag{4.3}$$

$$y_t = W_{voc} \cdot \max(W_1 x'_t + b_1, W_2 x'_t + b_2) + b_{voc} \tag{4.4}$$

where $\oplus$ does a vector concatenation. $W_{att} \in \mathbb{R}^{k \times 3n}, v_{att} \in \mathbb{R}^k$ and $b_{att} \in \mathbb{R}^k$ are the parameters of the attention model. $E' \in \mathbb{R}^{|V'| \times m}$ is the target embedding matrix. $W_1, W_2 \in \mathbb{R}^{\frac{n}{2} \times (3n+m)}$ and $b_1, b_2 \in \mathbb{R}^{\frac{n}{2}}$ are the parameters of the maxout layer. $\max$ does an element-wise maximum between its two arguments.[17] $W_{voc} \in \mathbb{R}^{|V'| \times \frac{n}{2}}$ and $b_{out} \in \mathbb{R}^{|V'|}$ do a linear projection of the output of the maxout layer to vocabulary size $|V'|$. y_t is a vector containing unnormalized

[15] Actually, all the data is already tokenized, and distributed on: `http://www-lium.univ-lemans.fr/~schwenk/nnmt-shared-task/`. For BLEU evaluation, the standard `multi-bleu.perl` is used (Koehn 2010).

[16] Official implementations, which use Theano, are available here: `https://github.com/lisa-groundhog/GroundHog` and here: `https://github.com/mila-udem/blocks-examples`

[17] This is a convenient way of implementing maxout, which takes the maximum of every two consecutive values in a vector.

scores for each item in the vocabulary. The decoder can output a new symbol by taking the argmax of this vector (greedy decoding), or compute a probability distribution over the entire vocabulary using a softmax layer. $\widetilde{z}_{t-1}$ is either the previously generated token (at decoding time), or the previous target token (teacher forcing). We optimize a cross-entropy loss between the softmax output and the target translations. The loss is normalized by batch size, but not by the number of time steps.

Like Bahdanau et al. (2015), we use a cell size of $n = 1000$, an embedding size of $m = 620$ and an attention size of $k = 1000$. We initialize all (non-bias) parameters with a centered normal distribution with $\sigma = 0.01$, except for recurrent weight matrices (inside the GRUs), which are initialized to random orthogonal matrices.

We use AdaDelta with a learning rate of 1.0 and a batch size of 80. We manually stop training when performance on the dev set does not seem to be improving anymore. We cycle through the entire training set, by reading the next 1600 sentence pairs, sorting them by target length, grouping them into 20 batches, and iterating randomly through this set of batches. This limits the amount of padding and reduces the average sentence length.

We also use gradient clipping with a maximum global norm of 1.0. We do not use any dropout as the training set seems large enough not to overfit.[18] We save and evaluate our model (with a greedy decoder) every 10 000 SGD steps, and keep the checkpoint with highest BLEU score on the dev set for final evaluation on the test set.

We use vocabulary shortlists with the 30k most frequent source and target words, and replace out-of-vocabulary words with a special `UNK` token. We set the maximum length to 50 words for source and target sequences. Longer sequences are truncated. We add an extra `EOS` token to each source and target sequence and pad the sequences to fit the maximum length in the batch with extra `EOS` symbols.[19] Source sequence reversing (Sutskever et al. 2014) is unnecessary because of the bidirectional encoder and the attention model.

This baseline model (called "AdaDelta" in the tables) is identical to Bahdanau et al. (2015)'s RNNsearch model. We also train variants (with a different decoder or training algorithm) that are described later.

UNK replacement Following Jean et al. (2015a), we extract a bilingual dictionary using `fast_align` (Dyer et al. 2013), and use this dictionary at decoding time to replace unknown words. Each time an `UNK` token is generated, we use the attention mechanism to get the source symbol with the largest weight:

$$i = \arg \max_{i \in \{1,\dots,T\}} \alpha_{ti} \tag{4.5}$$

$$\text{align}(\widetilde{z}_t) = x_i \tag{4.6}$$

Like Jean et al. (2015a), if x_i starts with a lowercase letter, then we look for a translation in the bilingual dictionary, and substitute the `UNK` token by this translation. If no translation is available, or the word does not start with a lowercase letter (more likely to be a proper noun or number), then we replace with the source word x_i itself.

[18] In all our experiments we do 3 full epochs at most, which seems hardly enough to overfit.

[19] We use a mask of zeros and ones in the attention model to avoid looking at the padding tokens (the attention model can look at the first `EOS` but not beyond). Similarly, the training loss does not take into account the tokens generated after the first `EOS`.

Jean et al. (2015a) do not detail how they obtain this bilingual dictionary. We use `fast_align` with the default settings to get alignments in both directions for the WMT14 training data. Then we symmetrize the alignments using `atools` with the `grow-diag-final-and` symmetrization heuristic (Koehn 2010). Then we simply build an English-to-French dictionary by using a simple procedure: for each English word, we get the French word which is most often aligned to this English word.

Modifications We also train a variant of RNNsearch that uses Adam (Kingma et al. 2015) instead of AdaDelta (Zeiler 2012) for optimization. This algorithm achieves considerably better convergence speed (at least with RNNsearch). We start with a learning rate of 0.0002 (the standard value of 0.001 is too high) and halve it every half epoch. In the table of results, this approach is called "Adam".

We also use the conditional GRU from Sennrich et al. (2017) and use a dense non-linear layer of size 620 instead of the maxout layer from (Bahdanau et al. 2015). Also, instead of using the last backward state for decoder initialization, we average all encoder hidden states. Finally, instead of being initialized to zero, the first states of the encoder are trained parameters. We call this approach "Adam + CGRU".

Finally, we train the same model but with subword units instead of words. For this purpose, we train a joint Byte-Pair Encoding model (Sennrich et al. 2016c) on a concatenation of source and target training data, with $30\,000$ merge operations. We transform all the data (train, dev, and test) into subword units using this joint BPE model. We use the same source and target vocabulary (with approximately 35k BPE units) and share the embeddings matrix between the encoder and decoder ($E = E'$). We call this model "Adam + CGRU + BPE". We merge these subword units before performing BLEU evaluation against the (word-based) target side of the dev set. Compared to the word-based model which generates sequences of up to 50 words, we increase the limit to 60 BPE units. We found that this was necessary for the model to be able to generate long sequences, as the BPE sequences are 15% longer on average.

Finally, similarly to Jean et al. (2015a), we take our best checkpoint (of "Adam + CGRU + BPE"), and continue training with pure SGD (without Adam), while keeping the embeddings fixed. We start with a learning rate of 1.0 that we decay by half every quarter epoch. We stop training when the dev loss does not seem to be improving. We call this model "SGD finetuning".

Results Table 4.6 shows our results, along with comparable models from the literature and SMT baselines. Finally, we also show for comparison the best results to date on this task (with single models).

We observe that our baseline copy of RNNsearch achieves the same results as those presented in the original paper (Bahdanau et al. 2015). However, it lags a little bit behind Jean et al. (2015a)'s version of this model. We see that our versions of RNNsearch that uses Adam for training achieve comparable results. Using subword units instead of words (Adam + CGRU + BPE) achieves a large boost in translation quality, even better than RNNsearch with `UNK` replacement. This model is about as good as the large-vocabulary model of Jean et al. (2015a), and much better than a baseline SMT model with comparable amounts of training data (Cho et al. 2014b).

When using an ensemble of large-vocabulary models (several instances of the same model whose predictions are averaged), along with their `UNK` replacement technique, Jean et al. (2015a)

Model	Data	Vocab	Test BLEU	+UNK
SMT				
Baseline Moses (Cho et al. 2014b)	12M + LM	All	33.3	
SMT SOTA (Durrani et al. 2014)	36M + LM	All	37.0	
RNNsearch				
Bahdanau et al. (2015)	12M	30k	28.5	
Jean et al. (2015a)			30.0	33.1
Large Vocab (Jean et al. 2015a)		500k	32.7	**34.6**
AdaDelta (default)	12M	30k	29.2	31.1
Adam			29.9	31.9
Adam + CGRU			30.6	32.8
Adam + CGRU + BPE		35k (BPE)	34.7	
SGD finetuning			**34.8**	
State-of-the-Art				
Deep-Att (Zhou et al. 2016)	12M	30k	35.9	
	36M	80k	37.7	
GNMT (Wu et al. 2016)			37.9	
		32k (WPM)	39.0	
Transformer (Vaswani et al. 2017)			**41.0**	
ConvS2S (Gehring et al. 2017a)		40k (BPE)	40.5	

TABLE 4.6: Results of different MT models on the WMT14 English-to-French translation task (test set). The 12M training set (which we use) is a filtered version of the larger 36M training set. The table shows three categories of approaches (from top to bottom), SMT models, attention-based models, and larger state-of-the art models. The test BLEU scores are obtained with a beam search decoder. The last column shows the test BLEU score obtained when applying the unknown-word replacement technique. Only attention-based word-based methods are eligible. The Vocab column shows the size of the target vocabulary size. WPM corresponds to a Word Piece Model (Wu et al. 2016), and BPE to Byte Pair Encoding (Sennrich et al. 2016c), two kinds of subword techniques. Cho et al. 2014b and Durrani et al. 2014 used language models estimated respectively on 16M lines (418M French words), and 3B lines (49B French words).

obtain a BLEU score of 37.2. A similar model model by Luong et al. (2015c) obtains 36.9 on the 12M training set, and 37.5 on the full 36M training set.[20] This is comparable to the best SMT results to date on this task (Durrani et al. 2014), obtained with the full 36M training set and the enormous Common Crawl corpus for language modeling (49B words).

Since then, much larger models have been developed that take advantage of the full training set, and achieve a huge boost in translation quality (Gehring et al. 2017b; Shazeer et al. 2017; Vaswani et al. 2017; Wu et al. 2016; Zhou et al. 2016). However, these models require orders of magnitude more computation power to train.

Table 4.6 shows results obtained with single models only (not ensembles). GNMT achieves a score of 41.2 when used in an ensemble of 8 models (+ Reinforcement Learning finetuning), and ConvS2S obtains 41.6 with an ensemble of 10 models. This is, to the best of our knowledge,

[20] This model uses a deep encoder and decoder with LSTMs, but no attention model. It also uses an unknown-word replacement technique.

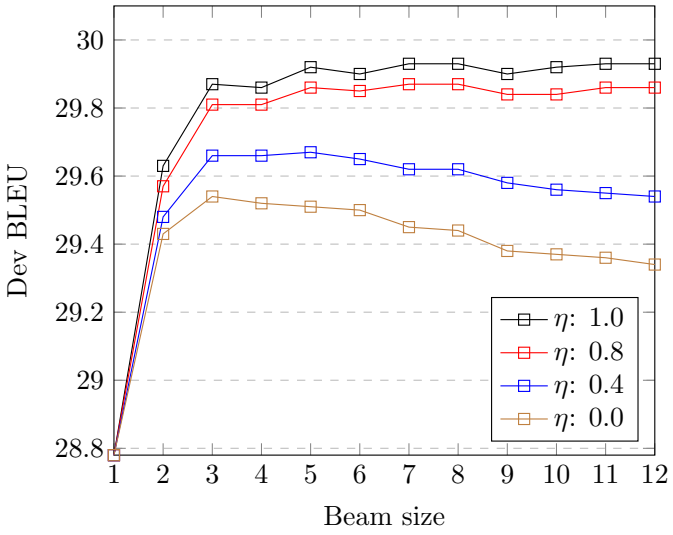


FIGURE 4.1: BLEU score of the "Adam + CGRU + BPE" model on WMT14 dev set depending on beam size, and length normalization coefficient (η). It is important to note that the y-axis does not start at 0, but at 28.8, which is the BLEU score obtained with greedy decoding (i.e., a beam size of 1).

the best score to date on this task. However, a single GNMT model takes 6 days to train on 96 NVIDIA K80 GPUs, and a single ConvS2S model takes 37 days to train on 8 GPUs.[21]

A model that shows great promise is the Transformer (Vaswani et al. 2017). It obtains a single model score of 41.0, very close to the ensemble performance of the best models to date. Even more interestingly, a large Transformer takes 3.5 days to train on 8 NVIDIA P100 GPUs, an order of magnitude less than GNMT or ConvS2S.

An interesting thing to note from the table is that using subword units (BPE or WPM) brings a consistent improvement over word-based translation, even with UNK replacement techniques. Also, using the larger 36M corpus seems to improve the results, both in NMT (Zhou et al. 2016) and SMT (Durrani et al. 2014). However, as the corpus is noisier, this may require larger models and longer training to take advantage of (and as a consequence, more computation power). When training RNNsearch with Adam, we do approximately 3 full epochs on the 12M training set. This would correspond to a single epoch on the noisier 36M training set.

All results in Table 4.6 use beam search decoding, with a varying beam size.[22] Our own results are obtained with a beam size of 8. Our beam search decoder reduces its beam size by one each time it finds a new finished hypothesis (with an `EOS` token), resulting in exactly 8 candidates at the end of decoding. We normalize the 8 final scores by hypothesis length (number of tokens), and rerank the hypotheses. This corresponds to a length penalty (η) of 1.0. The scoring formula is: $score(w_1, \dots, w_T) = \frac{\sum \log p(w_t)}{T^\eta}$, where T is the number of tokens up to (including) the first `EOS` token.

Figure 4.1 shows the BLEU scores of our best model ("Adam + CGRU + BPE") on the dev set, depending on the beam size and length penalty η used during decoding. We see that, while

[21] This is highly above our resources. As a measure of comparison, our `Adam+CGRU+BPE` model took 4 days to train on a single NVIDIA Titan X.

[22] Jean et al. (2015a) use a beam size of 12 with length normalization. Wu et al. (2016) use a more sophisticated beam search decoder with tuned coverage penalty, length penalty and early pruning. Vaswani et al. (2017) and Zhou et al. (2016) use smaller beam sizes of 5 and 3 respectively.

BLEU scores are not very sensitive to beam size, length normalization is important. An explanation is that the unnormalized score of a hypothesis is the product of the probabilities (or sum of log probabilities) of each of its words. This strongly disadvantages longer hypotheses, which results in unnaturally short translations (compared to greedy decoding), which are penalized by BLEU's brevity penalty.

The best scores are obtained with a length penalty of 1.0, and a beam size between 6 and 12. When using no length normalization ($\eta = 0.0$), BLEU scores tend to decrease with larger beam sizes. When measuring the brevity penalty of BLEU (which penalizes shorter translations), we observe indeed that the $\eta = 1$ curve has the same length ratio of 0.96 all through. On the other hand, the $\eta = 0$ (unnormalized) curve's length ratio drops from 0.96 to 0.93, which explains the drop in BLEU scores.

The base length ratio of 0.96 can be explained by the fact that the model is trained to output sequences with a maximum length of 60 subword units. It does not know how to produce longer sequences. We evaluated the performance of the same model (beam size of 8, $\eta = 1$) on the dev set with sequences of 60 tokens at most (5607 sentences pairs out of 6003 for the real dev set). The BLEU score is then 30.6 (vs. 29.9), and the length ratio is 0.993, very close to 1. We see that with the proper length normalization scheme, the beam search decoder is able to produce sequences of the right length.

Analysis Figure 4.2 shows the evolution of dev BLEU score and training loss during training for four models: the baseline RNNsearch model trained with AdaDelta or Adam, our modified model that uses a conditional GRU, and our BPE model. The training loss of the BPE model is not directly comparable to the other two, as the sequences are longer on average. We see that the model trained with Adam converges much faster than AdaDelta, and to a better BLEU score. Using a conditional GRU for the decoder (CGRU) slightly increases the model's performance. We observe similar performance as the CGRU decoder when using a single GRU with a "generate first" strategy and a maxout layer (not shown here).

Figure 4.3 shows the test BLEU score (with beam search) of these three models, depending on target sequence length. We see that their BLEU scores quickly degrade when the sequence length exceeds 50 words. This is due to the fact that they are trained with a maximum source and output length of 50 words (60 subwords for the "Adam + CGRU + BPE" model), and do not know how to generate longer sequences (even when increasing the maximum output length at decoding time). The decrease in BLEU scores is due to the brevity penalty component of BLEU that penalizes shorter hypotheses. We see that the models are robust to long sequences, provided that they were trained with examples of long sequences.

Figure 4.4 shows examples of alignments by the "Adam + CGRU" and "Adam + CGRU + BPE" models. Even though the attention model is trained in an unsupervised manner (the only supervision is the target sequence, we do not have a ground truth alignment), it is able to learn a sensible soft-alignment between the source and the output. It is able to capture simple word reordering, like `closed factory` with `usine fermée`, and to align phrases, like `be back` with `être de retour`, or `said` with `a dit que`. When the decoder produces an `UNK` token, it seems like the attention model is able to align it with the correct symbol in the input.

It is interesting to note that to produce the `ons` suffix to `imaginons`, the attention model looks at the `'s` symbol following `let`, which is the contraction of `us` (the subject of the verb). On the other hand, to output `montrent`, the model does not have to look at the corresponding `stats` subject. A possible explanation for this is that the decoder has already

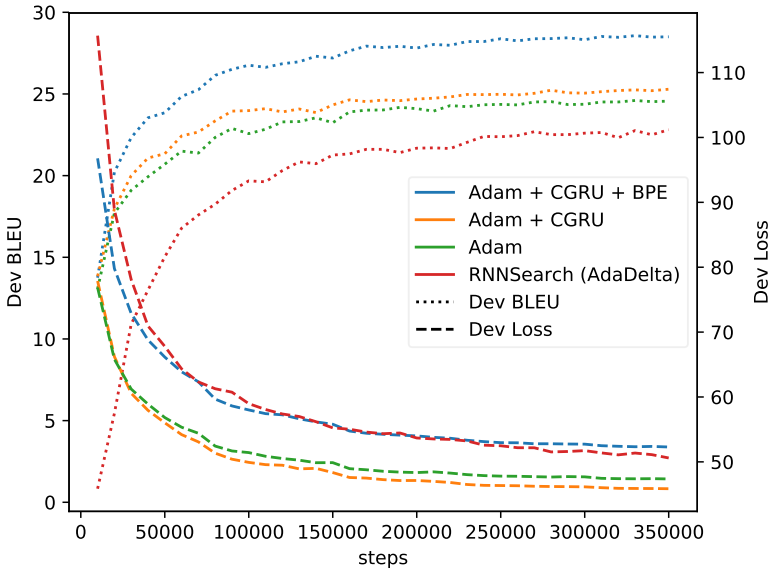


FIGURE 4.2: Evolution of dev BLEU score and training loss during training. A training step corresponds to a full mini-batch of 80 sentences. Since the `Adam+CGRU+BPE` method has a different target, its training loss is not directly comparable to the other two methods. The training loss is not divided by the number of target tokens, which explains why this method gets a higher loss (longer sequences). With the Adam models, the learning rate is decayed by half every 70k steps.

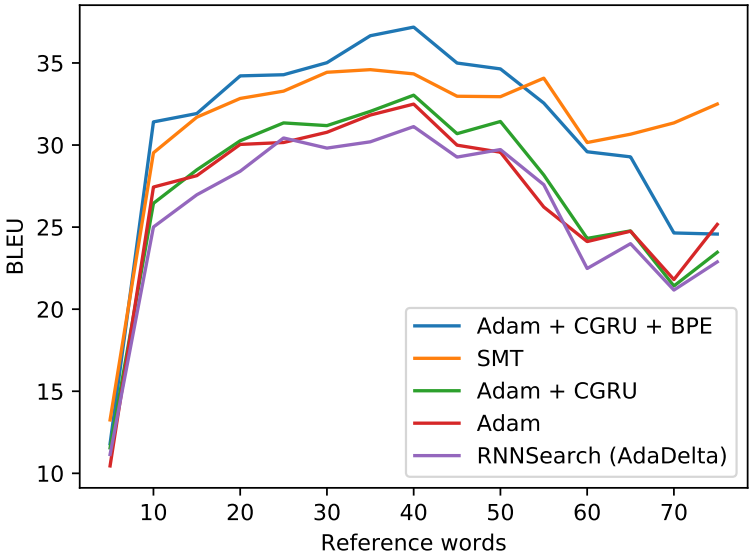


FIGURE 4.3: Test BLEU score as a function of reference sentence length (in terms of words). Each point whose x-value is a corresponds to a bin of sentences of length $a-4$ to a (the first point corresponds to sentences of length 1 to 5, and the last one 76 to 80). Since the test set contains 3003 sentences and most of the sentences are between 10 and 50 words, the extremities of the graph are noisier. At decoding time, we set the maximum source and output length to 80 for the word-based models (vs. 50 during training), and 100 for the BPE-based models (vs. 60 during training). It turns out that increasing the maximum sequence length has a negligible effect on BLEU scores. The SMT baseline is that of (Cho et al. 2014b).

produced `statistiques`, and thus its state already contains information about the number of the subject. We observe the same thing with `elle` and `heureuse`. This is not the case for `imaginons`, where no subject has been produced yet, and the decoder has no other choice but to look in the input sequence to find the subject of the verb.

Surprisingly, the attention model often gives high weights to the end-of-sentence symbol (last column in the pictures). A possible explanation for this is that the last state of the encoder contains information about the entire sequence, and is a good substitute when the attention model is unsure. We could argue that in a bidirectional encoder, all encoder states contain information about the entire sequence. This may be true, but this information is split into a forward and backward state. This discontinuity probably makes this kind of 'concatenated' state harder to interpret. The last forward state on the other hand supposedly contains an abstract representation of the entire sentence. The backward states are less natural, as they read the sequence from right to left, which may result in poorer representations.

Figure 4.5 gives examples of outputs by these two models (word-based and BPE-based).

We see that the unknown word replacement technique often gives satisfying results, in particular with proper names (`Zénith` and `M83`). However, it sometimes fails when the attention model is mistaken (example 1), or when the dictionary is wrong (example 6 with `flush`). Even when it works, the dictionary method has limits as it does not take into account the context of the word. For example it translates `scared` with `peur` (a verb with a noun), or `she...upset` with `elle...bouleversé` (should be `bouleversée`). Even though these translations are wrong, they are better than `UNK` symbols. The BPE-based method does not need this technique, as it can read and generate any word. For example, we see in example 1 that contrary to the word-based method, the BPE-based method is able to translate `$ 32.9 billion` by splitting the numbers. Sometimes, the BPE-based model hallucinates new words that don't exist, like `effondée` in example 7.

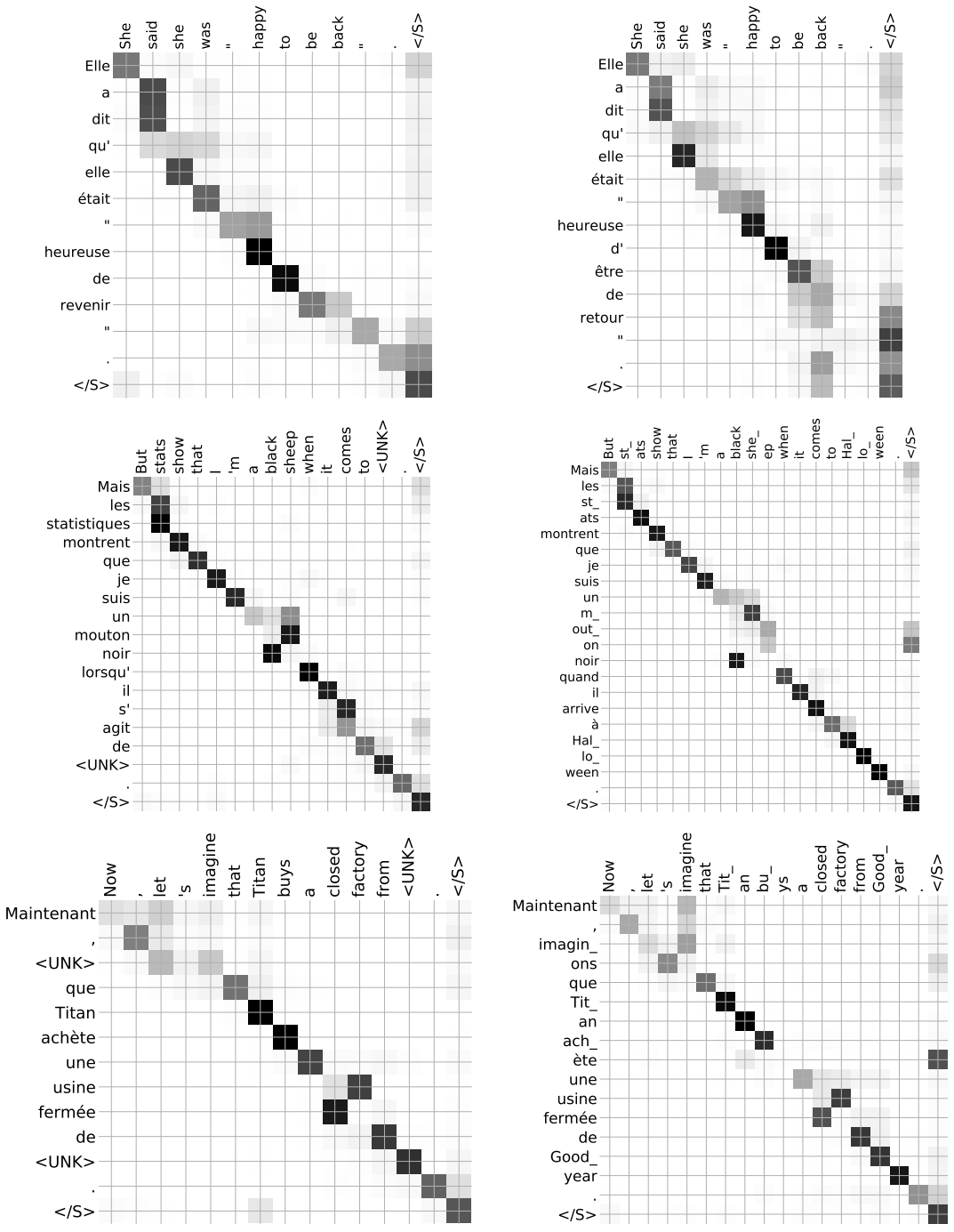


FIGURE 4.4: Examples of alignments on the WMT14 test set by the best word-based (left) and subword-based (right) models, with a beam size of eight. The alignment is not forced, which means that the target side (on the left of the graphs) is the output of the model, and not the translation reference. White squares correspond to an alignment score of zero, while black squares correspond to a score of one. Out-of-vocabulary symbols are replaced by an `<UNK>` symbol. The subword units are delimited by an underscore. For example: `Tit_` and `an` give `Titan`

1. The backlog in the aerospace division **was** $ 32.9 billion as of September 30 , unchanged from December 31 .
   L' arriéré dans la division aérospatiale était de **a** milliards de dollars à compter du 30 septembre , soit inchangé depuis le 31 décembre .
   The back|log in the aero|space division was $ 3|2.9 billion as of September 30 , unchanged from December 31 .
   L' arri|éré dans la division de l' aéro|spatiale s' élevait à 3|2,|9 milliards de dollars à compter du 30 septembre , soit le 31 décembre .

2. You have a wonderful baby and enjoy the fun .
   Vous avez un bébé merveilleux et profitez du plaisir .
   You have a wonderful baby and enjoy the fun .
   Vous avez un bébé merveilleux et appréci|ez le plaisir .

3. The shows are eagerly expected by the time they reach a provincial **Zénith** (theatre) .
   Les spectacles sont attendus avec impatience au moment où ils atteignent un **Zénith** provincial (théâtre) .
   The shows are e|ag|erly expected by the time they reach a provincial Z|éni|th (theatre) .
   Les spectacles sont attendus avec impati|ence au moment où ils atte|ignent un Z|éni|th provincial (théâtre) .

4. And this shape is definitive of the entire piece : it is a series of physical representations of drawings .
   Et cette forme est définitive de toute la pièce : il s' agit d' une série de représentations physiques de dessins .
   And this shape is defin|itive of the entire piece : it is a series of physical representations of drawings .
   Et cette forme est définitive de toute la pièce : il s' agit d' une série de représentations physiques de dessins .

5. Of these , 200 have still not found a new job .
   De ce nombre , 200 n' ont toujours pas trouvé de nouveaux emplois .
   Of these , 200 have still not found a new job .
   De ce nombre , 200 n' ont toujours pas trouvé de nouveaux emplois .

6. The aim of the assault is to **flush** the **M23** out of the hills overlooking Bunagana .
   Le but de l' assaut est d' **d'** les **M23** des collines qui donnent sur . .
   The aim of the assault is to fl|ush the M|23 out of the hills overlooking Bun|ag|ana .
   Le but de l' ass|aut est de fl|â|ner le M|23 sur les collines sur|plom|b|ant Bun|ag|ana .

7. She came home from school **scared** and **upset** to be the first among her friends .
   Elle est venue à la maison **peur** et **bouleversé** d' être la première parmi ses amis .
   She came home from school scar|ed and up|set to be the first among her friends .
   Elle est venue à la maison de l' école qui s' est eff|on|dr|ée et qu' elle a été le premier à être la première de ses amis .

8. The building damaged by the fire contained four apartments , but there was nobody at home when the fire started .
   Le bâtiment endommagé par le feu contenait quatre appartements , mais il n' y avait personne à la maison lorsque le feu a commencé .
   The building damaged by the fire contained four apartments , but there was no|body at home when the fire started .
   Le bâtiment endommag|é par le feu contenait quatre appartements , mais il n' y avait personne à la maison quand le feu a commencé .

FIGURE 4.5: Examples of beam search outputs on the test set by our word-based and BPE-based models. In each item, the first and second line are the word-based input and output. The third and fourth line are the BPE-based input and output. The '|' symbol delimits BPE units. UNK tokens in the output that have been replaced are in bold, along with the source word to which they are aligned. Source words that are out-of-vocabulary are also underlined.

Corpus		Total Lines	German		English	
			Words	Average	Words	Average
IWSLT14	Train	153k	2.69M	17.5	2.84M	18.5
	Dev	6969	122k	17.6	129k	18.5
	Test	6750	126k	18.6	131k	19.4
TED		1.81M	34.4M	18.9	36.4M	20.0
OpenSubtitles		40.1M / 430M	286M	7.0	3.21B	7.5

TABLE 4.7: IWSLT14 corpus statistics. TED is initially an English-language monolingual corpus. We generated the German side by back-translation. OpenSubtitles, which we use for language modeling, consists in two monolingual German and English corpora.

4.3.2 TED Talks (IWSLT14)

We now perform a series of experiments on a German-to-English translation task. This task consists in translation of subtitles of TED and TEDx talks. The data was released in the context of IWSLT 2014 evaluation campaign.[23]

We use the same data split (train, dev and test) as Ranzato et al. (2016). Several recent contributions have evaluated their NMT models on this corpus (Bahdanau et al. 2017; P.-S. Huang et al. 2018).

This dataset is small by MT standards: 153k sentence pairs for the training set, and 7k each for the dev and test sets. We also use monolingual datasets for language modeling or data augmentation. For this purpose, we use the English TED monolingual corpus,[24] which contains 36M words. We remove any line that also appears in the target side of the IWSLT14 dev or test sets (to avoid biasing the evaluation). We also use the large OpenSubtitles corpora (for English and German), which are in a similar domain as the IWSLT corpus (subtitles of movies and TV shows). Table 4.7 gives detailed statistics about these resources. We see that the TED corpus has a very similar average segment length as the IWSLT corpus. This is understandable as they both come from the same source (subtitles of TED talks). OpenSubtitles has much shorter occurrences on average. This is due to the fact that it contains many dialogues (from movies), while TED talks are essentially monologues.

All the data is already tokenized. For fair comparison with the existing literature (Bahdanau et al. 2017; Ranzato et al. 2016), the evaluation is done with tokenized case insensitive BLEU. For this reason, we lowercase all the data. For the IWSLT parallel data, we use the same pre-processing script as Ranzato et al. (2016).[25]

SMT baselines As a baseline, we train several phrase-based machine translation models using Moses (Koehn 2010). The first model, named 'SMT' uses the default Moses settings, with a trigram language model estimated on the target side of the parallel corpus. We also train two SMT models that use additional monolingual data for the language model. The 'SMT + LM' model uses a trigram language model estimated on the English side of the TED corpus (36M words), concatenated to the target side of the IWSLT corpus. The 'SMT + Large LM' model uses a language model of order 5, which is estimated on the large OpenSubtitles dataset (3.2

[23] The IWSLT14 corpus is available here: https://wit3.fbk.eu

[24] The English TED corpus is available here: http://opus.nlpl.eu/TED2013.php

[25] Ranzato et al. (2016)'s source code and pre-processing scripts are available here: https://github.com/facebookresearch/mixer

billion words). Finally, as a measure of comparison with NMT, we also train a baseline SMT system on BPE-segmented data ('SMT + BPE'). All the SMT models are tuned on the dev set with MERT (using the `mert-moses.perl` script).

NMT models The 'Basic' NMT model is a simplified (and smaller) version of Bahdanau et al. (2015)'s RNNsearch. Its (bidirectional) encoder and decoder use LSTM cells instead of GRUs (we found LSTMs to perform better on this task), with a cell size of $n = 256$ and an embedding size of $m = 128$. The encoder and decoder's initial states are set to zero.[26] And we do away with the maxout layer in the decoder. Instead, we do a linear projection to the embedding size, followed by a linear projection to the vocabulary size. Finally, the previous time step's prediction is not used for predicting the next symbol (only for updating the LSTM).

Equation 4.4 is modified as follows:

$$s_{t-1} : \text{LSTM's output} \qquad c_t : \text{attention vector} \tag{4.7}$$

$$y_t = W_{voc} W_{out}(s_{t-1} \oplus c_t) + b_{voc} \tag{4.8}$$

where $W_{out} \in \mathbb{R}^{m \times 3n}$, $W_{voc} \in \mathbb{R}^{|V'| \times m}$, and $b_{voc} \in \mathbb{R}^{|V'|}$, with $|V'|$ the target vocabulary size. y_t is a vector of unnormalized scores for each item in the target vocabulary, which can be used for greedy prediction (argmax), or to estimate a probability distribution over the vocabulary (softmax). This simple linear projection to embedding size is useful to reduce the number of parameters (instead of a direct projection to vocabulary size). It also makes weight tying possible (Press et al. 2017), where the output embedding matrix E' is used in place of W_{voc} (we do not actually do this).

Contrary to RNNsearch, we use a "generate first" strategy, where we generate the next symbol *before* updating the LSTM's state. We found this strategy to perform better than the reverse.

The model is trained with Adam, with a batch size of 32 and a learning rate of 0.001. The maximum source (German) sequence length is 45 words, and the maximum target (English) length is 47 (which covers 99% of all training sentences). All parameters (except biases) are initialized to a centered normal distribution with $\sigma = 0.1$. The source and target vocabularies contain the most frequent 30k tokens each.

We also train a variant of this model with dropout regularization ('Basic + dropout'). We use a dropout rate of 20% on the input of the LSTMs (both in the encoder and decoder). We also drop source and target words at random during training with a probability of 0.2 (their embeddings are replaced with zero). We tried to apply dropout in other places (on the output of the LSTMs, or in the attention model), but this was not useful, and sometimes even detrimental. Variational dropout (Gal et al. 2016) was not useful either.

The 'Advanced' model is identical except for its decoder, which uses a conditional LSTM (Sennrich et al. 2017). It also replaces the linear projection layer (W_{voc}) with a non-linear layer of the same size (with a bias vector, and a $\tanh$ activation).

Then, we train the same model but with BPE units instead of words ('BPE' model). 30k joint BPE merge operations are extracted from the IWSLT training data (by concatenating the source and target side). Then all the data is segmented using this BPE model. This gives a joint German-English vocabulary of size 27k.[27] The same embedding matrix is shared between the

[26] We found that initializing the decoder's state with the encoder's last state, like RNNsearch does, was not useful.

[27] Following Sennrich et al. (2016c)'s recommendations, we exclude BPE units that appear less than 10 times in the training files. This helps avoid segmentations in one language that are unknown in the other language.

encoder and the decoder ($E = E'$). The maximum sequence length is set to 52 subword units for German and 50 subword units for English (also to cover 99% of the training corpus).

We try two version of this model: one with the same size as the previous models ($n = 256$), and a larger one ($n = 512$), called 'BPE XL'. The larger model also uses a larger dropout rate of 0.4 on the LSTM's inputs (we keep the same word dropout).

The 'BPE to char XL' model is the same model, but with characters as target (and BPE units as source). The maximum output sequence length is set to 200. The target character-level vocabulary contains 140 tokens, including the whitespace character, that the model is trained to predict like any other character.

We train the word-based models and 'BPE' model for 200k steps. The 'BPE XL' model is trained for 400k steps and the 'Char XL' model for 800k steps. Some models are trained on larger amounts of synthetic data (see below). Those are trained for twice as long as the models trained on real data only, with a maximum of 800k steps. We sometimes stop early manually, when the performance has obviously stopped improving.

Monolingual data The training corpus is rather small (153k sentence pairs). However, we have access to large amounts of monolingual data (1.8M TED English sentences). In NMT, there exist two main approaches for improving a model with target-language data:

- Training an external language model (Gulcehre et al. 2015). This can be a statistical language model or a recurrent neural network. This model can be combined with the translation model at decoding time, using a log-linear model (shallow fusion). Gulcehre et al. (2015) also propose a 'deep fusion' scheme, where they concatenate the decoder states of a translation model with those of neural language model, and finetune this merged model.
- Producing synthetic parallel data to increase the size of the training corpus. Sennrich et al. (2015) propose to use monolingual data as the target side of a parallel corpus. As the source side, they either use a single dummy symbol for each sentence, or back-translate the entire corpus using another MT system. This synthetic corpus is concatenated to the real training corpus and shuffled.

We prefer using the second approach, as it is easier to implement (it requires no change in the code), and can work at any granularity level (words, characters or subwords). The first approach, on the other hand, would require training language models at each granularity level. Yet, these two approaches may be compatible (e.g., back-translation of a medium-sized monolingual corpus, and language modeling on a large corpus), which could be interesting to investigate as future work.

For back-translation, we train a baseline SMT system from English to German (reverse direction), on the IWSLT train set. The language model (of order 3) is estimated on a concatenation of the German side of IWSLT and the German OpenSubtitles corpus (286 million words). Then, we use this SMT system to translate the English TED monolingual corpus to German (1.8 million sentences). This gives a synthetic German-English parallel corpus of TED talks, which is an order of magnitude larger than the initial training corpus. We concatenate this corpus with the initial corpus oversampled 10 times. The final parallel corpus contains about 3.3 million sentence pairs. This oversampling strategy is roughly equivalent to that of (Sennrich et al. 2015),

Model	Data	Dev BLEU		SGD steps
		Greedy	Beam search	
Basic	153k	26.6	28.5	24k
Basic + dropout		28.3	29.7	104k
Advanced		29.5	30.8	184k
BPE		31.6	32.9	200k
BPE XL		32.7	34.0	300k
Ensemble of 5		35.4	**36.7**	–
BPE to char XL		32.8	34.3	792k
Advanced	153k + 1.8M	30.0	31.3	252k
BPE		31.8	33.2	292k
BPE XL		32.9	34.1	396k
BPE to char XL		33.0	**34.4**	792k

TABLE 4.8: Results of different NMT models on the IWSLT14 dev set. We do beam search with a beam size of 8 and length normalization. The ensemble combines 5 instances of 'BPE XL'. The last column gives the number of SGD steps before reaching the best dev BLEU score.

where they resample at each new epoch a new subset of the back-translated corpus to match the size of the true corpus.

We retrain the same 'Advanced' and 'BPE' configurations on this larger training set (for twice as long).

Results Table 4.8 shows the BLEU scores of our NMT models on the dev set, using greedy decoding or beam search decoding with a beam size of 8. We see that adding dropout to the 'Basic' model gives a large boost in BLEU scores. Figure 4.6 shows the dev BLEU score (greedy) of our models during training. We observe indeed that the only model that does not use dropout overfits dramatically, as its performance starts decreasing after only 24k step (about 5 epochs). Simply adding 20% dropout removes this overfitting problem, at the cost of slightly slower training. The more advanced model that uses a deeper decoder achieves a higher BLEU score (+1 BLEU). Using subwords instead of words also gives a large boost in BLEU scores.

Our best results are obtained with the large BPE and BPE-to-Char models. The latter seems to perform slightly better, at the cost of a much longer training time. Training our models on the large back-translated corpus does not give a significant increase in scores.

Table 4.9 shows the BLEU scores of our approaches on the test set, along with the SMT baselines, and other results published in the literature. Interestingly, NMT models beat the phrase-based baseline, even with a very large language model. This indicates that NMT can be useful even with medium size corpora.

Like the WMT14 task, we observe a BLEU increase when using Jean et al. (2015a)'s unknown word replacement technique, and using subwords gives even better results. Our BPE-based models beats the previous best score on the task by P.-S. Huang et al. (2018). Our BPE-to-Char model gets a +2 BLEU over the best published result. Combining five BPE models in an ensemble gives a large boost, with +4 BLEU compared to the best published result (obtained with a single model).

This suggests that the REINFORCE training objective from (Ranzato et al. 2016), the Actor-Critic model from (Bahdanau et al. 2017), and the Neural Phrase-based MT model of (P.-S.

Model	Data	Test BLEU	
		Greedy	Beam search
MIXER (Ranzato et al. 2016)	153k	20.7	21.8
LL (Bahdanau et al. 2017)		25.8	27.6
AC + LL (Bahdanau et al. 2017)		27.5	28.5
NPMT (P.-S. Huang et al. 2018)		28.6	29.9
NPMT + LM (P.-S. Huang et al. 2018)		–	**30.1**
SMT	153k	–	27.4
SMT + BPE			25.4
SMT + LM	153k + 1.8M		26.7
SMT + Large LM	153k + 430M		**28.1**
Advanced	153k	26.7	28.3
Advanced + UNK replace		27.3	28.8
BPE XL		29.8	31.2
Ensemble of 5		32.7	**34.1**
BPE to Char XL		30.6	**32.2**
Advanced	153k + 1.8M	27.2	28.4
Advanced + UNK replace		27.9	28.9
BPE XL		30.2	31.6
BPE to Char XL		30.8	**32.3**
Google Translate (Wu et al. 2016)	–	–	37.4

TABLE 4.9: BLEU scores on the IWLST14 test set. The Google Translate baseline is obtained thanks to Google Translate API. We send it the raw test set (non-tokenized and non-lowercased) and post-process its output by tokenizing and lowercasing it before evaluation.

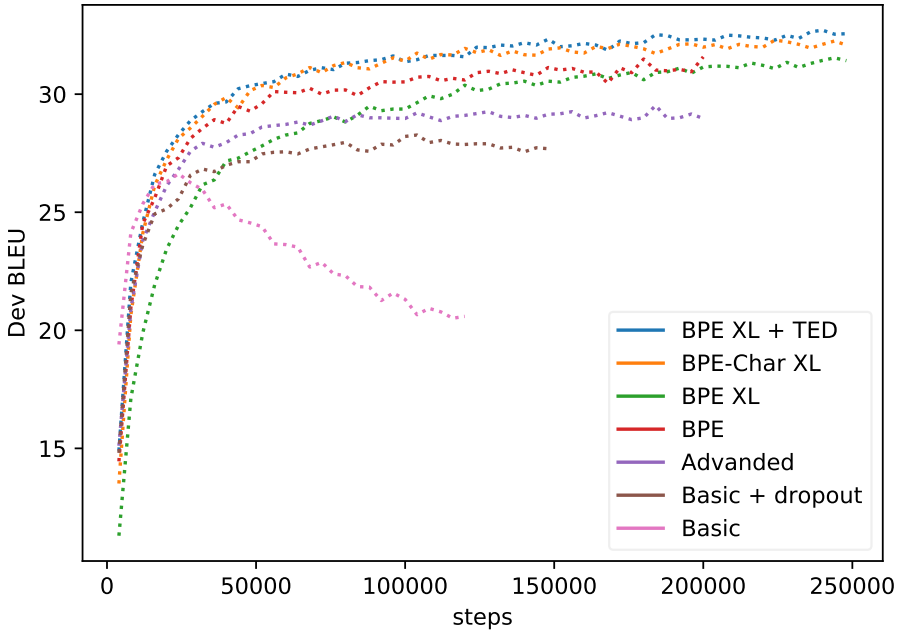


FIGURE 4.6: BLEU scores on the IWSLT14 dev set of our NMT models during training. The last 3 models are word-based, and all models use dropout except for the last one ('Basic'). The x-axis corresponds to the number of SGD steps performed, i.e., the number of batches of size 32 processed (100k steps corresponds to approximately 21 full epochs).

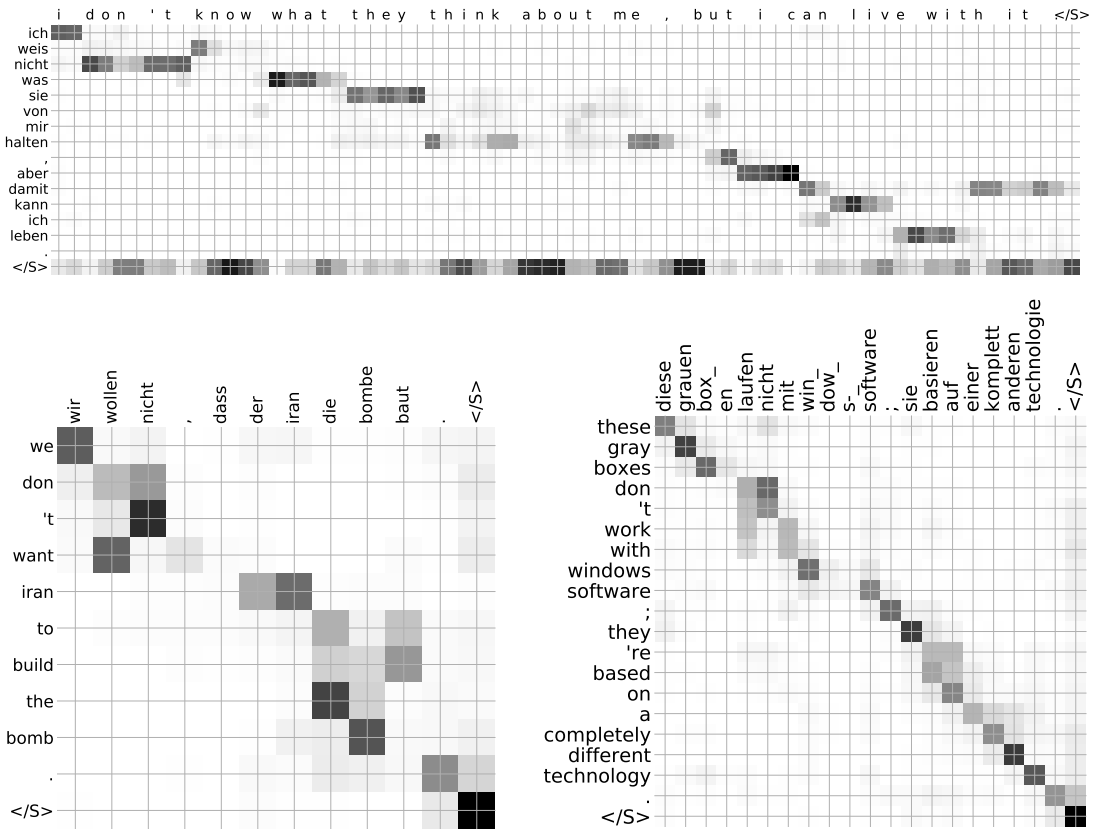


FIGURE 4.7: Examples of (non-forced) alignments on the IWSLT test set, by our 'BPE-Char XL + TED' model (top), and 'BPE XL + TED' model (bottom). The top alignment is reversed so as to better see the character sequence (which corresponds to the target).

Huang et al. 2018) should be compared against stronger baselines. It would be interesting to try Bahdanau et al. (2017)'s Actor Critic model on subword-level machine translation. Interestingly, our word-based NMT baseline obtains a BLEU score of 28.3, not very far from Bahdanau et al. (2017)'s log-likelihood baseline (also word-based). This is not so surprising as both models are very similar (bidirectional encoder with 256 units).

The Google Translate baseline obtains a staggering +3 BLEU over our best model. This suggests a large margin of improvement on this task. It is important to note that Google Translate uses a much larger model (GNMT), which is trained on large amounts of parallel data (Wu et al. 2016).[28] Yet, this BLEU score does not do GNMT justice as the Google Translate pipeline does its own pre-processing and post-processing (to be fairer, we should perform case-sensitive detokenized evaluation).

Figure 4.7 gives some examples of outputs and alignments by our models. Interestingly, the BPE to character model is able to find clear word boundaries. The attention models are also able to deal with German verb order (which often comes after the object). For example, `to build` is successfully aligned with `baut`, and `think` with `halten`. We see in the bottom right example that the subword model is able to recopy words that it does not know: `windows-software` is split into 4 subword units and translated as two words.

[28] It is also not completely unlikely that the test data is part of GNMT's training corpus, which could bias the results.

Model	Greedy	Beam search
BPE XL	17.9	19.0
Ensemble of 5	20.6	**21.9**
BPE XL + TED	18.6	19.8
BPE to Char XL	18.9	20.3
BPE to Char XL + TED	19.1	20.4
SMT	–	17.7
SMT + Large LM		19.5
MILA NMT (Jean et al. 2015b)	–	26.4
MILA Ensemble		28.4
Google Translate (Wu et al. 2016)	–	36.1

TABLE 4.10: BLEU scores on news-test 2015. The MILA results are from the best performing NMT models on the WMT 2015 German-to-English news translation task. We took their submitted outputs and post-processed them to be compatible with our evaluation (lowercased and tokenized BLEU). These models are trained on large amounts of news data (4.5M sentence pairs). The same post-processing was applied to Google Translate's outputs.

Out-of-domain translation Table 4.10 shows the results of our models, when evaluated on the WMT15 test set, which is in the news domain (out-of-domain translation). We see that our NMT models trained on TED talks perform considerably worse than the state-of-the-art models on this task (which were trained on large amounts of in-domain data).

However, they obtain comparable results to the baseline SMT models (that are trained on the same parallel data). Our ensemble model even performs considerably better than an SMT model that uses a very large language model. This contrasts with observations from Koehn et al. (2017) that NMT is catastrophic in out-of-domain translation.

Chapter 5

Speech Translation

In addition to Machine Translation, we explored the related task of Automatic Speech Translation. Prior to our own work, neural sequence to sequence models had not yet been applied to this task (or only to sub-problems like alignment), which made it a good fit for research study. We also had developed a framework for NMT that could be easily extended to other sequence to sequence tasks.

Speech translation consists in translating spoken language, either to text or speech in another language. Current speech translation systems integrate (loosely or closely) two main modules: source language speech recognition (ASR) and source-to-target text translation (MT) (Kumar et al. 2015; Post et al. 2013). In these approaches, the source language text transcript appears as mandatory to produce a text hypothesis in the target language. Optionally, a third module can be added which reads the translated text aloud using speech synthesis.

In this work, we are solely interested in text output. By "speech translation", we always mean automatic translation of recorded audio in a language to text in another language. ASR can be seen as a specific case of speech translation, where the source language and the target language are the same.

One major advantage of the sequence to sequence models is that they can take anything as input and as output, provided that they are sequences. Similar models have been used for speech recognition, where the input is a sequence of audio feature vectors, and the output is the text transcription (Bahdanau et al. 2016; Chan et al. 2016; Chorowski et al. 2015). It is relatively straightforward to port this kind of model to do end-to-end speech translation, i.e., speech translation without transcription. The encoder reads the speech input and builds a (hopefully universal) representation from it. This representation is then read by the decoder, which can be trained to output a transcription, or a translation directly.

Conventional ASR systems cannot be so easily adapted to do speech translation, because they do monotonicity assumptions, i.e., they generate output symbols by reading the input from left to right. However, the attention-based sequence to sequence models (trained with a negative log-likelihood objective) do no such assumption. The decoder can look anywhere in the sequence of encoder annotations (global attention), and it does not care about the format of the input, provided that the representation computed by the encoder is abstract enough.

There are several advantages to doing end-to-end speech translation (without transcription). Relaxing the need for source language transcription would drastically change the data collection methodology in speech translation, especially in under-resourced scenarios. For instance, in the

former project DARPA TRANSTAC (speech translation from spoken Arabic dialects), a large effort was devoted to the collection of speech transcripts. A prerequisite to obtain transcripts was often a detailed transcription guide for languages with little standardized spelling. Now, if end-to-end approaches for speech-to-text translation are successful, one might consider collecting data by asking bilingual speakers to directly utter speech in the source language from target language text utterances. Such an approach has the advantage to be applicable to any unwritten (source) language.

Furthermore, we can expect improvements from such a system, compared to cascading two systems. When chaining two (or more) individual models, their errors pile up. The downstream MT system, if trained on (clean) parallel data only, might not be robust to the noisy output of the upstream ASR system. Hopefully, the end-to-end approach would also require less hand engineering, and less training time than the conventional ASR-MT pipeline.

However, there are also limits. When chaining two models, it is easier to build two strong baselines, because in most language pairs there is much more transcription and translation data available, than transcriptions aligned with translations.

We can mitigate this problem by using tricks to make use of this data, such as multi-task training, where the encoder is shared with a transcription task, or the decoder is shared with a text translation task. It is also possible to build synthetic data by automatically translating the transcriptions, or by using speech synthesis over the source side of a parallel corpus.

In the first subsection, we present our early work on end-to-end speech translation on a synthetic corpus. The next subsection describes the construction of a new corpus for speech translation, based on audiobooks. The last subsection extends our work on speech translation to this new corpus.

5.1 Neural Speech Translation of Synthetic Data

5.1.1 Model Description

We propose a similar model to the LAS (Listen, Attend and Spell) model (Chan et al. 2016). This model was used for speech transcription (ASR), and we propose to adapt it for direct speech translation. In principle, it is very straightforward to take audio frames as input, instead of words or characters. However, a number of adjustments are needed for the model to learn anything and to be trained in a reasonable time. We apply this model on a synthetic speech translation corpus. This work was published as a short paper titled "Listen and Translate: A Proof of Concept for End-to-End Speech-to-Text Translation" (Bérard et al. 2016a).[1]

Encoder The encoder takes as input a sequence of pre-computed feature vectors: $f_1, \dots, f_T \in \mathbb{R}^r$. This sequence corresponds to frames in the audio signal, it has a much smaller time resolution than words or even characters (with our feature extraction process, a single vector corresponds to a frame of 10 ms). Because the complexity of the attention model is linear with respect to the input length, we need a way to make these sequences shorter.

The solution that Bahdanau et al. (2016) and Chan et al. (2016) propose for ASR, which we implement and evaluate on the speech translation task, is a deep pyramidal encoder. The encoder

[1] At the NIPS Workshop on End-to-End Learning for Speech and Audio Processing, in Barcelona (Spain).

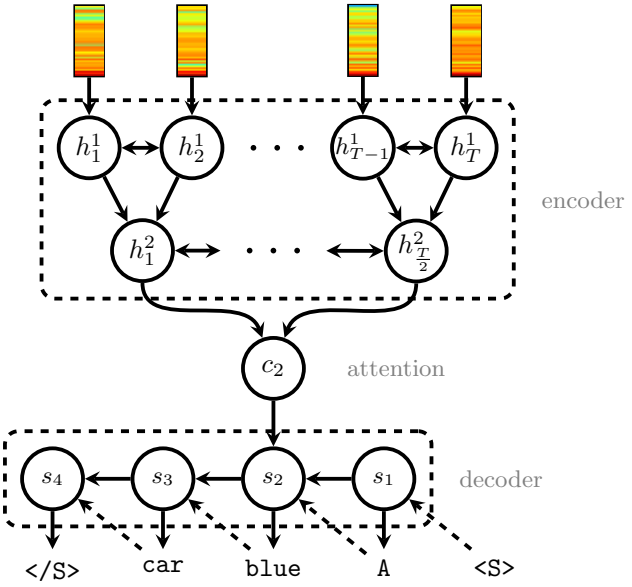


FIGURE 5.1: Simplified illustration of the pyramidal encoder-decoder model. The inputs are vectors of MFCCs (pre-computed features). The decoder looks at the hidden states of the last encoder layer (the shortest one).

has several layers of LSTMs, each layer shorter than the previous one, resulting in a much shorter sequence. The decoder reads from the top of the pyramid (the shortest layer). Figure 5.1 illustrates this pyramidal encoder (described in details shortly).

Before the first LSTM, we use two fully connected layers of size l and m. This helps computing better features, which is important for the encoder to produce a representation which is abstract enough to translate from. This gives a sequence $\mathbf{x}$ of the same length as the input sequence $\mathbf{f}$ and of dimension m.

$$f_i^{[2]} = \tanh\left(W^{[1]} f_i + b^{[1]}\right) \tag{5.1}$$

$$x_i = f_i^{[3]} = \tanh\left(W^{[2]} f_i^{[2]} + b^{[2]}\right) \tag{5.2}$$

$W^{[1]} \in \mathbb{R}^{l \times r}, W^{[2]} \in \mathbb{R}^{m \times l}$, $b^{[1]} \in \mathbb{R}^l$ and $b^{[2]} \in \mathbb{R}^m$ (where l and m are the size of the input layers, and r the feature size).

The pyramidal encoder has two hyperparameters: the depth k (or number of layers), and the stride b. The output of the encoder is a sequence of annotations of length $\hat{T} = T/b^{k-1}$. In our experiments, we use three layers with a stride of two. This reduces the length of the input sequence by a factor of 4:

$$h_i^{[1]} = \text{update}^{[1]}(h_{i-1}^{[1]}, x_i) \tag{5.3}$$
$$\widetilde{h}_i^{[1]} = \frac{1}{2}(h_{2i-1}^{[1]} + h_{2i}^{[1]}) \tag{5.4}$$
$$h_i^{[2]} = \text{update}^{[2]}(h_{i-1}^{[1]}, \widetilde{h}_i^{[1]}) \tag{5.5}$$
$$\widetilde{h}_i^{[2]} = \frac{1}{2}(h_{2i-1}^{[2]} + h_{2i}^{[2]}) \tag{5.6}$$
$$h_i = \text{update}^{[3]}(h_{i-1}^{[3]}, \widetilde{h}_i^{[2]}) \tag{5.7}$$

The Equations 5.4 and 5.6 correspond to time pooling, and are responsible for the time length reduction of the sequences. This means that we average every pair of consecutive annotations produced by a given layer. In the case where the input sequence length is not a multiple of two, the last annotation is used on its own. It is also possible to just skip encoder states instead of averaging them.

For simplicity, the previous equations are shown for unidirectional LSTMs. In practice we stack three layers of bidirectional LSTMs of size n in each direction, where each new layer reads both the backward and the forward states from the previous layer:

$$\vec{h}_i^{[k]} = \text{update}_{fwd}^{[k]}(\vec{h}_{i-1}^{[k]}, \widetilde{h}_i^{[k-1]}) \tag{5.8}$$
$$\overleftarrow{h}_i^{[k]} = \text{update}_{bwd}^{[k]}(\overleftarrow{h}_{i+1}^{[k]}, \widetilde{h}_i^{[k-1]}) \tag{5.9}$$
$$h_i^{[k]} = \vec{h}_i^{[k]} \oplus \overleftarrow{h}_i^{[k]} \tag{5.10}$$

Note that we use the same input $\widetilde{h}_i^{[k-1]}$ for both the backward and the forward LSTM. This corresponds to a concatenation of forward and backward annotations from the previous bidirectional layer (averaged over two time steps). The final encoder annotations used by the decoder are $h_i = W_{bidir} h_i^{[3]}$, where $W_{bidir} \in \mathbb{R}^{n \times 2n}$.

Decoder We use a two-layer LSTM decoder with attention. We initialize the decoder states with $s_0^{[1]} = \tanh(W_{init}^{[1]} h_{-1})$ and $s_0^{[2]} = \tanh(W_{init}^{[2]} h_{-1})$, where $h_{-1} = \overleftarrow{h}_1^{[1]} \oplus \overleftarrow{h}_1^{[2]} \oplus \overleftarrow{h}_1^{[3]}$ is a concatenation of the last states of all backward layers.

The next states of the decoder are computed as follows:

$$s_t^{[1]} = \text{update}_{dec}^{[1]}(s_{t-1}^{[1]}, E'(\widetilde{z}_{t-1})) \tag{5.11}$$
$$s_t^{[2]} = \text{update}_{dec}^{[2]}(s_{t-1}^{[2]}, s_t^{[1]}) \tag{5.12}$$

During training, $\widetilde{z}_{t-1}$ is the previous symbol in the reference sequence (teacher forcing). During evaluation, $\widetilde{z}_{t-1}$ is the predicted symbol at the previous time step. $E' \in \mathbb{R}^{|V'| \times m'}$ is the target embedding matrix, where $|V'|$ is the target vocabulary size.

We compute an attention context $c_t \in \mathbb{R}^n$ (over the encoder annotations h_i), which is concatenated to the LSTM's output $s_t^{[2]}$, and mapped to the target vocabulary size:

$$y_t = W_{voc}(s_t^{[2]} \oplus c_t) + b_{voc} \qquad c_t = \text{look}(s_t^{[1]} \oplus s_t^{[2]}, (h_i)_{i=1}^{\hat{T}}) \tag{5.13}$$

where $W_{voc} \in \mathbb{R}^{|V'| \times (n+n')}$ and $b_{voc} \in \mathbb{R}^{|V'|}$, with n the encoder cell size, and n' the decoder cell size.

This vector $y_t \in \mathbb{R}^{|V'|}$ contains unnormalized scores for all words in the target vocabulary. The greedy decoder just picks the word at each time step with the highest score. For training and beam search decoding, a probability distribution is estimated over the entire vocabulary by normalizing with a $\mathrm{softmax}$ function:

$$p(\widetilde{z}_t = j | y_{t-1}, s_{t-1}, (h_i)_{i=1}^{\hat{T}}) = \mathrm{softmax}(y_{tj}) \tag{5.14}$$

Convolutional attention We use the same attention model as Chorowski et al. (2015), which uses a convolution filter to take into account the attention weights at the previous time step. This helps the model learn to do a monotonous alignment. On an Automatic Speech Recognition Task (ASR) like Chorowski et al. (2015), this is particularly relevant as alignments are always monotonous. In Speech Translation between two close languages like French and English, we expect the alignment to be close to monotonous. Providing this extra information to the attention mechanism might help the model in the early stages of training.

$$r_{ti} = v_{att}^\top \tanh(W_{att}(h_i \oplus s_t^{[1]} \oplus s_t^{[2]}) + b_{att} + f_{ti}\mu_{att}) \qquad f_t = F * \alpha_{t-1} \tag{5.15}$$

$$\alpha_{ti} = \mathrm{softmax}(r_{ti}) \tag{5.16}$$

$W_{att} \in \mathbb{R}^{k \times (n+2n')}$, $b_{att}, v_{att}, \mu_{att} \in \mathbb{R}^k$, and $F \in \mathbb{R}^{2a+1}$ are trained parameters of the attention model, where n is the encoder cell size, n' the decoder cell size and a is the size of the convolution filter. The convolution is computed as follows:

$$f_{tj} = \sum_{i=j-a}^{j+a} F_i \times \hat{\alpha}_{t-1,i} \tag{5.17}$$

$$\hat{\alpha}_{t-1,i} = \begin{cases} \alpha_{t-1,i} & \text{if } 1 \leq i \leq \hat{T} \\ 0 & \text{otherwise} \end{cases} \tag{5.18}$$

$\alpha_{t-1} \in \mathbb{R}^{\hat{T}}$ is the vector of attention weights from the previous time step. $\hat{\alpha}_{t-1}$ is the same vector padded with zeros, for the convolution to result in a sequence of the right length ($\hat{T}$).

5.1.2 Synthetic Corpus

As we did not have any good-sized speech translation parallel corpus (i.e., a set of triples with speech segment, transcription and translation in another language), we generated one with speech synthesis. We decided to use the French→English BTEC corpus, which is a small parallel corpus with less that 20k segments.

BTEC (Basic Travel Expression Corpus) is very clean and contains short sentences (10 words on average), with a small vocabulary (9218 unique tokens on the French side, 7186 on the English side). Because end-to-end speech translation is a hard task (and unexplored), we wanted to place ourselves in favorable (but still low-resource) conditions. The BTEC corpus seemed to be a very good candidate for this.

Corpus		Total		French (per segment)			English (per seg)	
		segments	hours	frames	chars	words	chars	words
BTEC	train	19972	15:51	276	50	10	42	9.5
	dev	1512	0:59	236	40	8.1	33	7.6
	test	933	0:36	236	41	8.2	34	7.7

TABLE 5.1: Size of the French-English BTEC corpus. The training corpus contains 189k English words and 201k French words. The character counts take whitespaces into account. The speech side is synthetic (obtained through TTS), and the frames are of 40 ms with a step size of 10 ms. The counts are for the *Agnes* speaker. The full corpus used for training the AST models is actually 6 times this size, as we use TTS for all 19 972 segments with 6 different speakers and concatenate all the data.

Table 5.1 shows corpus size information. The test set was initially separated into two test sets of size 469 and 464, which we eventually merged into a single test set for ease of experimentation. This corpus also has multiple references: 7 references per source sentence in the test set, and 16 references per source sentence in the dev set; which we use for BLEU evaluation. This is a very small corpus by MT standards. Even though the setting is somewhat artificial, we were interested to see how NMT methods (which are notoriously data-hungry) would manage on this dataset.

Synthetic speech was generated using *Voxygen*,[2] a commercial speech synthesis system, for 4 different female voices (Agnes, Marion, Helene, Fabienne) and 3 different male voices (Michel, Loic, Philippe). It is important to note that this is corpus-based concatenative speech synthesis (Schwarz 2007) and not parametric synthesis. Hence, for each speaker's voice, speech utterances are generated by concatenation of units mined from a large speech corpus (generally around 3000 sentences/speaker). This means that despite having very little intra-speaker variability in our speech data, there is a realistic level of inter-speaker variability. To challenge the robustness of our system to inter-speaker variability, we concatenate the training data for 6 speakers, and leave the last speaker (Agnes) for evaluation (on separate dev and test sets).

Pre-processing All the text data is tokenized with the *Moses* tokenizer and lowercased. In our experiments, we use tokenized uncased BLEU for evaluation (Word Error Rate for ASR).

Like Chan et al. (2016), as input for our speech model, we segment the speech raw data into frames of 40 ms, with a step-size of 10 ms, and extract 40 MFCCs for each frame, along with the frame energy. This results in vectors of size 41.

Other works in ASR or speech translation sometimes use more features (Bahdanau et al. 2016; Chorowski et al. 2015), by including the first order and second order derivatives, but our first tests did not exhibit any benefit from doing this. Because our training set is very small, we do not want to give to the model more information than absolutely necessary (to avoid overfitting). Furthermore, more features means larger models and longer training times, and much larger datasets (in terms of memory footprint), which is the main reason why we settled for 41 features.

We use the *Yaafe* library to extract these features (Mathieu et al. 2010). It is less maintained, but easier to delve into than the well-known *Kaldi*. It also works with Python, which is very useful for prototyping.

[2] www.voxygen.fr

The audio side of our corpus is encoded in a simple binary format that we designed. It is composed of a small header of two 32 bits signed integers (2×4 bytes), corresponding to the number of audio segments (N), and the dimension (r) of the features for these entries (usually 41). This header is then followed by all the segments. A segment is encoded as a single 32 bits signed integer corresponding to the number of audio frames T_k in this segment, followed by as many frames. A frame is an array of r 32 bits floating-point values ($r \times 4$ bytes). Thus, the total size in bytes is: $8 + 4 \times \sum_k^N (1 + r \times T_k)$.

No additional alignment file needs to be kept. Like usual in machine translation, the alignment information is implicitly encoded by the position of the entries. There are 933 lines in the English side of the test set. There are also 933 entries in the corresponding French audio feature file, and its k$^{\text{th}}$ entry is aligned with the k$^{\text{th}}$ line in the English text file.

5.1.3 Experiments

Model settings Our base model for speech translation has a pyramidal encoder of three bidirectional LSTM layers with $n = 256$ units in each direction. There are no source embeddings, as the encoder takes as input a pre-computed sequence of vectors (of 40 MFCCs + frame energy). We also use two fully connected layers of size $l = m = 256$ between the input features and the first layer of the encoder.

The decoder predicts sequences of words, and it has two LSTM layers of size $n' = 256$, with embeddings of size $m' = 256$.

As a baseline, we also train text translation models. We want to see how they manage against SMT, and whether it makes sense to use NMT on such a small and specialized corpus. NMT results also give us an oracle on cascaded neural AST performance, assuming perfect ASR.

The NMT model is identical to the AST model, except for its encoder. The encoder reads words (instead of audio frames), with an embedding size of $m = 256$, and two bidirectional LSTMs of size $n = 256$ in each direction. Also, we use a basic global attention model (Bahdanau et al. 2015) without convolutions.

Training settings For training, we use Adam with an initial learning rate of 0.001 (Kingma et al. 2015), and a mini-batch size of 64. We apply dropout during training on the inputs of each LSTM (Zaremba et al. 2014) with a rate of 0.5. We also apply dropout on the outputs of each input layer, and on the initial state $[s_0^{[1]}, s_0^{[2]}]$ of the decoder. Regularization is essential to avoid overfitting, especially considering the small size of the training corpus. All parameters (including embeddings) are initialized to TensorFlow's defaults: Xavier uniform initialization.

We train our models for 20k steps, which takes less than 2 hours for the *text* models, and roughly 8 hours for the *speech* models (on a single GTX 1070). We save a new checkpoint of the model every 2000 steps, and evaluate its performance on the dev set. At the end of training, we keep the checkpoint whose BLEU score on the dev set is the highest.

Results Table 5.2 shows the results of our Machine Translation experiments. The SMT baseline is trained with Moses on the BTEC training data (with default settings, and a trigram language model) and tuned using MERT on the dev set. We see that our best NMT system, an

Corpus	BLEU score				
	Greedy	Beam search	+LM	+Ensemble	SMT
dev	42.5	43.6	45.1	51.6	54.3
test	40.4	41.4	42.9	**47.5**	**47.6**
dev (16 refs)	53.4	55.0	57.5	65.3	66.2
test (7 refs)	48.9	50.4	52.2	**57.8**	**56.6**
train.1000	78.4	80.5	82.9	89.6	76.6

TABLE 5.2: Results of the Text Translation task on BTEC, under different settings of the decoder. "Greedy" uses a greedy decoder. "Beam search" uses a beam search decoder with a beam size of 8. "LM" adds an external trigram language model estimated on the target side of *BTEC train*. "Ensemble" uses a log-linear model with 5 NMT models trained independently (+ LM). The *train* corpus used in evaluation is a subset of 1000 sentences from *BTEC train*. For all the non-ensemble configurations, the best model out of 5 is used (according to its score on *dev*). The SMT baseline is a phrase-based model (Moses) trained on *BTEC train*, and tuned on *BTEC dev*.

Corpus	Speaker	BLEU score				
		Greedy	Beam search	+LM	+Ensemble	Baseline
dev	*Agnes*	30.1	32.3	33.5	40.2	43.7
test		29.1	31.3	31.9	**37.9**	**40.9**
dev (16 refs)	*Agnes*	38.3	40.9	43.2	51.2	56.0
	Michel	43.1	46.3	47.9	54.5	55.7
test (7 refs)	*Agnes*	35.6	37.9	39.3	**46.0**	**49.7**
	Michel	39.0	41.4	42.7	**48.6**	**49.2**
train.1000	Michel	53.8	60.8	61.8	82.6	60.5

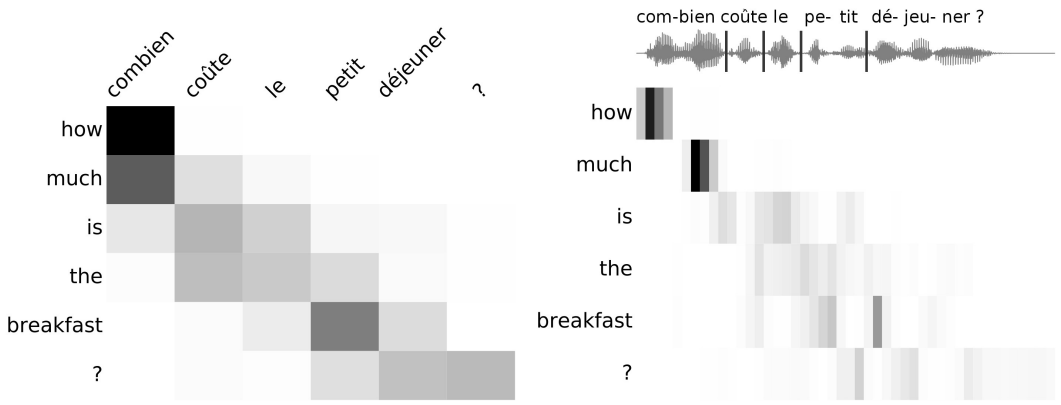
TABLE 5.3: Results of the speech translation experiments on BTEC. The models are trained with 6 different speakers (including *Michel*). *Agnes* is not used for training. The *Ensemble* configuration uses 5 models trained independently. The baseline system (last column) uses a pipeline Google Speech ASR + SMT system trained on BTEC. The WER scores obtained by the baseline ASR system range between 23% and 26%.

ensemble of 5 models with a language model (the same LM model as the SMT baseline),[3] gives similar results in terms of BLEU as the SMT system. This is rather surprising, considering the small training set, and observations by Koehn et al. (2017) that NMT is not very good in low-resource settings. Also, even though the models seem to overfit quite a lot (see "train.1000" scores), this does not seem to be a problem during evaluation.

Table 5.3 shows the results of our Speech Translation experiments. The baseline is a cascaded system, which uses Google Speech API for transcription, followed by an SMT system trained with Moses on BTEC and tuned on BTEC dev. The SMT system is slightly different than the MT baseline. Google Speech API outputs text without punctuation, except when actually pronounced. For this reason, we strip the source side of the BTEC train corpus from all punctuation and train an SMT system to output target-language text *with* punctuation.

Our best end-to-end AST results are obtained with an ensemble of 5 AST models and a trigram language model. All models (including the language model) are combined at decoding time (in the beam search decoder) by summing their log-probabilities at each time step. We see that the

[3]The language model's log probabilities are included into the log-linear model, like the 5 models in the ensemble.



(A) Text translation alignment (B) Speech translation alignment

TABLE 5.4: Alignments performed by the attention model on an example from the dev set. The alignments are forced, which means that we force the models to output the ground truth sequence (teacher forcing). The target of the model is on the left, while the input is at the top. Values on a given row sum to one (softmax normalization), and darker shades correspond to attention weights closer to 1 (white is 0).

BLEU scores for the known speaker *Michel* are equivalent to those of the baseline. The results on a new speaker (*Agnes*) are behind, but still encouraging, considering that we did not use any speaker adaptation technique.

Figure 5.4 shows alignments performed by the attention models of the text translation and speech translation models. These are forced alignments (using teacher forcing) on an example from the dev set. We see that the attention mechanism roughly aligns the output words with the corresponding part in the speech signal ("how much" aligns with "combien").

5.1.4 Improvements

Following this work on the synthetic BTEC corpus, we sought to improve our models on the MT and AST tasks. Weiss et al. (2017) propose a number of improvements for Automatic Speech Translation. In particular, they output sequences of characters, and not words. Their encoder is also different from the LAS pyramidal encoder: they use several layers of convolutions to reduce the length of the input sequence, and then a non-pyramidal multi-layer LSTM. They achieve very good results (better than a cascaded baseline) on a real-world corpus of recorded telephone conversations (Fisher-CALLHOME). We apply some of their architecture changes to the synthetic BTEC corpus.

Character-based AST We propose two character-based models for AST, one with a pyramidal encoder, and one with a convolutional encoder.

The encoder of the first model is similar to the model presented earlier (see Section 5.1.1): two input layers of size $l = m = 256$, followed by a pyramidal encoder with three layers of bidirectional LSTMs of size $n = 256$ and average time pooling with a stride of two.

The main difference to the previous model is with the decoder. It has two layers of LSTMs of size $n' = 256$. It outputs characters, with a vocabulary of size 43 and a maximum sequence

length of 120 characters (including whitespaces). We use character embeddings of size $m' = 64$.[4] Unlike the previous model, we adopt a "generate first" strategy, where the next token $\widetilde{z}_t$ is generated before updating the decoder's state. Also, the current attention context vector is used for updating the decoder's state (in addition to the previous state and the previous symbol):

$$c_t = \text{look}(s^{[2]}_{t-1}, (h_i)_{i=1}^{\hat{T}}) \tag{5.19}$$

$$y_t = W_{voc}(s^{[2]}_{t-1} \oplus c_t) + b_{voc} \tag{5.20}$$

$$s^{[1]}_t = \text{update}^{[1]}_{dec}(s^{[1]}_{t-1}, E'(\widetilde{z}_t) \oplus c_t) \tag{5.21}$$

$$s^{[2]}_t = \text{update}^{[2]}_{dec}(s^{[2]}_{t-1}, s^{[1]}_t) \tag{5.22}$$

To limit overfitting, we use dropout with a rate of 0.4. Dropout is applied on the inputs of the LSTMs, on the decoder's initial state (just before the dense layer), on the outputs of the encoder's input layers, and in the attention model (on the encoder's hidden states and the decoder state).

We use Adam for training, with default settings and a batch size of 64. We train for 100k steps, with a BLEU evaluation on the dev set every 1000 steps, and keep the best performing checkpoint for final evaluation. All the model parameters are initialized to TensorFlow's defaults, i.e., a Xavier initialization scheme (except for the bias vectors, which are initialized to zero). This model is called "Pyramidal" in Table 5.5.

Convolutional Encoder for AST We train a second model that uses a different encoder, borrowing some ideas from Weiss et al. (2017). Following the two input layers, we use a stack of two convolutional layers. These convolutions use a time-wise stride of two, which divides the length of the sequence by two after each layer. Like with the pyramidal encoder, the input sequence is reduced to 1/4$^{\text{th}}$ its initial length. Following these layers, we stack three standard (non-pyramidal) bidirectional LSTMs of size $n = 256$.

More precisely, the shape of the input features is $T \times r$, where $r = 41$. The two input layers are of size $l = 256$ and $m = 128$ (with a bias vector, and a $\tanh$ activation), which results in features of shape $T \times m$. Then, we use a first convolutional layer, with 16 filters of shape 3×3, with zero-padding and a stride of 2 w.r.t. both dimensions. This results in a tensor of shape $\frac{T}{2} \times \frac{m}{2} \times 16$. Then a second convolutional layer, with 16 filters of shape $3 \times 3 \times 16$ results in a new tensor of shape $\frac{T}{4} \times \frac{m}{4} \times 16$. We flatten this tensor to a $\frac{T}{4} \times 4m = \hat{T} \times \hat{m}$ tensor, which is then passed to a stack of three bidirectional LSTMs of size $n = 256$.

Note that our encoder is conceptually much simpler than (Weiss et al. 2017). Our convolutions are linear, we do not use a convolutional LSTM, and there are no dense layers between the LSTM layers. The decoder and training scheme are identical to the pyramidal model. We call this model "Convolutional" in Table 5.5. We also train a variant of this model that uses a single-layer conditional LSTM for the decoder (Sennrich et al. 2017), instead of a two-layer LSTM (model "Convolutional-Cond").

Character-based NMT For comparison, we also train a word to character model on the BTEC text translation task. The encoder reads words, with word embeddings of size $m = 128$ and a

[4] It makes sense to use smaller embeddings, as characters encode much less information than entire words.

bidirectional LSTM of size $n = 256$ in each direction. Like the two AST models, the encoder's backward and forward states are concatenated and mapped to vectors of size 256 (with a linear projection), which are then read by the attention mechanism. The initial states of the encoder are also set to zero. Contrary to the previous models, we average all the hidden states of the encoder (w.r.t. time axis) to initialize the decoder.

We use the same cond-LSTM decoder as with the "Convolutional-Cond" variant of our AST model, with the same character-level vocabulary (43 symbols), maximum sequence length (120) and embedding size ($m' = 64$). We call this model "LSTM-Char".

We use dropout with a rate of 0.2 on the source embeddings, the attention model, the inputs and the outputs of the LSTMs, and the initial state of the decoder. We also drop target characters at random during training with a probability of 0.2.[5]

Non-recurrent encoder for NMT We also train a word-based model with a radically different architecture. Instead of using recurrent neural networks in its encoder, we use convolutions. The idea is that we do not need to build a representation of the entire input sequence, when we have an attention mechanism that can look anywhere in the input sequence. This is inspired by the Transformer model (Vaswani et al. 2017), which does not use any recurrent neural network, but many dense layers with attention mechanisms that can look anywhere in the previous layer.

The encoder uses word embeddings of size 256. Because we do not encode the sentence with an LSTM, we want to give more representation power to the embeddings. We also use position embeddings of size 64, which encode the absolute position of each symbol in the input sequence. This is important, as contrary to RNNs, which read the sequence from left to right and are able to encode positional information, simple convolutions do not have this capacity (except at a local level). The position embedding matrix has size 25×64, as the maximum sequence length is 25.

The sequence of word representations (each of size $m = 256 + 64 = 320$) is processed by two convolutional layers. Similarly to the speech encoder, we use 16 convolution filters of size 3×3. We use a stride of 2 w.r.t. feature axis, but no stride for the time axis (we do not want to reduce the length of the sequence). This is followed by another linear convolution layer with 16 filters of size $3 \times 3 \times 16$. This results in a sequence of shape $T \times 1280$, which is directly read by the decoder's attention model (without being processed by a RNN). These vectors are also averaged w.r.t. time axis and used to initialize the decoder.

The decoder is a conditional LSTM of size $n' = 256$, with word embeddings of size $m' = 128$, and a dense output layer of size $d = 128$ (before the linear vocabulary projection). The word-based models tend to overfit more than character-based ones, so we use dropout with rate 0.4 on the word embeddings, the inputs of the RNNs, the initial state of the decoder, the decoder's dense layer, and in the attention mechanism. We train the model with Adam and a batch size of 32. All the model's weights (except bias vectors) are initialized to a centered normal distribution with standard deviation 0.1.

The results of this model on the BTEC MT task are shown in Table 5.5 (model "Conv-Word").

Results Table 5.5 shows the BLEU scores of our models on the BTEC MT task, and on the AST task.

[5] This is implemented by zeroing out the entire embedding vector for those characters. The "dropped' characters are still used in the computation of the loss. This is similar to dropout on the embeddings, but more aggressive.

Task	Model	Dev BLEU		Test BLEU		Steps
		Greedy	Beam	Greedy	Beam	
AST	Pyramidal	34.8	38.3	33.9	**36.8**	56k (×64)
	Convolutional	35.0	38.3	33.4	35.7	53k (×64)
	Convolutional-Cond	34.1	36.8	32.8	35.0	80k (×64)
MT	Conv-Word	53.2	53.7	47.8	48.8	27k (×32)
	LSTM-Char	51.9	53.2	47.4	49.2	64k (×64)
	LSTM-Char ensemble (5)	54.8	56.0	49.7	**51.0**	–

TABLE 5.5: Results of our best performing models on the BTEC Automatic Speech Translation task (AST), and on the Machine Translation task (MT). The dev and test data for is on the unknown speaker *Agnes*. The ensemble combines 5 instances of the same model (trained from scratch). All the AST models output characters. The beam search decoder uses a beam size of 8 with length normalization. The numbers in parentheses in the last column are the batch sizes. Our previous best test BLEU scores on end-to-end AST were 37.9 (ensemble) and 31.9 (single) (see Table 5.3). The best NMT score was 47.5 (ensemble) (see Table 5.2).

We see that our new best single model on AST outperforms the previous best single model by 5 BLEU points, falling just 1 BLEU point off our previous ensemble result.

On NMT, the improvement is even more impressive, our best single model outperforms the previous ensemble result by almost 2 BLEU points. An ensemble of 5 models now obtains a BLEU score of 51, which is 3.5 points above the previous best NMT result (ensemble of 5), and 3.4 points above the SMT baseline.

These large improvements are mostly due to the following changes: character-level decoding, "generate first" strategy, and a more powerful decoder. The character-level models tend to overfit much less, and we hand-tuned (on the dev set) the dropout level and the size of the models to limit overfitting while keeping the models as large as possible. This way, we were able to train the models for much longer without overfitting. We also removed some unnecessary elements that obfuscated the model with no obvious improvements: the convolutional attention model and the external language model.

Interestingly, the atypical "Conv-Word" model for NMT achieves very good scores, for a fraction of the training time of the character-based models. The sentences from BTEC often have a very simple grammatical structure, which the attention mechanism (combined with the decoder's language model) can manage on its own. However, it is unsure whether this result would transfer to other, more complex, MT tasks (unless using multiple non-linear layers with multiple attention heads like the Transformer).

5.2 Extraction of a New AST Corpus

The results presented in the previous section were obtained on a synthetic corpus. Even though the results are promising, they are not sufficient to validate the efficacy of our end-to-end models. The text corpus (BTEC) contains extremely short sentences, with a very limited vocabulary compared to real-world corpora.

The main limit of this study is that the audio side of the corpus was obtained with speech synthesis and with a small number of speakers. With such a limited variability, it is easy for a large neural network to learn by heart a mapping from audio frames to words or characters.

Our evaluation on an unknown speaker mitigates these doubts. We see that our model is able to accurately translate speech from an unknown speaker, which means that even if it learned how to read audio frames by heart, it is still able to generalize to unseen audio frames.

To continue this study and apply our speech translation techniques to a real-world scenario, we looked for non-synthetic speech-to-translation datasets. These resources are very scarce, and those that exist are quite small. There is the Fisher-CALLHOME corpus of recorded telephone conversations (Spanish→English) (Post et al. 2013), which Weiss et al. (2017) use in their follow-up work. But a part of this corpus is owned by the LDC and not available to us.

There exist many audio transcriptions in the public domain and also text translations, sometimes of the same content. One notable example is audiobooks. The free resource "LibriSpeech" contains a thousand hours of audiobooks that are in the public domain, aligned with the original written books. On top of that, there is "Project Gutenberg" which offers free access to public domain books. Many such books have been translated in several languages.

Kocabiyikoglu et al. (2018) proceeded to find the intersection of these two resources, so as to produce a sentence-aligned corpus of speech, text transcriptions and text translations, from English to French. This section presents this work, as well as the final extracted corpus.[6][7]

Another resource, which is left for future work, is the TED talks. Cettolo et al. (2012) provide a corpus of TED talk subtitles aligned with their translations in other languages, which is part of the yearly IWSLT evaluation campaign.[8] The TED talks themselves (audio and video) are also available online.[9] Rousseau et al. (2014) processed this resource and created a corpus for ASR.[10] These two corpora could be aligned so as to produce a speech translation corpus. Furthermore, it could be interesting to use the video as additional input and do multimodal machine translation. A well-performing machine translation of subtitles along with audio and video could prove a useful real-world application (e.g., to translate upcoming TED talks).

5.2.1 Alignment

LibriSpeech Augmented LibriSpeech (Kocabiyikoglu et al. 2018) is the intersection of LibriSpeech (Panayotov et al. 2015) with several public-domain e-book repositories (including Project Gutenberg).

LibriSpeech is a resource for ASR that was built from LibriVox,[11] a free database of public domain audiobooks, spoken by volunteers through crowd-sourcing (originally not intended for ASR). The original English-language books that are used for the recordings come from Project Gutenberg.[12]

[6] This work was performed while the main author was doing his Master's thesis in GETALP (LIG), under the supervision of Laurent Besacier. I had no part in this work: I only took part in some of the discussions and helped disseminate the corpus. The end goal was to use this dataset to continue the experiments done in (Bérard et al. 2016a). This work resulted in the creation of a resource that was published at LREC 2018 (Kocabiyikoglu et al. 2018), and a follow up work of (Bérard et al. 2016a) that was published at ICASSP 2018 (Bérard et al. 2018).

[7] Augmented LibriSpeech: https://persyval-platform.univ-grenoble-alpes.fr/DS91/detaildataset

[8] https://wit3.fbk.eu/ and http://opus.nlpl.eu/TED2013.php

[9] https://www.ted.com/talks

[10] http://www-lium.univ-lemans.fr/en/content/ted-lium-corpus

[11] https://librivox.org/

[12] http://www.gutenberg.org/

corpus	hours	per-speaker minutes	female speakers	male speakers	total speakers
dev-clean	5.4	8	20	20	40
test-clean	5.4	8	20	20	40
dev-other	5.3	10	16	17	33
test-other	5.1	10	17	16	33
train-clean-100	100.6	25	125	126	251
train-clean-360	363.6	25	439	482	921
train-other-500	496.7	30	564	602	1166

TABLE 5.6: Description of each subset of the LibriSpeech corpus. This table is extracted from (Panayotov et al. 2015). The *train-clean* subset is split in two so as to let the user choose the corpus size most suited to his needs (460 hours is a lot to deal with in many applications).

The creators of LibriSpeech carefully segmented the speech signal and aligned it with the English-language books (at the chapter-level and sentence-level). English sentences were lower-cased, stripped of special symbols and punctuation, and labeled as "transcriptions". The original (unprocessed) books are also available as part of LibriSpeech.

LibriSpeech is composed of 1568 books, with a total of 5831 chapters, spoken by over two thousand different speakers. One of the goals of this resource was to have as many speakers as possible. Speakers were limited to a maximum of 25 minutes of speech, to avoid major imbalances.

Panayotov et al. (2015) trained an ASR model on Wall Street Journal (WSJ), and computed the average Word Error Rate of this model on each of the speakers. LibriSpeech was split in two: a "clean" part that gathers the best-scored speakers, and an "other" part that gathers the noisier data (either lower quality recordings, or more challenging transcriptions). Then, these two parts were split into disjoint train, dev and test sets. Table 5.6 shows detailed statistics about the different subsets that are the result of this split.

Finding French Translations The first step to build the Augmented LibriSpeech corpus was to find French-language books corresponding to the English-language books used in LibriSpeech. First, the English book titles were automatically translated to French using DBPedia (a public knowledge base that contained the "official" translation of most titles). Then, an index of French-language public domain e-books was used to find web links matching these titles.[13] This, along with manual search on several public domain e-book websites, resulted in a total of 315 different books in French, corresponding to 1818 chapters in LibriSpeech. These books are mostly novels, but some are plays, poems, fables, treaties or religious texts. Then, the English and French books were split into matching chapters (regular expressions were used to identify chapter boundaries). This resulted in a set of 1423 pairs of chapters, from 247 books.

Sentence Alignment Then, these chapters were aligned at the sentence level using a sentence aligner called *hunalign* (Varga et al. 2005). This program takes an unaligned parallel corpus (i.e., a pair of documents that are translations of each other), segmented at the sentence-level, and uses a bilingual dictionary (Moore 2002) along with sentence-length information (Gale et al. 1993) to find an alignment between source and target sentences. This results in a set of pairs of sentences, more commonly referred to as "parallel corpus".

[13] http://noslivres.net/

Kocabiyikoglu et al. (2018) built a large bilingual dictionary of 128 000 entries, by merging several open source dictionaries. They also pre-processed the French and English text by: 1) removing clutter and normalizing symbols with regular expressions, 2) splitting into sentences using NLTK, 3) stemming (removing suffixes) to reduce vocabulary sparsity and facilitate dictionary matches by hunalign. With this pre-processing done, they used hunalign with the extracted dictionary to find an alignment. Once the alignment found, they reverted the sentences to their unstemmed form.

Speech Alignment The text alignment was done between the original English books used in LibriSpeech and the French-language translation (or original version) of these books. However, the speech segments provided in LibriSpeech correspond to English transcriptions that do not necessarily match the segmentation done by NLTK and hunalign.

First, Kocabiyikoglu et al. (2018) aligned the English-language transcriptions with the English side of the parallel corpus, using *mweralign*, a program for realigning texts in the same language that have a different segmentation. This alignment resulted in a new segmentation of the English transcriptions, which are correctly aligned with the French translations.

However, this new segmentation is unaligned with the actual speech segments from LibriSpeech (i.e., a single transcription does not correspond to a single speech segment anymore). For this reason, all the speech segments of a chapter were concatenated into a single file, and aligned (at the frame-level) with the transcriptions using the *gentle* toolkit.[14] Once this alignment done, it is straightforward to segment this single file into speech segments that are aligned with the new segmentation of the transcriptions.

At the end of this process, there are a little over 131k quadruplets of speech segments, with their transcription (clean), raw English text (untokenized and unnormalized), and raw French translation. This corresponds to a total of 236 hours of speech, from 1408 chapters, belonging to 247 different books. In each of these tuples, Kocabiyikoglu et al. (2018) also add an automatic translation of the English text, using Google Translate.

Table 5.7 shows some examples of English transcriptions along with their French translations, randomly sampled from this corpus.

Evaluation Kocabiyikoglu et al. (2018) manually evaluated the quality of the corpus by sampling 200 sentences from 4 different chapters: 2 chapters whose average alignment score (by hunalign) was close to the global average, 1 chapter with higher than average score, and 1 chapter with a worse than average score. They asked three annotators to rate the speech alignments on a 1 to 3 scale, and the text alignments on a 1 to 5 scale. The average speech alignment score is very close to three (2.89), which validates the quality of the audio alignments performed with *gentle*. The per-chapter text alignment ratings correlate with their average hunalign scores.[15] This suggests that the alignment scores produced by hunalign are a good way of assessing the quality of the alignments (and possibly filtering the alignments according to their quality).

[14] https://github.com/lowerquality/gentle

[15] The average human ratings are 4.64, 4.28, 3.86 and 3.58, and the hunalign scores are 1.34, 1.14, 0.96 and 0.66.

English	**French**	**Google Translate**
they returned to the hotel at the door franz ordered the coachman to be ready at eight	à la porte , franz donna l' ordre au cocher de se tenir prêt à huit heures .	ils sont retournés à l' hôtel ; à la porte , franz a ordonné au cocher d' être prêt à huit heures .
the ideal of oppression was realized by this sinister household	l' idéal de l' oppression était réalisé par cette domesticité sinistre .	l' idéal de l' oppression a été réalisé par ce foyer sinistre .
were simply running around blindfolded ned land was just pronouncing these last words when we were suddenly plunged into darkness utter darkness	nous marchons , nous naviguons en aveugles ... " -- ned land prononçait ces derniers mots , quand l' obscurité se fit subitement , mais une obscurité absolue .	nous courons simplement autour des yeux : " ned land ne faisait que prononcer ces derniers mots quand nous étions tout à coup plongés dans l' obscurité , l' obscurité totale .

TABLE 5.7: Examples of sentence tuples in Augmented LibriSpeech. Left: English transcription. Middle: automatically aligned French translation (extracted from a book). The transcriptions do not contain any punctuation symbols (as is usual in ASR). Right: automatic translation of the English (raw) text by Google Translate.

5.2.2 Final Corpus

The main motivation for building this corpus is to use it as a public benchmark for End-to-End Automatic Speech Translation. To encourage the reproducibility of research and facilitate the comparison of results, Augmented LibriSpeech was split into "official" train, dev and test sets.

This corpus is particularly challenging, as the segments are quite long on average, the vocabulary is large, and the language can be very specialized, formal, and sometimes archaic. The alignment process is noisy, and sometimes results in inaccurate alignments, which makes the corpus even more difficult. For this reason, the tuples were sorted according to their "alignment quality", a combination of the alignment score by hunalign and a score obtained with a crosslingual similarity measure from (Ferrero et al. 2016). Augmented LibriSpeech was split into two subsets according to these scores: a clean part and a noisier part (not unlike the data split done in LibriSpeech).

One hundred hours of speech were used to build a "clean" training set. Another 2 hours and a little less under 4 hours of clean speech were used to build a development set and a test set. The remaining (noisier) 123 hours were put in a more challenging set, yet still potentially useful, named "other".

The chapters inside the dev and test sets were carefully selected to respect the following rules:

- The chapters should be unique (absent from the other subsets).
- The speakers should be unknown from the training set (this limitation in not true for "other" however).
- The books should have several other chapters inside the training set (to make the task less challenging).

corpus	segments	words	hours	avg len (s)	spkrs	books	chapters	avg score
train	47271	961k	100:00	7.62	728	240	1232	1.44
dev	1071	18.7k	2:00	6.73	12	9	17	1.40
test	2650	36.3k	3:44	6.57	16	10	22	1.39
other	61369	1.21M	122:46	7.20	648	232	1091	0.96

TABLE 5.8: Information about each subset of the Augmented LibriSpeech corpus. The train, dev and test corpora are cleaner, with higher quality alignments on average. The score in the last column is the alignment score produced by hunalign (averaged over the entire subset). The "avg len" column gives the average time length in seconds of speech segments.

- The chapters should belong to the "clean" part of the original LibriSpeech (i.e., with a good speech quality and not too challenging English transcription).

Finally, the test set was manually inspected to remove all inaccurate alignments. Table 5.8 shows detailed statistics about the final subsets.[16]

5.3 Speech Translation of Audiobooks

In this section, we apply our speech translation models to the audiobook corpus. As this task is much harder, we propose new techniques, which we also evaluate on the BTEC corpus as a measure of comparison to the previous results. We also train MT and ASR models. This work was published at ICASSP,[17] as a paper titled "End-to-End Automatic Speech Translation of Audiobooks" (Bérard et al. 2018).

5.3.1 Data and Pre-Processing

Like in Section 5.1.2, audio files were pre-processed using Yaafe (Mathieu et al. 2010), to extract 40 MFCC features and frame energy for each frame, with a step size of 10 ms and window size of 40 ms. We tokenize and lowercase all the text, and normalize the punctuation, with the Moses scripts.[18] For BTEC, the same pre-processing as in the previous section is applied. Character-level vocabularies for LibriSpeech are of size 46 for English (transcription) and 167 for French (translation). The decoder outputs are always at the character-level (for AST, MT and ASR). For the MT task, we translate from English transcriptions to French translations. The transcriptions are pre-processed into BPE units (Sennrich et al. 2016a). We limit the number of merge operations to 30k, which gives a vocabulary of size 27k. The MT encoder for BTEC takes entire words as input.

Table 5.9 gives detailed information about the size of each corpus. We perform all our experiments using "train" only (without "other"). We double the size of the training set by concatenating the French data with the Google Translate data. The source side (speech for AST, transcriptions for MT) is simply duplicated. For ASR, we keep the training set as it is.

[16] More information can be found about the GitHub page of the corpus: https://github.com/alicank/Translation-Augmented-LibriSpeech-Corpus

[17] International Conference on Acoustics, Speech, and Signal Processing, Calgary (Canada).

[18] http://www.statmt.org/moses/

Corpus		Total		Source (per segment)			Target (per seg)	
		segments	hours	frames	chars	words	chars	words
Augmented LibriSpeech	train 1	47271	100:00	762	111	20.3	143	28.2
	train 2						126	24.6
	dev	1071	2:00	673	93	17.4	110	22.0
	test	2048	3:44	657	95	17.7	112	22.5
BTEC	train	19972	15:51	276	50	10	42	9.5
	dev	1512	0:59	236	40	8.1	33	7.6
	test	933	0:36	236	41	8.2	34	7.7

TABLE 5.9: Size of the Augmented LibriSpeech and BTEC corpora, with the average frame, character and word counts per segment. Whitespaces are also counted as characters. The source side of BTEC actually has six times this number of segments and hours, because we concatenate multiple speakers (synthetic voices). LibriSpeech "train 1" (alignments) and "train 2" (automatic translation) share the same source side.

5.3.2 End-to-End Models

The Automatic Speech Translation model on Augmented LibriSpeech is almost identical to the "Convolutional-Cond" model presented in Section 5.1.4 ("Improvements"). The main differences are its size (larger cell size and embeddings), the structure of the decoder (which has a dense output layer, and uses the previous symbol for predicting the next symbol), and the training scheme (smaller batch size, smaller dropout, larger sequence length).

Speech Encoder We use the same convolutional speech encoder as in the previous section. The model takes a sequence of MFCCs of length T, which is passed to a stack of two non-linear layers (the same transformation is applied to all vectors in the sequence), resulting in a sequence of T vectors of size m. This sequence of vectors is then processed by two convolutional layers, with 16 filters of size 3×3 and a stride of two w.r.t. time axis and feature axis. This results in a new sequence of shape $\frac{T}{4} \times 4m = \hat{T} \times \hat{m}$. Then, this shorter sequence of features is used as input to a stack of three bidirectional LSTMs, which results in a sequence of annotations $h_1, \cdots, h_{\hat{T}}$, where each annotation h_i is a concatenation of a forward and a backward state: $h_i = (\vec{h}_i^{[3]} \oplus \overleftarrow{h}_i^{[3]}) \in \mathbb{R}^{2n}$, with n the encoder cell size.

Contrary to the AST models on BTEC, these concatenated states are used by the attention mechanism as they are (they are not mapped to size n). We average all these states into a single vector, which is used to initialize the decoder.

Character-level decoder We use a character-level decoder composed of a conditional LSTM (Sennrich et al. 2017), followed by a dense layer of size d.

$$s_t^{[1]} = \text{update}_{dec}^{[1]}(s_{t-1}^{[2]}, E'(\widetilde{z}_{t-1})) \tag{5.23}$$

$$c_t = \text{look}(s_t^{[1]}, (h_i)_{i=1}^{\hat{T}}) \tag{5.24}$$

$$s_t^{[2]} = \text{update}_{dec}^{[2]}(s_t^{[1]}, c_t) \tag{5.25}$$

$$x'_t = s_t^{[2]} \oplus c_t \oplus E'(\widetilde{z}_{t-1}) \tag{5.26}$$

$$y_t = W_{voc} \tanh(W_{out} x'_t + b_{out}) + b_{voc} \tag{5.27}$$

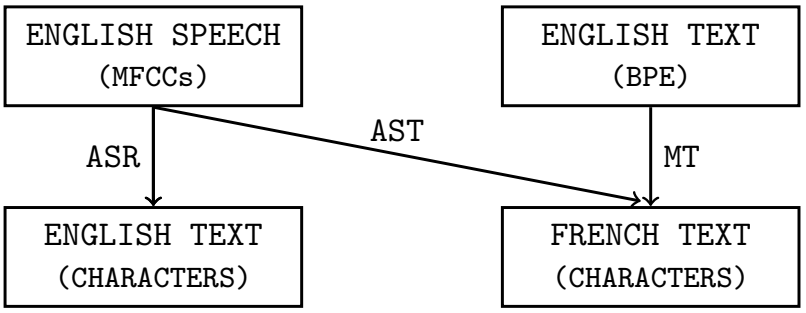


FIGURE 5.2: Multi-Task training of English-to-French AST, English-to-French MT and English ASR. The same speech encoder can be shared between AST and ASR, and the same character-level decoder can be shared between MT and AST. This can be exploited by pre-training the AST encoder and decoder on MT and ASR tasks, or multi-task training on the three tasks at once.

where $\mathrm{update}^{[1]}$ and $\mathrm{update}^{[2]}$ are two LSTMs with cell size n'. look is a vanilla global attention mechanism (Bahdanau et al. 2015), which uses a feed-forward network with one hidden layer of size k. $E' \in \mathbb{R}^{|V'| \times m'}$ is the target embedding matrix, with m' the embedding size and $|V'|$ the vocabulary size, $W_{voc} \in \mathbb{R}^{|V'| \times d}$, $b_{voc} \in \mathbb{R}^{|V'|}$, $W_{out} \in \mathbb{R}^{d \times (n' + 2n + m')}$, $b_{out} \in \mathbb{R}^{d}$. As always, $\widetilde{z}_{t-1}$ is either the previous ground truth symbol (teacher forcing), or the prediction of the model at the previous time step (when decoding).

Multi-task training As illustrated by Figure 5.2, the same speech encoder (acoustic model) can be used both for AST and ASR, and the same character-level decoder (language model) can be used for AST and MT.

We train ASR and MT models whose architecture is compatible with our AST model. Thanks to these two models, we have a baseline cascaded speech translation model, where the MT model is used to translate the outputs of the ASR model.

We also train a separate AST model that we initialize with these two pre-trained models: the encoder parameters are initialized with the parameters of the ASR model's encoder, while the decoder parameters are initialized with the MT model's. We call this model "Pre-trained AST". The only parameters that are not pre-trained (therefore initialized at random) are those that link the encoder with the decoder: the attention model and the transformation from encoder's final state to decoder's initial state.

We also train a Multi-Task model, which combines all three models (with shared encoders and decoders) and trains on the three tasks at once. Finally, we do a combination of both settings, where a multi-task model is initialized with pre-trained MT and ASR models ("Pre-trained Multi-Task").

Each task has its own training loss and its own training set and dev set. At each training step, we pick a task at random, with probability 0.6 for AST (the main task), and 0.2 for ASR and MT (auxiliary tasks). Then, we read the next batch from this task's training set, and do a mini-batch SGD update on this task's loss. This is a similar multi-task setting as Luong et al. (2016). Every 500 steps on each task, we evaluate this task's performance on its dev set, using BLEU for MT and AST, and WER for ASR. We keep the best checkpoint according to AST performance.

Model	Total Time	Total steps	Best dev loss
ASR	320h	500k	18.0 (0.28 BPC)
MT	44h	80k	68.6 (0.90 BPC)
End-to-End AST	356h	378k	78.0 (1.02 BPC)
Pre-trained AST	135h	140k	74.6 (0.98 BPC)
Multi-Task AST	382h	232k	75.4 (0.99 BPC)
Pre + Multi-Task AST	152h	101k	74.0 (0.97 BPC)

TABLE 5.10: Training time of our models on Augmented LibriSpeech. The "Pre + Multi-Task" model was trained on a Quadro P6000 (with 24 GB of memory). The MT model was trained on a GTX 1070 (8 GB), and all the other models were trained on a GTX 1080 Ti (12 GB).

We had to find matching AST, MT and ASR architectures, which explains why our single-task models do not always have the best possible performance. We performed most of our hyperparameter hand-tuning on the BTEC AST and MT tasks (see improvements section), while trying to maximize the scores on both tasks. Then we ported our models to Augmented LibriSpeech, by essentially increasing their size.

5.3.3 Experiments

Model Settings Our BTEC models use an LSTM size of $n = n' = 256$, while the LibriSpeech models use a cell size of $n = n' = 512$, except for the speech encoder layers whose cell size of $n' = 256$. We use character embeddings of size $m' = 64$ for BTEC, and $m' = 128$ for LibriSpeech. The MT encoders are shallower, with a single bidirectional layer. The source embedding size for words (BTEC) and subwords (LibriSpeech) is respectively $m = 128$ and $m = 256$.

The input layers in the speech encoders have a size of $l = 256$ for the first layer and $m = 128$ for the second. The LibriSpeech French decoder (MT and AST) has an output layer size of $d = 512$, and the English decoder (ASR) uses $d = 256$. For BTEC, we do not use any non-linear output layer, as we found that this led to overfitting.

Training settings We train our models with Adam (Kingma et al. 2015), with a learning rate of 0.001, and a mini-batch size of 64 for BTEC and LibriSpeech MT, and 32 for LibriSpeech AST and ASR (because of memory constraints). We use dropout with a rate of 0.2 for LibriSpeech and 0.4 for BTEC. Dropout is applied on the inputs of the LSTMs, on the initial state of the decoder, in the attention model, and on the outputs of the encoder's two input layers (for ASR and AST). In the MT tasks, we also apply dropout on the source embeddings with rate 0.2, and drop target symbols at random with probability 0.2.

Because of GPU memory limits, we set the maximum length to 1400 frames for LibriSpeech audio input (600 for BTEC), and 300 characters for its output (120 for BTEC). This covers about 90% of the training corpus. Longer sequences are kept but truncated to the maximum size. We evaluate our models on the dev set every 1000 mini-batch updates (500 for Multi-Task training) using BLEU for AST and MT, and WER for ASR, and keep the best performing checkpoint for final evaluation on the test set. Table 5.10 gives the total training time of our LibriSpeech models.

Results Table 5.11 presents the results for the ASR and MT tasks on BTEC and Augmented LibriSpeech. The MT task (and by extension the AST task) on Augmented LibriSpeech (translating books) looks particularly challenging, as we observe BLEU scores around 20%.[19]

	Model	ASR (WER ↓)		MT (BLEU ↑)	
		Dev	Test	Dev	Test
BTEC	greedy	16.2	14.9	51.9	47.4
	beam search	15.9	13.8	53.2	49.2
	ensemble (2)	13.1	**11.3**	54.7	**50.7**
LibriSpeech	greedy	21.0	19.9	21.2	19.2
	beam search	18.8	17.9	21.2	18.8
	ensemble (2)	14.8	**14.2**	21.8	19.3
	Google Translate			24.4	**22.2**

TABLE 5.11: MT and ASR results for BTEC and Augmented LibriSpeech. We use a beam size of 8 and ensembles of 2 models trained from scratch. The dev and test data for BTEC is on the unknown speaker *Agnes*.

We also evaluated our LibriSpeech ASR model on the official LibriSpeech "test-clean" and "test-other" datasets, for comparison with other results in the literature. The ensemble of two models obtains a WER of 18.5 on test-clean and 38.4 on test-other. There is a huge gap with the baseline WER scores of 6.6 and 22.5 that Panayotov et al. (2015) obtain with a DNN model trained on 100h of LibriSpeech. The main reason for this staggering gap in performance, is that we use a completely different architecture and training objective, which are not optimized for ASR but for MT. For instance, there is no monotonicity constraint or prior, which is a strong disadvantage against classic ASR models.

Tables 5.12 and 5.13 present the results for the AST task on LibriSpeech and BTEC. Contrary to Weiss et al. (2017), in both BTEC and LibriSpeech settings, best AST performance is observed when a symbolic sequence of symbols in the source language is used as an intermediary representation during the speech translation process (cascaded system). Pre-training and multi-task learning[20] improve AST performance, at a fraction of the training time. However, the training

[19] Google Translate is also scored as a topline (22.2%).
[20] If source transcriptions are available at training time.

Model	Dev BLEU		Test BLEU			Steps	Params (million)
	Greedy	Beam	Greedy	Beam	Ensemble		
Cascaded	15.2	15.5	14.6	14.6	**15.9***		6.3 + 15.9
End-to-End	12.2	13.0	12.3	12.9	**15.8**†	369k	9.4
Pre-trained	13.1	14.1	12.6	13.3		129k	
Multi-task	13.3	14.3	12.9	13.6		206k	
Pre + Multi-task	13.1	14.0	12.8	13.4		95k	

TABLE 5.12: AST results on Augmented LibriSpeech. $\star$ chains an ensemble of two ASR models with an ensemble of two NMT models (see Table 5.11). The non-cascaded ensemble‡ combines all four models. When doing an ensemble with the best 2 models, we get a BLEU score of 15.2. With the best 3 models, we get 15.7. We do beam search with a beam size of 8 and length normalization. All the ensemble results use a beam search decoder. The "steps" column gives the number of SGD updates that were performed before reaching the best dev BLEU score (greedy). We trained the models for an indefinite period of time, and manually interrupted training when dev set performance stopped improving.

Model	Dev BLEU		Test BLEU			Steps	Params (million)
	Greedy	Beam	Greedy	Beam	Ensemble		
Bérard et al. (2016a)	30.1	32.3	29.1	31.3	37.9†	12k	10.4
Cascaded	40.1	42.4	38.9	40.7	**43.8**		7.9 + 3.4
End-to-End	34.1	36.8	32.8	35.0	**43.9**‡	80k	6.7
Pre-trained	37.9	40.4	33.7	36.3		58k	
Multi-Task	37.7	40.4	34.6	37.4		60k	
Pre + Multi-task	37.0	40.3	35.1	37.6		41k	

TABLE 5.13: Results of the AST task on BTEC. † was obtained with an ensemble of 5 models. The non-cascaded ensemble‡ combines the *end-to-end*, *pre-trained*, *multi-task* and *pre-trained + multi-task* models. When ensembling the two best AST models, we get a BLEU score of 42.7. With three models, we get 43.7. As a measure of comparison with †, an ensemble of 5 instances of the *end-to-end* model gives a Test BLEU score of 41.1. Contrary to Section 5.1.3, we only give mono-reference scores.

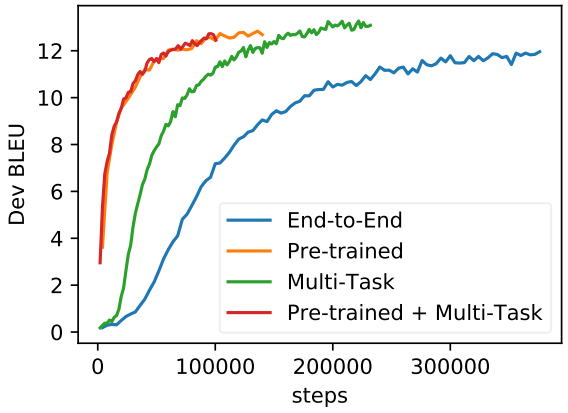


FIGURE 5.3: Greedy dev BLEU scores while training of four models for end-to-end AST of audiobooks. For the Multi-Task models, we only count the updates done on the AST task. The pre-trained models were actually trained for much longer if we count the training time of the MT and ASR models.

time of the pre-trained ASR and MT models needs to be taken into account (61k steps for MT and 490k steps for ASR). A useful property of our end-to-end models is that they are much more compact (at decoding time) than the cascaded model.

When combining several models in an ensemble, we obtain close performance to that of the cascaded baseline. However, our ASR baseline is rather weak (as demonstrated by our evaluation on LibriSpeech test-clean). A cascaded system that uses a more conventional ASR system would probably achieve better performance. Assuming perfect ASR, we would get a BLEU score of 19.3 on the Augmented LibriSpeech test set (performance of the NMT model), or 22.2 using a strong commercial NMT system.

The AST results presented on Augmented LibriSpeech demonstrate that our augmented corpus is useful, although challenging, to benchmark end-to-end AST systems on real speech at a large scale. We hope that our baselines (both for AST and MT) will be challenged in the future.

Figure 5.3 shows the progression of dev BLEU scores during training for our four LibriSpeech AST settings. We see that pre-training helps the model converge much faster. Eventually, the

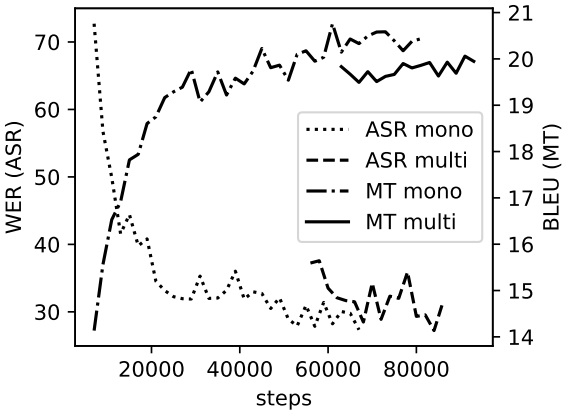


FIGURE 5.4: Augmented LibriSpeech dev BLEU scores for the MT task, and WER scores for the ASR task, with the initial (mono-task) models, and when multi-task training picks up.

End-to-End system reaches a good solution, but after three times as many updates. Multi-Task training does not seem to be helpful when combined with pre-training.

Figure 5.4 shows the progression of ASR and MT performance when training single models, and when we continue training these models as part of a multi-task training procedure (i.e., while sharing parameters with the AST task, and alternating SGD updates). Because the multi-task procedure focuses on the AST task (60% of all updates are done on this task), the MT and ASR results degrade slightly. Yet, we observe that the speech encoder and text decoder are still able to generalize well to other tasks.

Examples Table 5.14 gives examples of outputs by our AST models on LibriSpeech (the single end-to-end model, and the ensemble model).

Interestingly, the models seem to mishear certain words. For instance, “eyed” is translated as “mourions”, the French translation of “died” (phonetically very close to “eyed”). Similarly, it seems to mistake “narrowly” for “narrow alley” (which translates as “passage”).

We observe that the French language model is not very good. For example, “much superior” is translated as “beaucoup supérieure”, which is a word by word translation and not fluent French. This could probably be improved with more target language data (e.g., with multi-task training). On the other hand, the models seem very good at predicting punctuation symbols.

By looking at the transcriptions and French translations, we start to see why this is a difficult task. Some turns of phrase look quite “bookish” and almost archaic (“have you to depend upon”). For all three examples, we checked the speech input, and it matches perfectly the English transcription. The absence of punctuation in the English transcription makes it ambiguous and rather hard to understand. This may have an impact on the quality of the cascaded system and the multi-task models.

Figure 5.5 gives examples of alignments obtained by the attention mechanism of our LibriSpeech AST (pre-trained), ASR and MT models. We see that even though there is no monotonicity prior, the ASR model has learned to do a monotonous alignment. The AST attention is able to handle speech segmentation and local word reordering (see the alignment of “groupe domestique”). Both models are able to detect when the sentence actually starts (by ignoring the silence at the beginning).

but my sister must know you she must in case of need have you to depend upon		
– je veux , vous dis-je , que ma soeur vous connaisse ; je veux qu' elle puisse au besoin compter sur vous .	ma soeur doit vous connaître ; elle doit , en cas de fait , avez-vous entendu parler ?	mais ma soeur doit vous connaître ; elle doit , en cas de besoin , avez-vous à descendre ?
we eyed one another narrowly in passing and with no favour		
nous nous jetâmes un coup d' oeil peu amical .	nous mourions en passant , et sans faveur .	nous avons eu un autre passage en passant et sans faveur .
to the civil inquiries which then poured in and amongst which she had the pleasure of distinguishing the much superior solicitude of mister bingleys she could not make a very favourable answer		
quand elle entra dans la salle à manger , elle fut assaillie de questions parmi lesquelles elle eut le plaisir de noter la sollicitude toute spéciale exprimée par mr . bingley .	pour les inquiétudes civilisées qui portaient alors et parmi lesquelles elle avait le plaisir de distinguer la sollicitude beaucoup plus supérieure de m. bingley , elle ne pouvait pas faire une réponse très favorable .	aux enquêtes civiles qui s' enfonçaient et parmi lesquelles elle avait le plaisir de distinguer la sollicitude beaucoup supérieure de m. bingley , elle ne pouvait pas faire une réponse très favorable .

TABLE 5.14: Examples of outputs on the Augmented LibriSpeech test set, by the end-to-end model (middle column), and the ensemble model (right column). The top row is the English transcription, and the left column is the reference French translation.

FIGURE 5.5: Examples of forced alignments performed by the pre-trained AST model (top left), the ASR model (top right), and MT model (bottom). To better visualize the character-level outputs, the alignments are reversed: the x-axis corresponds to the target sequence and the y-axis to the source sequence.

Model	AST BLEU	MT BLEU	ASR WER
Single models	35.0	**49.2**	**13.8**
Pre-trained	36.3		
Pre-trained + Multi-Task	37.6	46.1	15.3
Multi-Task	37.4	43.8	17.2
Multi-Task joint	37.3	46.8	17.8
Ensemble (all)	**45.1**		

TABLE 5.15: Results of different Multi-task models on BTEC test, on the AST, MT and ASR tasks (with beam search decoding). The ensemble combines the 5 previous models. The "single models" row actually corresponds to three different models (AST, MT and ASR).

Multi-task We go a little more in depth and explore different multi-task training strategies on BTEC AST.

Table 5.13 reports the scores of a multi-task model which is initialized with pre-trained ASR and MT models (pre-trained + multi-task setting). It also shows the scores of an end-to-end AST model (mono-task and not pre-trained), and a pre-trained AST model (mono-task). We observed that pre-training helps the model converge faster, and to a slightly better score than the end-to-end model. However, combining pre-training with multi-task training did not seem to be useful.

We now train a multi-task model that is not pre-trained (all parameters are initialized at random), and compare its performance against the other settings in Figure 5.6.

Because we have a 3-fold parallel corpus (with speech segments aligned with their transcription and translation), we can train the AST, MT and ASR models jointly. This can be done by building a single large joint model (with two encoders and two decoders), and computing a training loss for each of the tasks. The joint loss is a weighted sum of the three tasks' losses: $\mathcal{L}_{joint} = 0.6 \times \mathcal{L}_{AST} + 0.2 \times \mathcal{L}_{MT} + 0.2 \times \mathcal{L}_{ASR}$. By optimizing $\mathcal{L}_{joint}$ we train the model on the three tasks at once. This is different from our previous multi-task training procedure, where we had three models with their own loss and optimized one after another.

Table 5.15 compares the performance of these different training strategies on the AST task, and on the auxiliary ASR and MT tasks. Figure 5.6 shows their training progression (dev BLEU as a function of SGD steps).

We see that the "Multi-task" and "Joint Multi-Task" models obtain similar AST performance and are equally fast to converge. Even though the joint model is more elegant, the multi-task framework of (Luong et al. 2016) is more convenient, as the different tasks can be trained with different data (which we have not tried). The joint multi-task model seems better on the auxiliary MT task. This may be an artifact of the (arbitrary) choice of task ratios. Finally, we see that the "pre-trained" and "pre-trained + multi-task' models are faster to train, although they converge to similar scores. The multi-task model that is also pre-trained performs better on the auxiliary tasks than the non pre-trained version.

By combining all these models into an ensemble of 5 models, we obtain our best score to date on the BTEC AST task: 45.1 BLEU, 1.3 points above the cascaded baseline.

Cold-start problem One big problem that we had with AST and ASR on Augmented LibriSpeech was to make the models converge. We often observed a long period at the beginning

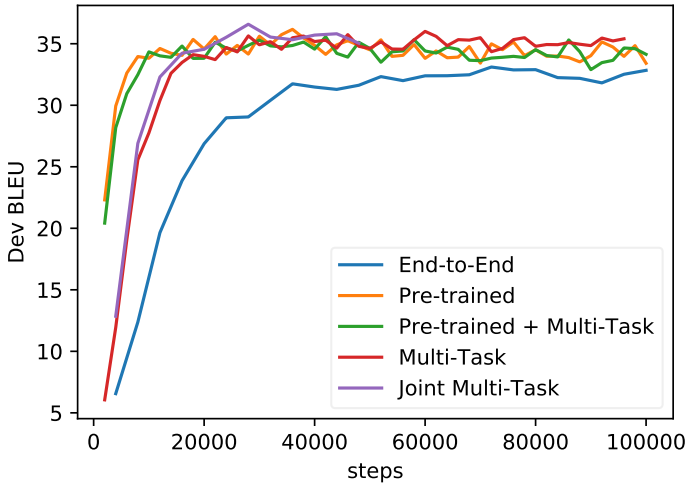


FIGURE 5.6: Progression of BTEC Dev BLEU scores of different AST models during training. The "pre-trained" model is a mono-task model (like "end-to-end"), but initialized with the parameters of existing MT and ASR models. The "pre-trained + multi-task" model combines this with multi-task training on AST, ASR and MT. "Multi-task" is initialized at random (like "end-to-end") but trained on the three tasks. "Joint multi-task" has a single training loss which is the sum of the AST, ASR and MT losses.

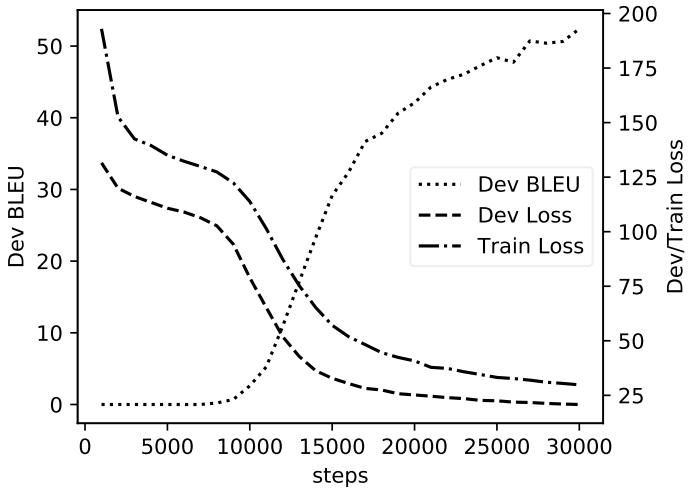


FIGURE 5.7: Progression of the dev BLEU score and dev/train loss of an attention-based ASR model trained on Augmented LibriSpeech, at the beginning of its training. We chose to show BLEU (and not WER) as it better illustrates the cold-start problem.

of training where the training loss seemed to plateau (and the dev BLEU score stayed at zero), which gave us little hope as to the future convergence of the model. This "cold-start" problem is illustrated by Figure 5.7. It turns out that, most of the time, if we wait long enough, the training loss gets past this plateau and starts decreasing again and the dev set performance starts increasing. This cold-start problem gets worse with more sophisticated encoders. For example, when we tried using non-linear convolutions (with a ReLU activation), the models took much longer to converge (even with batch normalization).

We think that this problem may be related to the attention mechanism.[21] The attention model is the main means of gradient propagation back to the encoder layers.[22] Because the input sequence is very long, if the attention model gets the alignment wrong, the gradient feedback is very likely to flow to the wrong place (or everywhere, but with a very small magnitude). This makes training the encoder very difficult. Conversely, it is very hard for the attention mechanism to learn a proper alignment when its input is just random noise.

Figure 5.8 shows that the attention model starts improving around the same time that we get past the training loss plateau (between 8k and 10k steps). At 6k steps, it has only figured out where the speech signal begins. Between 8k and 10k steps, the alignment starts to be more localized and monotonous. At 12k steps, the attention model has completely figured out the monotonous alignment, and there is almost no difference with the alignment at 80k steps. And yet, the Word Error Rate decreases from 90.1 (at 12k steps) to 25.9 (at 80k steps).

We think that in the early stage of training, our ASR model basically learns an English language model, without making use of the speech signal at all. At some point where the language model is good enough, the attention model starts improving. Then, the gradients can correctly flow up to the right place in the encoder, and the acoustic model can start to improve.

The dev cross-entropy plateaus around 110 (nats per sentence), which corresponds to ≈ 1.7 bits/character. This looks like a typical number for the entropy of a language model on the English language.[23] This supports our idea that what the ASR model does at this point is just language modeling (not conditioned on the input sequence at all).

We could probably facilitate the training of such models by giving a prior on monotonous alignments. If the attention model is constrained in some way to do a monotonous alignment at the beginning of training, then the gradients can be back-propagated correctly early on, and we could maybe avoid the cold-start problem.

Future work We have several leads for improving this work in the future.

- Fixing the cold-start problem. Some ideas in this direction are to:
  - Modify the attention mechanism to bias it towards monotonous alignments: with local attention (Luong et al. 2015b), convolutional attention (Chorowski et al. 2015), or by giving it positional information (Cohn et al. 2016).

[21] Thanks to Antonis Anastasopoulos, who suggested the idea.

[22] Apart from the decoder's initial state, this is the only thing that links the encoder and the decoder together. The attention model is supposed to ease training by providing a shorter path for the gradients.

[23] Brown et al. (1992) gives an upper bound for the entropy of printed English of 1.75 BPC, obtained with a word trigram language model estimated on the Brown corpus. The current winner of the Hutter Prize (http://prize.hutter1.net/), obtains an entropy of 1.22 BPC on the English Wikipedia.

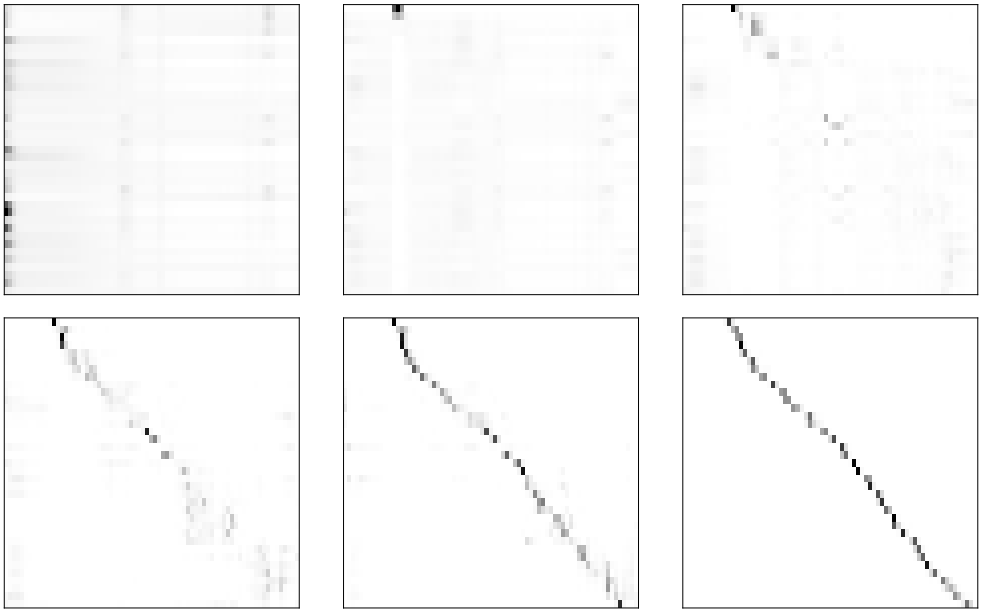


FIGURE 5.8: Forced alignments of the same segment performed by the attention mechanism of an ASR model (on LibriSpeech) at different stages of training (at 4k, 6k, 8k, 10k, 12k and 80k steps). The source spoken sentence is "their eclipse is never an abdication" (from the dev set).

  - Add a term in the training loss that penalizes non-monotonous alignments. This term could multiplied by a weight that decays with time, so that in the long-term the model can learn to do non-monotonous alignments.
  - Pre-process the speech data to remove blanks at the beginning and at the end (this could facilitate a monotonous alignment).
  - Do a better initialization and/or use batch-norm (or layer-norm). The difficulty with training the attention model and the acoustic model is maybe due to a vanishing gradient problem.

- Training a more expressive AST model, with non-linear convolutions and more layers (Weiss et al. 2017). A prerequisite for this step is to have fixed the cold-start problem (as "deeper" models are even harder to train).

- Exploring more multi-task scenarios, for example with additional ASR and MT data, or with more auxiliary tasks (image captioning, lip reading, parsing, grammar correction, etc.)

- Training a stronger cascaded baseline (e.g., with a baseline ASR model trained with Kaldi), for a more rigorous comparison.

- Training AST models that do not use the synthetic Google Translate target, and possibly use the noisier "other" corpus. This would probably be more difficult, as the alignments are noisier, and the translation is often not very straightforward (due to the nature of the training set).

Chapter 6

Neural Post-Editing

This chapter presents our contributions to the Automatic Post-Editing field (APE). As a reminder, APE consists in automatically improving the outputs of a Machine Translation system (MT), where this MT system is considered as a black-box, i.e., we do not have any view or control of its inner workings. We generally get instances of translation hypotheses generated by this system, along with their manually post-edited version by a human, and train a system to imitate this post-editing process. For a more detailed description of this task and of the previous contributions, see Section 1.2.

The first section defines the problem and the notations that we use. It also describes the datasets and evaluation methodology of the tasks that we seek to solve. The second section replicates several results from the literature on neural post-editing. The third section describes our original contributions to this domain: new neural post-editing architectures, along with an experimental validation and an analysis of these techniques.

6.1 Task Description

6.1.1 Definitions

Translation-based models As we have seen in Section 1.2, most solutions to Automatic Post-Editing cast this problem as a *Machine Translation* problem.

A machine translation model takes as input a sequence of symbols (most often words, sometimes phrases, subwords or characters), and outputs a sequence of symbols in another language. Machine translation models are trained with instances of such translations: generally pairs of sentences in the source language with their reference translation (human-made) in the target language.

However, the automatic post-editing task has a number of differences with machine translation:

1. Two input sequences are generally available: the translation hypothesis (that we will refer to as `MT`), and the original sentence in the source language (`SRC`).
2. Different metrics are used for automatic evaluation: in MT, popular metrics are BLEU (Papineni et al. 2002) or METEOR. In APE, the most popular metric is HTER (Snover et al. 2006).

3. A post-editing hypothesis is expected to be relatively close to the original MT hypothesis. Post-editing references are produced by humans, who aim to minimize the number of edits with respect to the MT hypothesis (as there is a direct correlation with the time spent post-editing). HTER rewards PE hypotheses that are a few edits away from this reference, and by extension, PE hypotheses which are not too far away from the original MT hypothesis.

4. The training corpora in APE are several orders of magnitude smaller than those available in Machine Translation: an MT corpus (or *parallel* corpus) can have millions of sentence pairs (e.g., Europarl, UN, etc.), while APE corpora generally contain a few thousands tuples (Potet et al. 2012b; Turchi et al. 2016, 2017). Since Neural Machine Translation often necessitates large amounts of data (Koehn et al. 2017), this constitutes a difficulty if we wish to apply such techniques to APE.

We call "translation-based", automatic post-editing models which output brand new sequences of words (or other units), i.e., models that treat APE as a machine translation problem, where the source language is `MT`, and the target language is `PE` (post-edited `MT`). We will refer to the outputs of such a system as `PE` hypotheses (or `APE`), and the gold standard to strive for as `PE` references.

Most contributions in APE belong to this category. However, several modifications have been proposed to handle each of the points mentioned earlier. Multi-source statistical post-editing was proposed by Béchara et al. (2011). Multi-source neural post-editing is also possible by using a multi-encoder architecture, with multiple attention heads (Junczys-Dowmunt et al. 2017a; Zoph et al. 2016a); or a log-linear combination of a translation model with a post-editing model (Junczys-Dowmunt et al. 2016b). The closeness between MT input and PE output can be enforced with tricks like Post-Editing Penalty (Junczys-Dowmunt et al. 2016b), or hard attention (Junczys-Dowmunt et al. 2017a). The small amounts of training data can be mitigated by using synthetic data, like simulated PE data (Negri et al. 2018; Potet et al. 2012a), or round-trip translations (Junczys-Dowmunt et al. 2016b).

Op-based models Another way to force the models to stay relatively close to the `MT` input, is to predict edit operations instead of new sequences of words.

This way of doing Automatic Post-Editing is probably closer to the reality of Post-Editing (by humans). Indeed, a human post-editor does not rewrite the translation hypothesis from scratch, but rather chooses which words to keep or remove, or new words to insert.

Our work is based on (Libovický et al. 2016), who train a model to predict edit operations instead of words. They predict 4 types of operations: `KEEP`, `DEL`, `INS(word)`, and `EOS` (the end of sentence marker). This results in a vocabulary with three symbols plus as many symbols as there are possible words to insert (about the same size as an MT vocabulary).

A benefit of this approach is that, even with little training data, it is straightforward to learn the identity function, i.e., the MT baseline. Predicting a sequence of `KEEP` symbols is equivalent to keeping the MT hypothesis as it is. This is useful, as we want to avoid a scenario where the APE system is weaker than the original MT system and only degrades its output.[1] However, this approach also has shortcomings that we shall see in the remainder of this work.

[1]This is not as easy as it seems. The APE literature contains many negative results, where the APE system degrades the MT hypotheses instead of improving them (Bojar et al. 2015; Potet et al. 2012a; Wisniewski et al. 2015). The MT baseline, generally an SMT system trained with large amounts of data, is often hard to beat.

We will refer to these methods as "op-based". We will refer to the output (i.e., a sequence of post-editing operations) of such a system as `OP`; and to the post-processed output (where each edit op has been applied to the input sequence) as `PE`.

Example If the `MT` sequence is `<The cats is grey>`, and the `OP` sequence is `<KEEP INS(cat) DEL KEEP KEEP INS(.)>`, this equals the following sequence of operations: keep `<The>`, insert `<cat>`, delete `<cats>`, keep `<is>`, keep `<grey>`, insert `<.>`
The result is the post-edited sequence `<The cat is grey .>`

We pre-process the data to extract such edit sequences by following the path with the smallest edit distance, where `KEEP` has a cost of zero, and `INS` and `DEL` have a cost of one. This is easily implemented by using a Levenshtein Distance algorithm (the same dynamic programming algorithm that is used for computing WER), with a substitution cost of $+\infty$. We did not find any advantage of incorporating substitutions: this doubles the size of the vocabulary, and the automatically extracted substitutions are often noisy.[2] Also, when performed by a human post-editor, a substitution actually corresponds to two atomic operations: a deletion and an insertion.

Algorithm 4 shows how we apply a sequence of edit operations to an MT hypothesis, to output a new post-editing hypothesis. There is a pointer to the current word being post-edited, which moves each time we read a `KEEP` or `DEL` operation. The insert operations insert a new symbol without moving the pointer. At line 6, we ensure that the pointer cannot move beyond the length of the MT hypothesis. If we cannot apply an operation (`KEEP` or `DEL`) because the pointer has reached the end of the hypothesis, then we interrupt post-editing and discard the next operations. At line 14, in case the post-editing sequence is too short, we complete the output with the MT symbols that are beyond the pointer. This is equivalent to padding the incomplete post-editing sequence with `KEEP` symbols. To delete a symbol, we need an explicit `DEL` operation.

The goal of these two heuristics is to "fix" broken post-editing sequences, by defaulting to the safe identity behavior. When training an NMT model to do post-editing, we often observe many broken outputs at the beginning of training. At the end of training, the post-editing sequences are generally sane ($\approx 99\%$ of the time).

6.1.2 Data & Evaluation

A number of post-editing corpora were made available (Potet et al. 2012b; Turchi et al. 2016, 2017). Such corpora generally contain aligned triples of sentences (`SRC`, `MT`, `PE`). Contrary to MT, post-editing corpora are often domain specific and cannot be used in other tasks.[3]

We choose to work in the framework of the APE tasks of the Workshop on Machine Translation (2016 and 2017 editions).[4] They consist in English-German Post-Editing in two translation directions. The datasets that are made available in the context of these tasks (Turchi et al. 2016, 2017) are described below. For each task, a train set, a dev set, and a test set are provided.[5] See Table 6.1 for information about the size of each dataset. Bojar et al. (2017, 2016) describe these tasks, along with the systems that were submitted and their results.

[2] The edit operations are extracted automatically so as to minimize the number of operations, without any linguistic consideration. Substitutions often concern words that are unrelated.

[3] An Automatic Post-Editing task consists in improving the outputs of a given MT system on a given domain.

[4] http://www.statmt.org/wmt17/ape-task.html

[5] The PE side of the test set was not distributed until after the end of the competition.

```python
def post_edit(hypothesis, edit_ops):
    i = 0   # index of current word being post-edited
    output = []   # result of the post-editing
    for op in edit_ops:
        if op is KEEP or op is DEL:
            if i >= len(hypothesis):
                break
            if op is KEEP:
                output.append(hypothesis[i])
            i += 1
        else:  # op is an insertion or EOS
            output.append(op)

    output += hypothesis[i:]  # symbols after i

    return output
```

Algorithm 4: Function which takes as input an MT hypothesis and a sequence of edit operations, and outputs a new (post-edited) hypothesis.

Task	Train	Dev	Test 2016	Test 2017	Additional
$en \rightarrow de$	23k (12k + 11k)	1000	2000	2000	4M + 500k
$de \rightarrow en$	25k	1000	–	2000	–

TABLE 6.1: Size of each available corpus for the 2017 edition of the APE Task (number of (SRC, MT, PE) sentence tuples). The additional data is synthetic (Junczys-Dowmunt et al. 2016a). The "4M" and "500k" datasets actually contain respectively 4.39M triples and 526k triples.

English to German This corpus ($en \rightarrow de$) was made available for the 2016 edition of the shared APE task, with a training set of 12k triples. New data was added (11k triples) to the training set for the 2017 edition. Table 6.2 gives an example of sentence tuple in this corpus. We generated the OP side by computing the shortest edit path between MT and PE.

The MT segments were obtained by translating IT domain data[6] with a strong SMT system[7] (Bojar et al. 2017). The post-edited segments were obtained thanks to a manual revision of the MT segments by professional translators, using the PET post-editing tool (Aziz et al. 2012). All the data, including dev and test data comes from the same source. This task spanned on both the 2016 and 2017 editions of the Workshop on Machine Translation (there are two test sets).

In addition to this real-world PE data, Junczys-Dowmunt et al. (2016a) built a large synthetic APE corpus. This corpus was obtained by round-translating a large monolingual dataset from German to English and back to German (using two large SMT systems). They also used a language model and TER statistics to sort this data according to its proximity to the real PE data distribution. This results in a large corpus of 4M triples, and a smaller but better quality (closer to the real data) corpus of 500k triples. This corpus was built (and released) as part of the authors' submission to the 2016 APE task. It was officially available for the 2017 edition as additional data.

[6]The source segments were provided by TAUS (https://www.taus.net/) and come from an IT vendor.
[7]Developed in the context of the QT21 project: http://www.qt21.eu/

SRC	Selection color boxes appear next to each selected item in the panel .
MT	Auswahlfelder neben jeder ausgewählten Element im Bedienfeld angezeigt werden .
PE	Auswahlfelder werden neben jedem ausgewählten Element im Bedienfeld angezeigt .
OP	KEEP INS(werden) KEEP INS(jedem) DEL KEEP KEEP KEEP KEEP KEEP DEL KEEP

TABLE 6.2: Example of sentence tuple from the WMT17 $en \rightarrow de$ APE corpus. The OP segments were generated by us (by taking the shortest edit distance.)

SRC	Bei versehentlichem Kontakt sofort mit viel frischem Wasser spülen .
MT	In case of accidental contact with either the skin immediately with much pull wire .
PE	In case of accidental contact rinse immediately with fresh water .
OP	KEEP KEEP KEEP KEEP KEEP INS(rinse) DEL DEL DEL DEL KEEP KEEP INS(fresh) INS(water) DEL DEL DEL KEEP

TABLE 6.3: Example of sentence tuple from the WMT17 $de \rightarrow en$ APE corpus.

German to English A corpus in the opposite translation direction ($de \rightarrow en$) was made available for the 2017 edition, with a training set containing 25k triples. Table 6.3 gives an example of sentence tuple in this corpus. The same pipeline was used as with the previous corpus (SMT system followed by manual post-editing). However, the data comes from the pharmaceutical domain.

These two corpora are used jointly by the Quality Estimation task,[8] which consists in predicting the HTER of an MT hypothesis (without knowing the reference), or to predict "bad" or "correct" labels for each word. In the English-to-German datasets, there are approximately 20% of "bad" words (as estimated by HTER) (Bojar et al. 2017). The German-to-English datasets have less errors, with approximately 12% of bad words. All the work that we present here is on the English-to-German task.

6.1.3 Experimental Protocol

Evaluation The main evaluation metric is HTER, and an additional (informative) evaluation is performed with BLEU. Both evaluations are case sensitive, and done against the tokenized post-editing reference.

- BLEU is a precision-based metric. It counts the proportion of n-grams which are also present in the reference, with a brevity penalty to penalize too short sentences.
- TER (Translation Edit Rate) is an edit-based metric. It counts the smallest number of edit operations (word insertions, deletions and substitutions, and phrase shifts) needed to change the hypothesis so that it matches the reference.

[8] http://www.statmt.org/wmt17/quality-estimation-task.html

This metric is called HTER (Human TER) when the reference is a human post-edited version of the hypothesis.

HTER correlates very well with human judgment (Snover et al. 2006), however it is generally too expensive in MT, because it requires a human to post-edit every newly generated translation hypothesis. In Automatic Post-Editing, the human post-editing reference is always available, which makes it an evaluation metric of choice.

- Another way of assessing the quality of an APE system is to measure the number of sentences that have been modified, and the number of sentences that have improved or deteriorated (according to sentence-level HTER). Chatterjee et al. (2017) propose a "Precision" metric, which counts the number of improved sentences divided by the number of sentences whose HTER has changed:

$$\text{Precision} = \frac{\text{Improved}}{\text{Improved} + \text{Deteriorated}} \tag{6.1}$$

6.2 Research Replication

We replicated two main works of Neural Automatic Post-Editing. The first, by Junczys-Dowmunt et al. (2017a), is a translation-based model which uses large amounts of synthetic data. This model was able to obtain unprecedented results on the WMT16 APE task. We also test this model in other conditions than those presented by the authors, in order to evaluate its robustness to low-resource settings. The second work, by Libovický et al. (2016), is an op-based model (the first that we know of), which gives promising results in a low-resource setting.

6.2.1 Translation-based Post-Editing

Models Junczys-Dowmunt et al. (2016a) proposed a neural post-editing model, which combines a translation model ($\texttt{SRC} \rightarrow \texttt{PE}$), with a monolingual post-editing model ($\texttt{MT} \rightarrow \texttt{PE}$) in an ensemble. By training these models on a large amount of synthetic post-editing data, they obtained excellent results on the 2016 APE task, and won the contest by a large margin.

Then, Junczys-Dowmunt et al. (2017a) proposed an improvement of this model, which uses two encoders and two attention models to read both the source (SRC) and the translation hypothesis (MT) at the same time (see Figure 6.1). This model was trained on the same data, and gives a large improvement over the combination of mono-source models.

We proceeded to replicate this model (M-CGRU), along with the monolingual model (CGRU) that they present in the same paper.

The architecture of the monolingual model is very similar to other models that we presented in the previous chapters. It has a bidirectional GRU encoder that reads the MT input, segmented as BPE units. The forward and backward states are concatenated, and are read by the attention mechanism. These hidden states are also averaged time-wise and used to initialize the decoder (with a linear projection and a $\tanh$ activation).

The decoder is a conditional GRU (Sennrich et al. 2017) which outputs BPE units. It has two GRU cells that work together:

$$s_t^{[1]} = \text{GRU}^{[1]}(s_{t-1}^{[2]}, E'(\widetilde{z}_{t-1})) \tag{6.2}$$

$$c_t = \text{look}(s_t^{[1]}, (h_i)_{i=1}^T) \tag{6.3}$$

$$s_t^{[2]} = \text{GRU}^{[2]}(s_t^{[1]}, c_t) \tag{6.4}$$

$$x'_t = s_t^{[2]} \oplus c_t \oplus E'(\widetilde{z}_{t-1}) \tag{6.5}$$

$$y_t = W_{voc} \tanh(W_{out} x'_t + b_{out}) + b_{voc} \tag{6.6}$$

where $\text{GRU}^{[1]}$ and $\text{GRU}^{[2]}$ are two cells of size n. The context vector $c_t \in \mathbb{R}^{2n}$ is computed by a vanilla global attention mechanism (Bahdanau et al. 2015), which uses a feed-forward network with one hidden layer of size k.
$E^{|V'| \times m}$ is the target embedding matrix, with m the embedding size and $|V'|$ the target vocabulary size. At training time $\widetilde{z}_{t-1}$ is the previous ground truth symbol (teacher forcing). At test time, $\widetilde{z}_{t-1}$ is either the argmax of y_{t-1} (greedy decoding) or the output of the beam search decoder. $E'(\widetilde{z}_{t-1})$ maps this symbol to an embedding vector of size m. $W_{out} \in \mathbb{R}^{m \times (3n+m)}$, and $W_{voc} \in \mathbb{R}^{|V'| \times m}$. $y_t \in \mathbb{R}^{|V'|}$ contains a score for each item in the target vocabulary.

As illustrated by Figure 6.1, the bilingual model (M-CGRU) has two bidirectional GRU encoders, one for `MT` and one for `SRC`. They result in two sequences of hidden states $(h_i)_{i=1}^T$ and $(h'_j)_{j=1}^{T'}$. Their average states are concatenated and used to initialize the CGRU decoder:

$$h_i = \vec{h}_i \oplus \overleftarrow{h}_i \qquad h'_j = \vec{h}'_j \oplus \overleftarrow{h}'_j \tag{6.7}$$

$$h_{-1} = \left(\frac{h_1 + \ldots + h_T}{T}\right) \oplus \left(\frac{h'_1 + \ldots + h'_{T'}}{T'}\right) \tag{6.8}$$

$$s_0^{[1]} = \tanh(W_{init}^{[1]} h_{-1} + b_{init}^{[1]}) \tag{6.9}$$

$$s_0^{[2]} = \tanh(W_{init}^{[2]} h_{-1} + b_{init}^{[2]}) \tag{6.10}$$

where T and T' are respectively the lengths of the `MT` and `SRC` sequences. $\vec{h}_i$ is the i$^{\text{th}}$ hidden state of the `MT` encoder's forward encoder. $\overleftarrow{h}_i$ is the i$^{\text{th}}$ backward state, and $\vec{h}'_j$ and $\overleftarrow{h}'_j$ are the j$^{\text{th}}$ forward and backward states of the `SRC` encoder.

There are two attention heads, one for each encoder. Their context vectors $c_t^1 = \text{look}^1(s_t^{[1]}, (h_i)_{i=1}^T)$ and $c_t^2 = \text{look}^2(s_t^{[1]}, (h'_j)_{j=1}^{T'})$ are simply concatenated ($c_t = c_t^1 \oplus c_t^2$) and used to update the decoder's state and predict the next symbol (following Equations 6.4 and 6.5).

Data Junczys-Dowmunt et al. (2017a) train their models on the *en* → *de* data from the WMT16 APE task. They also use the synthetic data that was produced by Junczys-Dowmunt et al. (2016a).

All the data (SRC, MT and PE) is truecased and then segmented into BPE units. We use the same truecasing model and joint BPE model as the authors.[9] We merge the BPE units from the system's outputs and detruecase the segments before running the evaluation.

[9] Distributed by the authors on: `https://marian-nmt.github.io/examples/postedit/`

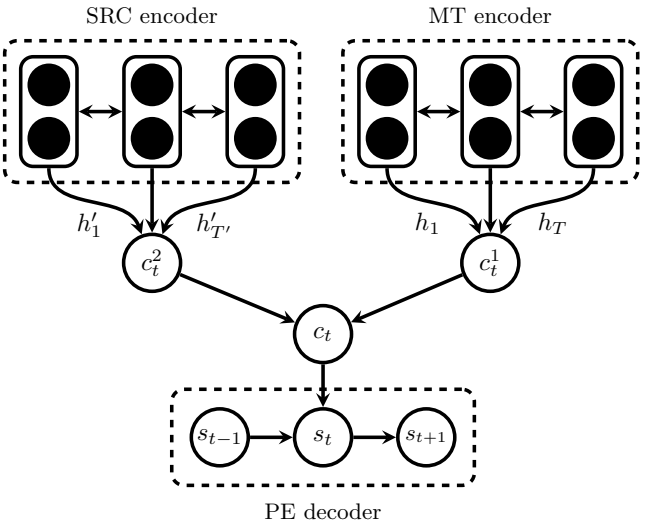


FIGURE 6.1: Bilingual model for Neural Post-Editing, with an MT encoder and an SRC encoder, and two attention heads whose context vectors are concatenated. The PE decoder can look anywhere in the translation hypothesis and source-language sequence to do its predictions.

We train models with corpora of different sizes: small, medium, large, XL and XXL (see Table 6.4). The XL and XXL models use all the synthetic data available, and are comparable (respectively) to Junczys-Dowmunt et al. (2017a) and Chatterjee et al. (2017). While the XL models use only the data available for the 2016 edition (12k segments of real PE data, plus synthetic data), the XXL model makes use of all the real PE data available for the 2017 edition (11k additional real PE segments).

In addition to these large models, we also train smaller models that use less data. The "Large" setting uses all the real PE data available, along with the medium-size synthetic corpus (500k). The "Medium" and "Small" settings are more challenging, as they use only real PE data: the 23k segments available for the 2017 edition, and the 12k segments from the 2016 edition.

For the "Small", "Medium" and "Large" (smaller-than-XL) settings, we follow Sennrich et al. (2016a)'s recommendations[10] and only keep the BPE units whose frequency in the train set is above a threshold of 5. This is important to avoid segmenting the dev and test data with BPE units that are unknown at training time. Because the large BPE model provided by the authors was estimated on all the data available, and we are using only a subset of this training data, it is likely that many BPE units are out-of-vocabulary.

As we use less BPE units, we get smaller vocabularies and longer sequences on average (more aggressive segmentation). Table 6.4 gives the size of each corpus once pre-processed. To be able to share the MT and PE embeddings, we use the same vocabulary, extracted by concatenating the MT and PE sides of the training corpus.

Settings Our XL and XXL models have the same size as Junczys-Dowmunt et al. (2017a)'s CGRU and M-CGRU models: we use GRUs of size $n = 1024$, embeddings of size $m = 512$, and an attention mechanism of size $m = 2048$. We use the same amount of dropout (0.2 on the GRU's inputs and outputs). We also drop source and target symbols at random during training with a probability of 0.2.

[10]The details and the scripts are available here: https://github.com/rsennrich/subword-nmt

Corpus	Content	Segments	Vocab size			Average PE length
			SRC	MT	PE	
Small	12k	12k	7602			26.6
Medium	23k	23k	11181			24.2
Large	500k + 23k	986k	34505	39327		21.8
XL	4M + 500k + 12k	5.16M	39028	39770		
XXL	4M + 500k + 23k	5.38M	40583	40844	41110	

TABLE 6.4: Size of all the data splits that we use for training translation-based automatic post-editing models, on the $en \rightarrow de$ task. The last column gives average BPE counts per segment in the PE side of the dev set. In the "Large", "XL" and "XL" datasets, the real PE corpus (12k or 23k) is oversampled 20 times. In the "Small" and "Medium" settings, a single joint vocabulary is used for SRC, MT and PE. For "Large" and "XL", we share the MT and PE vocabularies.

Our smaller-than-XL models use a larger dropout rate of 0.4 (but the same amount of word dropout). The "Large" models is half as large, with GRUs of size $n = 512$, embeddings of size $m = 256$, and an attention model of size $k = 1024$. The "Medium" and "Small" models are again half as large (but both of equal size).

To avoid overfitting, the smaller-than-XL models do not have a non-linear output layer (x'_t is directly mapped to the vocabulary size in eq. 6.6). Also, we share the embedding matrix between the `MT` encoder and the `PE` decoder. The "Small" and "Medium" models go even further, and use the embedding matrix for vocabulary projection ($W_{voc} = E'$) (Press et al. 2017).

Our implementation has a few simplifications compared to Junczys-Dowmunt et al. (2017a)'s implementation.[11] While the authors stop training automatically using a "patience" parameter,[12] we stop manually once we observe that dev set performance does not improve anymore (except for our smaller models that are trained for a fixed number of steps). Contrary to the authors, we do not use layer normalization, and we do not save a moving average of the model's weights.

Like the authors, we train using Adam, with a learning rate of 0.0001 and a batch size of 64. We save the model and evaluate its performance on the dev set every 10k steps. We select the best model according to its TER score on the dev set (using greedy decoding).[13] For training, we read 100 batches ahead of time and sort their content according to target sequence length. This increases training speed as batches contain sentences of similar size, which reduces the amount of padding that is necessary. We initialize all the weights (except biases) to a centered normal distribution with $\sigma = 0.01$.

The smaller-than-XL models use a batch size of 32, a learning rate of 0.001, and a look ahead of 10 batches. We run the evaluation on the dev set every 1000 steps. The "Large" and "Medium" models are trained for 150k steps, and the "Small" model for 75k steps.

Like Junczys-Dowmunt et al. (2017a), we use a maximum sequence length (input and output) of 50 BPE units. The "Small" model has a maximum output length of 60, because its sequences are longer on average.

[11] Available here: `https://marian-nmt.github.io/`
[12] Training is interrupted if the best score (dev loss) has not improved for 10 checkpoints.
[13] The authors select the best model according to its dev loss, with beam search decoding.

Results Table 6.5 compares the performance of different decoding strategies on the dev set. We see that for all model sizes, the beam search decoder (with a beam size of 12 and length normalization) brings a large improvement in TER scores over greedy decoding. Averaging several checkpoints from the same training instance, like (Junczys-Dowmunt et al. 2016b, 2017a) also gives a consistent improvement.

Finally, we see that using an ensemble of 4 models (with beam search decoding) gives a huge boost in TER scores, in particular for the smaller models. Ensembling the averaged checkpoints (last column) does not seem to be better than ensembling the best checkpoints.

Table 6.6 gives the TER and BLEU scores of these techniques on Test 2016 and Test 2017. Our single-model scores are obtained with beam search decoding from averages of 4 checkpoints. We average single-model scores from 4 different runs. The ensemble scores are obtained by ensembling 4 non-averaged models.

We see that our implementation of the CGRU and M-CGRU models compares favorably with (Junczys-Dowmunt et al. 2017a) on the XL dataset, even though it is simplified (no layer-norm and no moving average). When using all the training data available for the 2017 edition (XXL model), we obtain similar results to Chatterjee et al. (2017)'s single model.[14] This is not surprising as their approach is very similar (multi-encoder, trained with the same data).

More surprisingly, our single XXL model's performance is not very far off from that of Junczys-Dowmunt et al. (2017b), even though they use an ensemble of 4 M-CGRU models that are trained on much more data (an additional 15M synthetic corpus is used). They seem to get a slight improvement in scores when running the APE system twice, i.e., feeding the output of a first pass to a second pass of automatic post-editing. We did not observe any improvement when doing this with our models.

Interestingly, our smaller-than-XL models obtain very decent results, even though they use much smaller amounts of data. This contradicts our prior belief that positive results on APE using neural translation-based methods, by Junczys-Dowmunt et al. (2016a), had only been possible by using huge amounts of synthetic data. We are able to show that Neural APE is a good approach (at least on this task), even with realistic amounts of PE data and a reasonable training budget.[15] Figure 6.3 shows that our Medium model is able to do a sensible alignment between its `PE` output and the `SRC` and `MT` inputs, even though it was trained with only 23k segments (e.g., `dar/stellt` with `represents`).

Even the "Small" ensemble, which uses only 12k segments of data is able to outperform the MT and SPE baselines by a large margin. Yet, we see a large difference in scores between the "Small" and "Medium" models. This suggests that the gap between 12k and 23k segments is consequential. To confirm this, we see that the XXL models largely outperform the XL models, even though they just have 11k extra segments of real PE data (out of more than 4M synthetic segments).

Table 6.7 shows the number of modified sentences on Test 2017, by different APE models, as well as the number of improved and deteriorated sentences. We consider a sentence as modified if the `APE` output is not identical to the `MT` input.
A sentence is improved if: $\text{TER}(\texttt{APE}, \texttt{PE}) < \text{TER}(\texttt{MT}, \texttt{PE})$, and deteriorated if the TER is strictly greater.

[14] Their ensemble of models is the winner of the 2017 edition.

[15] The XL and XXL model took 150 hours to train on a single GTX 1080 Ti. The single Small, Medium and Large models took respectively 11h, 23h and 30h.

Model	Dev TER					Steps
	Greedy	Beam search	**Average**	**Ensemble**	Ensemble + Average	
XXL	20.5	19.7	19.7	–	–	510k
XL	21.6	21.0	20.6	–	–	640k
Large	22.7	22.2	21.8	21.1	21.0	140k
Medium	24.0	23.3	22.9	21.9	22.0	118k
Small	26.3	25.5	24.9	23.7	23.8	45k

TABLE 6.5: Comparison of different decoding strategies for APE on the $en \rightarrow de$ dev set. The beam search decoder has a beam size of 12. The ensembles contain 4 models. The non-ensemble results for smaller-than-XL models are averaged over 4 runs (the same 4 models used for ensembling). The "average" column corresponds to averaging the weights of the 4 best checkpoints (according to greedy TER) of each model. For the "XL" and "XXL" models we average 8 checkpoints. The "steps" column is the average number of SGD steps (over 4 runs) before reaching the best performance on the dev set.

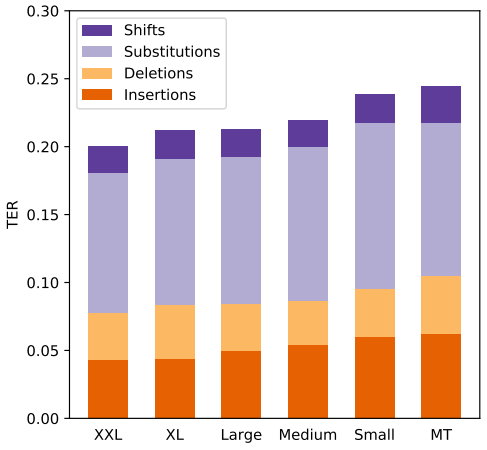


FIGURE 6.2: TER statistics on Test 2017 by our translation-based APE models (single averaged XL and XXL models, and ensembles of Large, Medium and Small models). The right-most bar is the MT baseline ("do-nothing" scenario). The height of a colored segment represents the ratio of this type of error over the number of reference words. The total height of a bar is the TER of the corresponding model.

All our single-models modify approximately the same number of sentences. However, when trained with more data, they improve more and deteriorate fewer sentences (higher precision). Interestingly, the ensembles tend to modify fewer sentences than the single models. The single "Small" model is actually detrimental as it produces more deteriorated sentences than improved ones (precision lower than 50%). Its TER and BLEU scores on Test 2017 are also worse than the MT baseline. It is only beneficial when used in an ensemble. Figure 6.2 shows the statistics of each type of TER operation (insertion, deletion, substitution and shift) of our different models on Test 2017.

6.2.2 Op-based Post-Editing

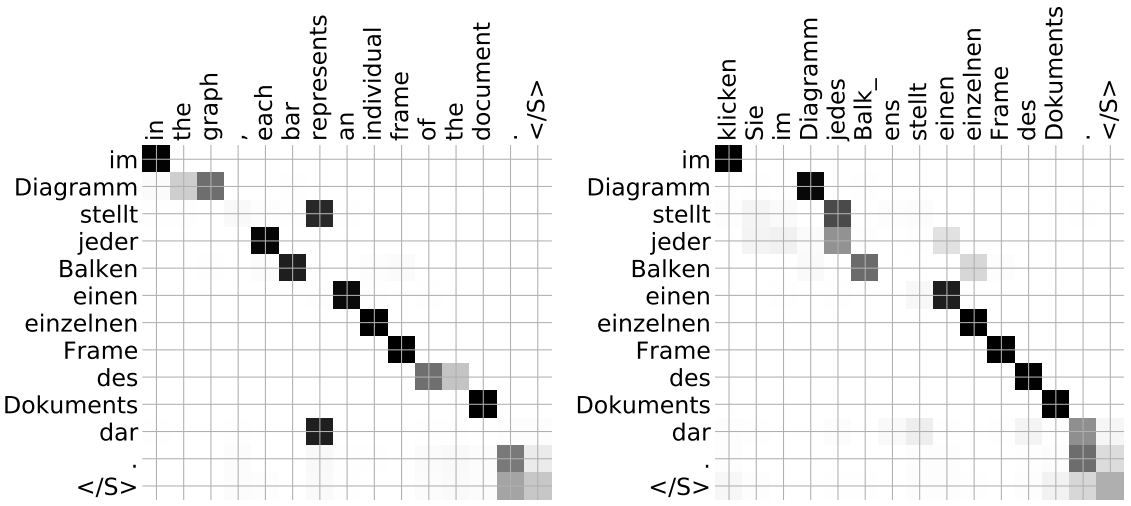
We were also interested in extending the work from Libovický et al. (2016). This was the only successful work on Neural APE that we knew of that used only real PE data (i.e., very limited

Model	Data	Dev		Test 2016		Test 2017	
		TER	BLEU	TER	BLEU	TER	BLEU
Bojar et al. (2017, 2016)							
MT Baseline	–	24.8	62.9	24.8	62.1	24.5	62.5
SPE Baseline	Small/Medium	–	–	24.6	63.5	24.7	63.0
Junczys-Dowmunt et al. (2017a)							
AMU - CGRU	XL	22.0	68.1	22.3	66.9	–	–
AMU - M-CGRU		20.8	69.3	20.7	68.6		
ensemble (4)		20.1	70.2	19.9	69.4		
Junczys-Dowmunt et al. (2017b) – WMT 2017 (2nd)							
AMU - ensemble (4)	XXXL†	19.7	70.6	19.3	70.3	19.8	69.4
ensemble (4)[2]		–	–	19.2	70.5	19.8	69.5
Chatterjee et al. (2017) – WMT 2017 (1st)							
FBK - single	XXL	19.8	70.7	–	–	20.3	69.1
ensemble (8)		19.2	71.9	19.3	70.9	19.6	70.1
Our results							
CGRU	XL	22.3	68.1	22.5	66.9	22.5	66.8
M-CGRU		20.6	69.9	20.8	68.8	21.2	67.8
	XXL	**19.7**	70.7	**19.9**	70.0	**20.0**	69.3
M-CGRU	Large	21.8	67.9	21.5	67.2	21.9	66.3
ensemble (4)		21.1	68.7	20.9	68.0	21.3	67.2
M-CGRU	Medium	22.9	66.1	22.4	66.0	22.9	64.6
ensemble (4)		21.9	67.5	21.6	67.0	21.9	66.0
M-CGRU	Small	24.9	63.5	24.7	62.9	25.1	61.7
ensemble (4)		23.7	64.9	23.7	64.0	23.8	63.4

TABLE 6.6: Scores of different translation-based model on the WMT16 and WMT17 $en \rightarrow de$ APE tasks. The scores are obtained with a beam search decoder of size 12. Our smaller-than-XL non-ensemble results are averaged over 4 different training runs. The XXXL† dataset contains even more synthetic data, with 21M segments in total. The SPE Baseline is a monolingual PBMT model (Simard et al. 2007), and the MT Baseline corresponds to keeping the MT hypothesis as it is.

Model		Modified	Improved	Deteriorated	Precision
FBK	Ensemble (8)	1607	1035	334	75.6%
AMU	Ensemble (4)	1583	1040	322	76.4%
Large	Ensemble (4)	1474	872	327	72.7%
Medium		1539	854	427	66.7%
Small		1510	694	532	56.6%
XXL	Average	1607	1025	361	74.0%
XL		1604	942	407	69.8%
Large		1538	850	402	67.9%
Medium		1605	802	517	60.8%
Small		1616	650	656	49.8%

TABLE 6.7: Number of modified and improved/deteriorated sentences by our APE models on Test 2017. The full corpus has 2000 sentences. The FBK and AMU baselines are the winners of the 2017 APE Task (Chatterjee et al. 2017; Junczys-Dowmunt et al. 2017b). Their statistics are taken from (Bojar et al. 2017).



(A) SRC → PE alignment (B) MT → PE alignment

FIGURE 6.3: Examples of forced alignments by the SRC → PE and MT → PE attention mechanisms of our Medium M-CGRU model.

amounts of data). This was also the first contribution to propose to train a model that predicts edit operations (op-based model).

We first replicated their results on the WMT 2016 APE task (i.e., "Small" setting, Test 2016 evaluation). The authors describe their contribution in two tasks: automatic post-editing and multimodal translation. However, they are much more focused on the latter, and the paper lacks many technical details about their APE model.

Our work is not an exact replication of their work, as we use different parameters as the authors. Nevertheless, we strived to obtain similar results, by using similar techniques and by following the same experimental protocol: same task and datasets, and same data pre-processing and post-processing (with edit operations).

Data Table 6.8 describes the datasets that we use to train our op-based models. Libovický et al. (2016) use only the "Small" real PE corpus that was available for the 2016 edition. We train models in this setting to compare our results with theirs. We also train the same models with the Medium and Large datasets, to see how these techniques improve with more data, and how they compare against the translation-based models tested previously. We do not apply any pre-processing, except for the edit-op extraction described in the previous section. Contrary to the previous subsection, in the "Large" dataset, we oversample the real PE data 10 times (and not 20 times).

Our models are trained by minimizing the cross-entropy between the predicted sequences of edit operations, and the ground truth sequences of edit operations (that we automatically extracted). However, the TER and BLEU evaluation is performed against the untouched PE reference, after a post-processing step that transforms our sequences of edit operations to sequences of words.

Model description Our model is a basic attention-based sequence to sequence model. The encoder is a bidirectional LSTM that reads words, and the decoder is a single LSTM that outputs

Corpus	Content	Segments	Vocab size			Max length		
			SRC	MT	OP	SRC	MT	OP
Small	12k	12k	9336	18985	9076	33	37	45
Medium	23k	23k	11923	27093	13949			
Large	500k + 23k	756k	30000			40	40	50

TABLE 6.8: Size of the data splits that we use for training op-based post-editing models on the $en \rightarrow de$ task. In the "Large" dataset, the real corpus (23k) is oversampled 10 times. The "max length" column is the maximum sequence length that we set in our models for this type of input, chosen to cover 99% of the training corpus.

edit operations. We use a "generate first" strategy, which consists in generating the next symbol before updating the state:

$$c_t = \text{look}(s_{t-1}, (h_i)_{i=1}^T) \tag{6.11}$$
$$y_t = W_{voc}(s_{t-1} \oplus c_t \oplus E'(\widetilde{z}_{t-1})) + b_{voc} \tag{6.12}$$
$$s_t = \text{update}_{dec}(s_{t-1}, c_t \oplus E'(\widetilde{z}_t)) \tag{6.13}$$

where $(h_i \in \mathbb{R}^{2n})_{i=1}^T$ is the sequence of hidden states produced by the encoder, $E' \in \mathbb{R}^{|V'| \times m}$ is the target embedding matrix, and $W_{voc} \in \mathbb{R}^{|V'| \times (3n+m)}$ maps the output of the decoder to vocabulary size. At training time, $\widetilde{z}_t$ is the ground truth symbol. At test time, it is the prediction of the model (argmax of y_t or beam-search decoding).

The initial states of the encoder's LSTMs are trainable parameters of the model. The decoder's state is initialized with the average of the encoder's hidden states (concatenation of forward and backward states), followed by a linear projection to the right size and a $\tanh$ activation.

We use LSTM cells of size $n = 128$, embeddings of size $m = 128$, and a global attention model (Bahdanau et al. 2015) of size $k = 128$.

The model parameters (except biases) are all initialized to a centered normal distribution with $\sigma = 0.1$.

Contrary to the authors, we train our models with pure SGD, with a batch size of 32 and an initial learning rate of 1.0. We decay the learning rate by 0.95 every epoch starting at the fourth epoch. We found SGD (with a carefully chosen learning rate and decay strategy) to outperform Adam on this task. Adam shows very early overfitting effects, while SGD seems to act as some sort of regularization. We apply dropout on the inputs of the LSTMs with a 50% rate. We also do dropout on the decoder's initial state (before the linear projection).

We train two kinds of models: a monolingual model, which only uses the translation hypothesis `MT`; and a multi-source model, like Libovický et al. (2016), which uses both `MT` and `SRC`. Like in previous subsection, the multi-source model has two bidirectional encoders, and two attention mechanisms whose context vectors are concatenated. The average hidden states of both encoders are also concatenated and used to initialize the decoder's state.

Experiments and results Table 6.9 shows the results of our op-based models on the WMT 2016 task.

We see that our "Small" models compare favorably with the similar (multi-source) model by Libovický et al. (2016). Using both the SRC and MT inputs in a multi-encoder model does not seem to be useful, as the TER and BLEU scores are almost identical to those of the mono-source model. The "Small" models are only slightly better than the MT baseline (which keeps the MT hypothesis unchanged). Their TER scores are also slightly better than those of the monolingual Statistical Post-Editing baseline (Simard et al. 2007).

Adding more data (Medium and Large settings) seems to improve the results, though much less dramatically than with the translation-based models of previous subsection.

Model	Data	Dev		Test 2016	
		TER	BLEU	TER	BLEU
MT Baseline	–	24.8	62.9	24.8	62.1
SPE Baseline	Small	–	–	24.6	63.5
Libovický et al. (2016)	Small	24.4	–	24.3	63.3
Mono-source		24.2	64.3	24.2	63.5
Multi-source		24.2	64.3	24.2	63.4
Mono-source	Medium	23.9	64.9	23.7	64.2
Multi-source		24.0	64.7	23.8	64.0
Mono-source	Large	23.3	66.4	23.3	65.7
Multi-source		23.4	65.9	23.2	65.2

TABLE 6.9: Replication of the results from Libovický et al. (2016) on the WMT16 APE task, and comparison with models trained with larger amounts of data. Our results are obtained with greedy decoding, and averaged over 4 different runs.

6.3 New Models

In this section, we propose improvements to the op-based model of Libovický et al. (2016), and perform more extensive experiments. We propose a new task-specific attention mechanism, and a different architecture for incorporating the source sequence (SRC).

6.3.1 Hard Attention

Context Most Neural Post-Editing methods (Junczys-Dowmunt et al. 2016a, 2017a; Libovický et al. 2016) use a global attention mechanism, akin to Bahdanau et al. (2015).

This attention model is a feed-forward network that takes as input the current state of the decoder s_{t-1} and predicts a weight for each of the encoder's hidden states h_i:

$$r_{ti} = v_{att}^{\top} \tanh(W_{att}(h_i \oplus s_{t-1}) + b_{att}) \tag{6.14}$$

$$\alpha_{ti} = \mathrm{softmax}(r_{ti}) \tag{6.15}$$

with $W_{att} \in \mathbb{R}^{k \times 3n}$, $b_{att} \in \mathbb{R}^k$ and $v_{att} \in \mathbb{R}^k$. The $\mathrm{softmax}$ function ensures that the weights α_{ti} constitute a probability distribution over the input length, i.e., $\sum_i^T \alpha_{ti} = 1$ and $\forall i, \alpha_{ti} \in (0, 1]$. Some attention models differ as to how they compute r_{ti}. Some attention models use additional information (Cohn et al. 2016); or a different aggregation function, like multiplicative attention (Luong et al. 2015b). The model presented above is however the most popular attention

model in NMT and Neural Post-Editing (Chatterjee et al. 2017; Junczys-Dowmunt et al. 2016a; Libovický et al. 2016).

These weights are then used to compute a weighted average of the encoder's hidden states:

$$c_t = \text{look}_{global}(s_{t-1}, (h_i)_{i=1}^T) = \sum_{i=1}^{T} \alpha_{ti} h_i \tag{6.16}$$

The decoder uses this attention vector c_t to help predict the next symbol $\widetilde{z}_t$ and update its state.

The parameters of the attention model W_{att}, b_{att}, v_{att} are trained jointly with the rest of the model, so as to minimize the translation loss. Very often, this results in an informative attention model, which will put more weight on the input words (or more precisely their hidden state) that are useful for predicting the next word. Intuitively, this results in a soft-alignment between the input sequence and the output sequence.

Hard attention In our op-based framework, we do not predict words, but edit operations. Because an edit operation always applies to a specific word in the `MT` input sequence, we can do stronger assumptions on the `MT` $\rightarrow$ `OP` alignment.

Instead of a soft unconstrained attention mechanism which can look at the entire input, we propose to use a hard attention mechanism, which directly aligns current decoder state s_t with a single `MT` word x_i and its encoder state h_i.

The $t \rightarrow i$ alignment is straightforward: i is the number of `KEEP` and `DEL` symbols in the decoder's history $(\widetilde{z}_1, \ldots, \widetilde{z}_{t-1})$ plus one:

$$i = 1 + \sum_{k=1}^{t-1} \mathbb{1}(\widetilde{z}_k = \texttt{DEL}) + \mathbb{1}(\widetilde{z}_k = \texttt{KEEP}) \tag{6.17}$$

$$\text{look}_{hard}(s_{t-1}, (h_j)_{j=1}^T) = h_i \tag{6.18}$$

where $\widetilde{z}_k$ is the token generated by the `OP` decoder at the k$^{\text{th}}$ time step (or the k$^{\text{th}}$ ground truth symbol during training), and $(h_j)_{j=1}^T$ is the sequence of hidden states produced by the `MT` encoder. Actually, this is not a trained model (it has no trainable parameter), but more of a heuristic.

Following the example presented in Section 6.1 `"The cats is grey"`, if the decoder's past output is `"KEEP DEL INS(cat)"`, the next token to generate is naturally aligned with the third input word ($i = 3$). The decoder has decided to keep `"The"` and to replace `"cats"` with `"cat"`. Now, it has to decide what to do with the third word `"is"` (delete it, keep it, or insert a new word before it).

Junczys-Dowmunt et al. (2017a) also propose a hard attention model for automatic post-editing. However, their model is not as straightforward as it applies to translation-based post-editing, and it does not seem to be better than global attention. In addition to generating words in the output, they generate special `STEP` symbols that move a pointer in the source sequence.

Model	Data	Dev		Test 2016		Test 2017	
		TER	BLEU	TER	BLEU	TER	BLEU
MT Baseline	–	24.8	62.9	24.8	62.1	24.5	62.5
SPE Baseline	Small/Medium			24.6	63.5	24.7	63.0
Mono global	Small	24.2	64.4	24.1	63.7	24.2	63.3
Mono hard	Small	**23.4**	66.3	**23.3**	65.5	**23.4**	64.7
Mono global	Large	23.3	66.4	23.1	65.7	23.3	65.1
Mono hard	Large	**22.9**	66.9	**22.8**	66.2	**23.0**	65.5

TABLE 6.10: Comparison of global-attention models with hard-attention models. All the models shown here are monolingual (they do not use the source sequence SRC). These results are obtained with a beam search decoder, with a beam size of 6 and length normalization. We average the 4 best checkpoints of each model. The results are averaged over 4 different runs.

Experiments To compare our hard attention with the global attention mechanism, we train similar models to the op-based models of Section 6.2.2. We use the exact same architecture, hyperparameters and training settings, except for the attention, which is replaced by our "hard attention" heuristic. Table 6.10 shows the results of our experiments on the $en \rightarrow de$ APE task.

We see that our mono-source models with hard attention fare much better than the global attention one. The gap is smaller when more training data is available, because the global attention model is able to learn a better alignment.

Figure 6.4 shows examples of alignments performed by the global attention mechanism. We observe that when the global attention model is trained on a small amount of data ("Small" 12k dataset), the alignment is very fuzzy. When more training data is available ("Large" dataset), the soft alignment becomes more focused and seems to draw closer to our "hard" alignment. But even then, we get better TER and BLEU scores with a hard attention.

The use of a hard attention mechanism makes particular sense in low-resource scenarios, where the model is unable to learn a meaningful soft alignment.

6.3.2 Multi-Source Post-Editing

Multi-Encoder In automatic post-editing, two source sequences are generally available: the machine translation hypothesis that is being post-edited (MT), and the original source-language sequence (SRC). While it is tempting to only use the MT hypothesis, the source sentence is often necessary to ensure translation adequacy.

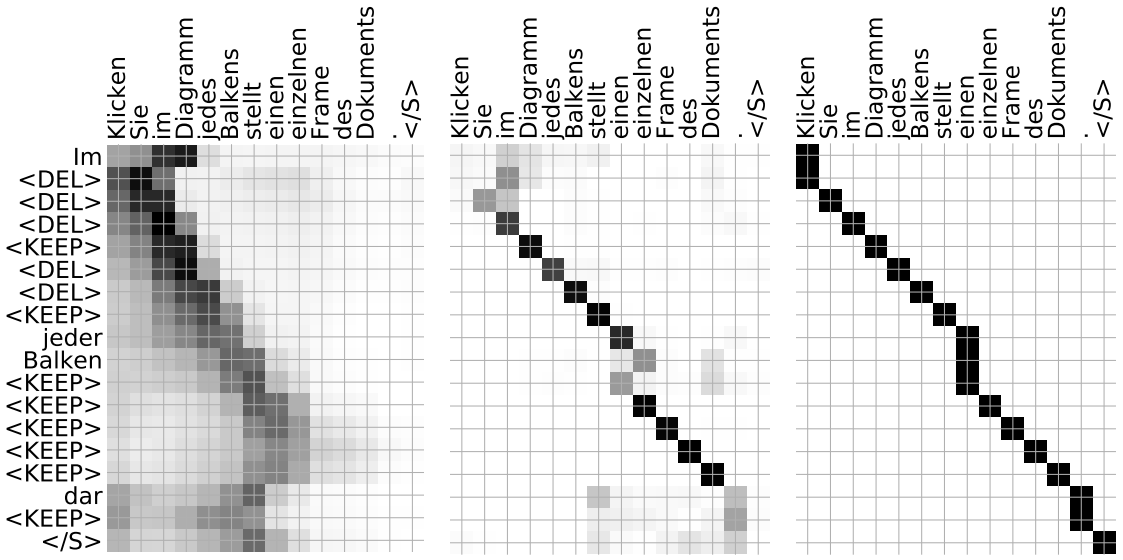
As we have seen in the previous section, one way of handling multi-source inputs is to have multiple encoders, and to combine their representations before giving them as input to the decoder (Junczys-Dowmunt et al. 2017a; Libovický et al. 2016; Zoph et al. 2016a).

To do so, we use two encoders (MT and SRC), and a decoder with two attention heads:

$$c_t^1 = \text{look}^1(s_{t-1}, (h_i)_{i=1}^T) \tag{6.19}$$

$$c_t^2 = \text{look}^2(s_{t-1}, (h'_j)_{j=1}^{T'}) \tag{6.20}$$

$$c_t = c_t^1 \oplus c_t^2 \tag{6.21}$$



(A) Global attention (Small) (B) Global attention (Large) (C) Hard attention

FIGURE 6.4: Alignments performed by global attention models trained with varying amounts of data (Small and Large datasets), and hard attention.

where $(h_i \in \mathbb{R}^{2n})_{i=1}^{T}$ and $(h'_j \in \mathbb{R}^{2n})_{j=1}^{T'}$ are respectively the sequences of hidden states produced by the MT and SRC encoders, and s_{t-1} is the current state of the decoder. The look^2 function is always a global attention: look_{global}. The look^1 function can be either look_{global} or look_{hard}.

Chained Encoders We propose a chained architecture, which combines two encoder-decoder models (see Figure 6.5). A first model SRC $\rightarrow$ MT, with a global attention mechanism, tries to mimic the translation process that resulted in MT (from SRC). The attention vectors of this first model summarize the part of the SRC sequence that led to the generation of each MT token. A second model MT $\rightarrow$ OP learns to post-edit and uses a hard attention over the MT sequence, as well as the attention contexts over SRC computed by the first system (see eq. 6.22). Both models are trained jointly, by optimizing a sum of both losses (see eq. 6.24).

$$c_t = \tanh(Hc'_i + b) + h_i \tag{6.22}$$

$$c'_i = \text{look}_{global}(s'_i, (h'_j)_{j=1}^{T'}) \tag{6.23}$$

where $t \rightarrow i$ is our hard alignment heuristic, which maps the current decoder state and history to a position i in the MT sequence. s'_i is the i$^{\text{th}}$ state of the MT decoder, $(h_j)_{j=1}^{T'}$ is the sequence of hidden states produced by the SRC encoder, and h_i is the i$^{\text{th}}$ hidden state of the MT encoder. $H \in \mathbb{R}^{2n \times 2n}$, and $b \in \mathbb{R}^{2n}$ are trained parameters of the model.

We use the same embedding matrix $E \in \mathbb{R}^{|V| \times m}$ in the MT decoder and in the MT encoder. To make sure that the MT decoder's output sequence and the MT encoder's input sequence are the same, we always force-feed the ground truth symbols to the MT decoder (teacher forcing). This is possible because the MT sequence is also available at test time (contrary to the OP sequence).

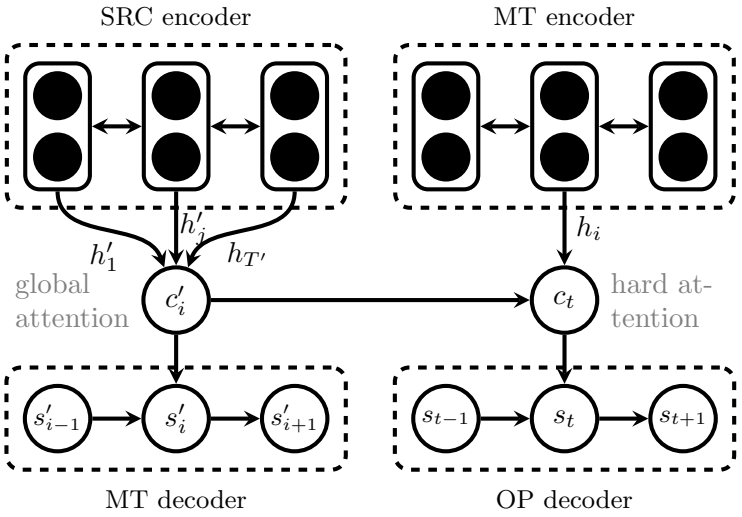


FIGURE 6.5: Illustration of the chained model for automatic post-editing. It has two bidirectional encoders that read the SRC and MT sequences, and two decoders that output MT and OP sequences. We maximize two training objectives: a translation objective (SRC $\rightarrow$ MT) and a post-editing objective (MT $\rightarrow$ OP). The OP decoder uses hard attention with the MT encoder ($t \rightarrow i$), and uses the corresponding global attention context c'_i over SRC.

The training loss that we minimize is a combination of both cross-entropy losses. This allows us to train the translation model and the post-editing model jointly:

$$\mathcal{L} = 0.5 \times \mathcal{L}(\texttt{SRC} \rightarrow \texttt{MT}) + \mathcal{L}(\texttt{MT} \rightarrow \texttt{OP}) \tag{6.24}$$

The gradients are also back-propagated from the OP decoder up to the SRC encoder, through the chained attention mechanism (eq. 6.22).

6.3.3 Experiments

We train three kinds of models: monolingual models with hard attention (which we have shown to be superior to global attention), multi-encoder models, and chained-encoder models. The latter two models use a global attention mechanism for SRC $\rightarrow$ OP (multi) and SRC $\rightarrow$ MT (chained), and hard attention for MT $\rightarrow$ OP.

Each of these architectures is trained in three different data settings: Small, Medium and Large (see Table 6.8). This gives a total of 9 different models. We train 4 instances of each model that we combine in ensembles. Single-model scores are averaged over these 4 runs.

We use the same hyperparameters, training settings and datasets as in Sections 6.2.2 (replication of Libovický et al. (2016)) and 6.3.1 (global attention against hard attention).

The only differences between the Small, Medium and Large settings are the maximum sequence length (see Table 6.8), the total training time, and learning rate decay. For the Small and Medium models, we start with a learning rate of 1.0, and multiply it by 0.95 every epoch starting from the 4$^{\text{th}}$ epoch. For the Large models (where epochs are longer), we decay the learning rate by 0.8 every half epoch starting from the 2$^{\text{nd}}$ epoch.
The Small models are trained for 40k steps, the Medium ones for 60k steps, and the Large ones

Model	Data	Dev TER					Steps
		Greedy	Beam	**Average**	**Ensemble**	Ens + Avg	
Mono (Hard)	Large	23.0	22.8	22.9	22.3	22.4	117k
Chained		22.2	22.0	**21.9**	**21.6**	21.7	172k
Multi		22.4	22.0	22.0	21.9	21.9	132k
Mono (Hard)	Medium	23.1	23.0	23.0	22.9	22.8	20k
Chained		22.6	22.4	**22.4**	**22.0**	22.0	47k
Multi		23.0	22.9	23.0	22.6	22.4	22k
Mono (Hard)	Small	23.5	23.4	23.4	23.2	23.2	14k
Chained		23.4	23.3	**23.2**	**23.1**	23.1	24k
Multi		23.4	23.3	23.2	23.1	23.1	14k

TABLE 6.11: Comparison of different decoding strategies for op-based APE on the $en \rightarrow de$ dev set. The beam search decoder has a beam size of 6, with length normalization. The ensembles contain 4 models. The non-ensemble results are averaged over 4 runs (the same 4 models used for ensembling). The "average" column corresponds to averaging the weights of the 4 best checkpoints (according to greedy TER) of each model. The "steps" column is the average number of SGD steps (over 4 runs) before reaching the best dev performance.

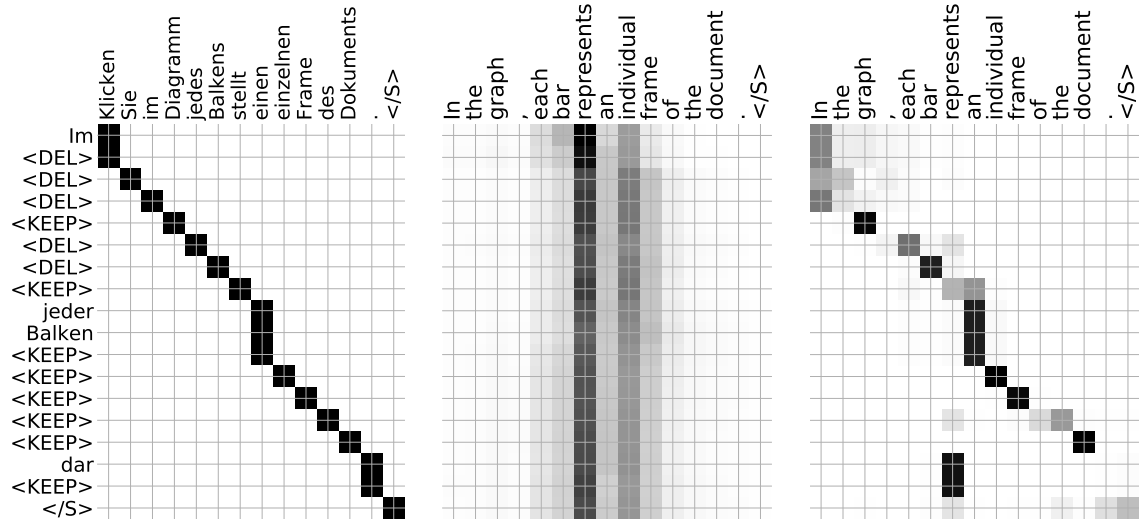
for 200k steps. We evaluate the models on the dev set every 1000 SGD steps, and keep the checkpoints that have the best greedy TER.

Figure 6.6 gives examples of alignments by our multi-source op-based models. We see that the SRC → OP global attention mechanism from the multi-encoder model (Figure 6.6b) is unable to align the source sequence with the output sequence of edit operations. This is not so surprising, as edit operations apply to the MT input only, and it is difficult to link them with SRC symbols without appropriate MT context. The reader would have a hard time doing this alignment herself without looking at the MT sequence. In other words, a SRC → OP alignment makes little sense. Because of this, the SRC encoder probably receives irrelevant feedback during training, and is unable to compute a good representation of the source input sequence.

On the other hand, the SRC → MT attention mechanism from the chained model is able to do a good alignment (Figure 6.6d), which is easier as this is a "basic" machine translation task. Figure 6.6c shows the dot product between the (soft) SRC → MT alignment and the (hard) MT → OP alignment in the chained model. This is how the OP decoder sees the SRC input. This alignment seems more informative and may help the decoder to make relevant lexical choices, based on the original sentence.

Table 6.11 compares the scores of our models on the dev set, with different decoding techniques. Surprisingly, we see that there is a very minor improvement when using beam search, when averaging the weights of several checkpoints, or when using ensembles of models. This is a big difference with translation-based post-editing, where beam search decoding and ensembles give a large boost in post-editing quality. A possible explanation for this is the strong class imbalance between the KEEP operation and the other operations.

Table 6.15 gives statistics about the most common edit operations in the training set, and in the outputs of our ensemble model (medium-chained). We see that the training set contains a vast majority of KEEP symbols (67%). Because of this imbalance, the models are unlikely to generate other symbols than KEEP (especially insertions, whose probability mass is very stretched). When combining several models in an ensemble, or using beam search decoding, output sequences that contain only KEEP symbols are even more likely to be generated. Our chained model generates 92.7% of KEEP symbols (on dev) with greedy decoding, 93.0% with



(A) Hard alignment between MT input and OP output.

(B) Soft alignment between SRC input and OP output in the Multi-encoder model.

(C) Dot product between the SRC → MT (Fig. 6.6d) and MT → OP (Fig. 6.6a) attention matrices in the Chained model.

(D) Soft alignment between SRC input and MT input in the Chained model (translation model).

FIGURE 6.6: Forced alignments (with teacher forcing) performed by our Medium Chained and Multi-Encoder models on the $en \rightarrow de$ dev set.

beam search decoding, and 93.6% with ensemble decoding. Table 6.14 shows the number of sentences that are modified, improved or deteriorated on Test 2017 by our models. We see indeed that the ensembles tend to modify much fewer sentences than the single models.

We also observe from Table 6.11 that training with more data has a less dramatic effect that with translation-based post-editing. We did not train models on the full synthetic corpus, but we suspect that the improvement (if any) would be minor.

Table 6.12 gives the results of our techniques on Test 2016 and Test 2017. We see that all of our models outperform the MT and SPE baselines, even single models in low-resource conditions. Yet, none of our models come even close to the SOTA result of Chatterjee et al. (2017), a translation-based model trained with all the synthetic data available. In general, the chained model is superior to the multi-encoder model, except in the "Large" data setting, where they obtain identical results. The multi-source (multi-encoder and chained) models are superior to the mono-source (hard attention) model, except in the "Small" data setting. 12k segments of data is maybe not enough to train a good `SRC` encoder. Finally, the ensembles bring a minor but consistent improvement over the single models.

Table 6.13 compares our "chained" ensemble models against the translation-based "M-CGRU" models from Section 6.2.1. We see that translation-based post-editing outperforms op-based post-editing in the "Large" and "Medium" settings.[16] The M-CGRU model is able to better take advantage of larger training sets, and gets improved results with beam search decoding and ensembling. In the "Small" data setting, our op-based model outperforms the translation-based ensemble. This suggests that op-based post-editing is a good alternative in low-resource settings.

Op-based post-editing also takes less computing power to train.[17] The "chained" models can be trained on a low-end GPU, and took respectively 2h30,[18] 4h,[19] and 17h[19] to train in the Small, Medium and Large settings.

[16] With single models, our op-based method is superior in the Small and Medium settings. With greedy decoding, it is superior in all three settings.

[17] Also at test time, as we can do away with ensembles and beam search decoding.

[18] On a single GTX 750 Ti

[19] On a single GTX 1070

Model	Data	Dev		Test 2016		Test 2017	
		TER	BLEU	TER	BLEU	TER	BLEU
MT Baseline	–	24.8	62.9	24.8	62.1	24.5	62.5
SPE Baseline	Small/Medium	–	–	24.6	63.5	24.7	63.0
FBK - single	XXL	19.8	70.7	–	–	20.3	69.1
ensemble (8)		19.2	71.9	19.3	70.9	19.6	70.1
Single models							
Mono (Hard)	Large	22.9	66.9	22.8	66.2	23.0	65.5
Chained		**21.9**	68.1	**21.7**	67.3	**22.1**	66.5
Multi		22.0	67.9	21.8	67.2	22.2	66.5
Mono (Hard)	Medium	23.0	66.8	22.7	66.3	23.1	65.2
Chained		**22.4**	67.4	**22.3**	66.6	**22.6**	65.9
Multi		23.0	66.9	22.6	66.4	23.0	65.4
Mono (Hard)	Small	23.4	66.3	23.3	65.5	**23.4**	64.7
Chained		**23.2**	66.4	**23.2**	65.6	**23.4**	64.7
Multi		**23.2**	66.4	**23.2**	65.5	23.5	64.6
Ensembles of 4							
Mono (Hard)	Large	22.3	67.3	22.4	66.6	22.5	66.0
Chained		**21.6**	68.5	**21.4**	67.6	21.8	66.9
Multi		**21.6**	68.4	**21.4**	67.7	**21.7**	67.1
Mono (Hard)	Medium	22.9	66.8	22.5	66.5	22.8	65.5
Chained		**22.0**	67.9	**22.0**	66.9	**22.3**	66.2
Multi		22.6	67.2	22.3	66.6	22.5	65.9
Mono (Hard)	Small	23.2	66.5	23.1	65.9	23.2	65.0
Chained		**23.1**	66.6	23.0	65.8	23.2	64.8
Multi		**23.1**	66.7	**22.9**	65.9	**23.1**	65.0

TABLE 6.12: Scores of different op-based models on the WMT16 and WMT17 $en \rightarrow de$ APE tasks. The scores are obtained with a beam search decoder of size 6. Our non-ensemble results are averaged over 4 different training runs. The SPE Baseline is a monolingual PBMT model (Simard et al. 2007), and the MT Baseline corresponds to keeping the MT hypothesis as it is. The FBK baseline is the winner of the 2017 APE Task (Chatterjee et al. 2017).

Model	Data	Test 2016		Test 2017	
		TER	BLEU	TER	BLEU
M-CGRU	Large	**20.9**	68.0	**21.3**	67.2
Chained		21.4	67.6	21.8	66.9
M-CGRU	Medium	**21.6**	67.0	**21.9**	66.0
Chained		22.0	66.9	22.3	66.2
M-CGRU	Small	23.7	64.0	23.8	63.4
Chained		**23.0**	65.8	**23.2**	64.8

TABLE 6.13: Comparison between the translation-based "M-CGRU" models (see Section 6.2.1) and the op-based "Chained" models. All these results are obtained with ensembles of 4 models and beam search decoding.

Model		Modified	Improved	Deteriorated	Precision
FBK	Ensemble (8)	1607	1035	334	75.6%
Large	Ensemble (4)	1146	744	189	79.7%
Medium		972	645	138	82.4%
Small		628	408	89	82.1%
Large	Average (4)	1538	748	244	75.4%
Medium		1094	652	213	75.4%
Small		777	433	169	71.9%

TABLE 6.14: Number of modified and improved/deteriorated sentences on Test 2017, by our "chained" APE models. The full corpus has 2000 sentences. The FBK baseline is the winner of the 2017 APE Task (Chatterjee et al. 2017).

Token	Count	Percentage
KEEP	326581	66.93
DEL	76725	15.72
"	5170	1.06
,	3249	0.67
die	2461	0.50
der	1912	0.39
zu	1877	0.38
werden	1246	0.26
Sie	1209	0.25
den	1195	0.24

Token	Count	Percentage
<KEEP>	18745	93.56
<DEL>	545	2.72
"	197	0.98
>	130	0.65
,	72	0.36
zu	48	0.24
wird	21	0.10
an	21	0.10
werden	20	0.10
.	12	0.06

Deletions	Count	Percentage
,	3857	0.79
die	3010	0.62
Sie	2797	0.57
"	1848	0.38
der	1697	0.35

Deletions	Count	Percentage
Sie	52	0.26
,	52	0.26
werden	33	0.16
de	24	0.12
wird	24	0.12

TABLE 6.15: Top 10 edit ops in the target side of the training set for $en \rightarrow de$ (left), and most generated edit ops by our ensemble of medium-chained models on the dev set (right). The bottom tables show the 5 most deleted symbols (target of the DEL op) in the training set (left), and in the output of our system (right).

Conclusion

This thesis focused on two broad objectives: review and replication of reported results from the literature in Neural Machine Translation (NMT); and applying NMT techniques to three related tasks: Machine Translation, Speech Translation, and Automatic Post-Editing.

Research Replication

Our replication work led to the implementation of two large libraries: MultiVec, and Seq2seq.

MultiVec implements several techniques for computing vector representations for text: Mikolov et al. (2013a)'s CBOW and Skip-Gram models, Luong et al. (2015a)'s model for bilingual embeddings, and Q. V. Le et al. (2014)'s Paragraph Vector. MultiVec is blazingly fast (as fast as Word2vec) and its modular C++ code is aimed at being easily understood and extended. It allows easy and efficient training of models, thanks to a command-line interface and a Python wrapper. Furthermore, it exposes several functions in its API for manipulating trained models (e.g., to continue training a model, save it to another format, compute a distance between words or sequences, etc.) We also distribute pre-processing and benchmarking scripts.

Seq2seq is an improvement of TensorFlow's "seq2seq" NMT example. We replicated the exact attention-based model from Bahdanau et al. (2015), as well as improvements from Jean et al. (2015a), Luong et al. (2015b), and Sennrich et al. (2017, 2016c). It also implements techniques for Automatic Speech Recognition (Chan et al. 2016), and Automatic Post-Editing (Junczys-Dowmunt et al. 2017a; Libovický et al. 2016), as well as our architectures for AST and APE. Seq2seq supports beam search decoding, ensemble decoding and multi-task training. It facilitates experimentation and research replication thanks to a system of configuration files and detailed logging.

In addition to implementing the aforementioned techniques, we performed a number of experiments to validate our implementations, and replicate reported results from the literature. We share the trained models and the pre-processing scripts, as well as the configuration files for retraining these models.

In particular, we trained models on the WMT 2014 task of English→French translation of news. We were able to replicate pioneer work from Bahdanau et al. (2015) and Jean et al. (2015a). We also replicated results on the WMT 2016 Automatic Post-Editing task: a large "translation-based" model by Junczys-Dowmunt et al. (2017a) which is trained on synthetic data; and a smaller "op-based" model by Libovický et al. (2016).

Original Results

As part of our NMT framework, we developed architectures that allowed us to present original contributions in three related tasks: Machine Translation, End-to-End Speech Translation, and Automatic Post-Editing.

Machine Translation We worked on three Machine Translation corpora of varying size:

- WMT 2014, for English to French translation of news (12M segments). In addition to replicating existing results, we also trained subword-level models, which achieve competitive results with Jean et al. (2015a)'s large-vocabulary approach with unknown-word replacement. While this technique is not new, we provide a useful baseline on this corpus (which is a popular benchmark in NMT). We also compared the effects of using Adam versus AdaDelta for training.
- IWSLT 2014, German to English translation of TED talks (150k segments). We applied existing techniques for NMT that had not yet been applied to this task. By doing so, we were able to obtain state-of-the-art results on this corpus, outperforming by 4 BLEU points the previous best NMT result (by 2 points when not using ensembles) and by 6 points a strong SMT baseline. We were able to show that, contrary to Koehn et al. (2017)'s claims, NMT is also a strong alternative in medium-resource settings.
- BTEC, French to English translation of basic travel expressions (20k segments). This task is easy (very short and simple sentences), but the corpus is very small by MT standards. By doing an extensive architecture search, we were able to outperform the SMT baseline by a large margin (3 BLEU points).

Speech Translation We modified existing attention-based architectures for ASR (Chan et al. 2016), and applied them to Speech Translation. We presented the first results in the literature for End-to-End AST. While our initial work was on a synthetic corpus (built by using TTS on the French-side of BTEC), we later extended our work on a real corpus of audiobooks. Augmented LibriSpeech (Kocabiyikoglu et al. 2018) is a new resource that was built by aligning the LibriSpeech ASR dataset (containing English-language audiobooks and their transcriptions), with French translations of the same books (extracted from Project Gutenberg). We designed AST models that we trained on this dataset, and experimented with different Multi-Task training strategies. We presented the first AST, MT and ASR results on this dataset.

Automatic Post-Editing For this specific problem, we worked in the context of the WMT 2016 and 2017 tasks. We extended the work of Junczys-Dowmunt et al. (2017a), by training similar translation-based models in various data conditions. We showed that these models, even though they work best when trained with large amounts of (synthetic) data, are also viable in lower-resource settings. Then, we extended the work of Libovický et al. (2016), by proposing improved architectures for their op-based post-editing technique. We obtained a large increase in scores by constraining the attention mechanism to look at the exact symbols that are being post-edited. We also proposed a "chained-encoder" architecture that makes a better use of the source-language sequence than the usual multi-encoder architectures. We compared this type of op-based model against the state-of-the-art translation-based methods, and showed that while it is inferior in medium to high-resource settings, it presents a good alternative in low-resource scenarios.

Future work

In this section, we present ideas for future work (in the short-term or long-term), in the three tasks that we explored.

Machine Translation In the short-term, we would like to try the Transformer model (Vaswani et al. 2017) on low-resource settings (and other tasks). Current results with this model are outstanding, but in high-resource conditions only. We would also like to try NMT in more low-resource scenarios, and try to disprove Koehn et al. (2017)'s assumptions that NMT is worse than SMT in these settings.[20]

In the long-term, we think that NMT will take the following trends: low-resource or zero-resource NMT using unsupervised methods and transfer learning (maybe with multilingual MT); multilingual NMT (it is so much more convenient to have a single model for all language pairs); character-level machine translation (current BPE units work fine but are completely arbitrary); and multi-task learning.

A fun research idea would be to try to train NMT models by using images of text as input (e.g., scanned text or just typeset text in several fonts). This would be closer to how humans read text, and would present an interesting challenge. Also, this presents nearly infinite possibilities for data augmentation (different fonts, rotations, cropping, etc.)

Speech Translation In the short-term, we should focus on reducing the cold-start problem that we observed when training attention-based ASR and AST models. We have several ideas in this direction, like constraining the attention model to be close to monotonous (at least at the beginning of training), or giving it ways to learn a monotonous alignment more easily (e.g., with positional information). Once this problem solved, we will be able to train bigger models, which may obtain considerably better results. Other short-term goals are to train stronger cascaded baselines on Synthetic BTEC and Augmented LibriSpeech. We also should train AST models that do not use the Google Translate references (but only the automatically extracted alignments). On the other hand, we could make use of the noisier data from the Augmented LibriSpeech ("other" dataset).

In the longer term, we think that a corpus of TED talks aligned with their translated subtitles would constitute an interesting resource, even more so if it is aligned with the video. We also think that more Multi-Task strategies should be tried out. We could add auxiliary tasks, like language modeling, lip reading (from TED videos), or MT from other languages. The encoder and/or decoder could be trained (with pre-training and/or multi-task training) using data from these tasks, and not only our AST corpus. For instance, when doing AST from English to French, we could train the English acoustic model on English ASR data (e.g., the larger LibriSpeech), and the French decoder with parallel data (MT) or monolingual data (language modeling). Potentially larger amounts of parallel data can be extracted from Project Gutenberg if we do not align with LibriSpeech. If we are in a low-resource scenario, with an unwritten language as input, and a high-resource language as target (e.g., French), we could train the decoder on an auxiliary MT task where French is the target language (e.g., English to French).

[20] This is actually hard to "prove", as NMT is very unconstrained, and new architectures or hyperparameter values can very well void previous (negative) observations. Koehn et al. (2017)'s models can be overfitting for all we know.

Automatic Post-Editing We suspect that APE will become less useful as we get stronger MT models, and better techniques for low-resource NMT and domain adaptation. As a possible direction, we would like to explore character-level models for Neural APE (op-based or neural-based), using NMT models like Lee et al. (2016)'s.[21]

Reinforcement Learning can be used to optimize APE models on the evaluation metrics directly (TER or BLEU), using the REINFORCE algorithm (Ranzato et al. 2016) or Actor-Critic methods (Bahdanau et al. 2017). Our attempts at applying REINFORCE to op-based models did not yield any improvement, but it might be more successful with translation-based APE.

An interesting short-term experiment would be to train an APE model with its own outputs. This could be used as a technique for artificial data augmentation, which could help with regularization, and maybe improve multiple-pass post-editing.

We also think that other forms of synthetic data should be explored. The synthetic corpus produced by Junczys-Dowmunt et al. (2016a) is a very useful resource, and helped obtain the best results to date on this task, however we think that this is not a very realistic APE scenario. The corpus is domain specific (and cannot be used in other APE tasks), and was obtained thanks to large amounts of monolingual and parallel data, not available in many language pairs. It is possible that the same models could achieve excellent results with less "expensive" synthetic data: like monolingual data with artificial noise, or simulated PE (Potet et al. 2012a).

Finally, we would like to test these APE techniques on other public (potentially more challenging) datasets, like (Potet et al. 2012b)'s French-English dataset, and the data from the WMT 2015 edition of the APE task (where no one was able to beat the MT baseline).

[21] Varis et al. 2017 already explored this setting, but their results on the 12k corpus are not very good, and could probably be improved a lot: their dev BLEU score by a BPE-based model is 42.4 (we get 61.7).

Bibliography

Abadi, Martin, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Łukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Rajat Monga, Sherry Moore, Derek Murray, Jon Shlens, Benoit Steiner, Ilya Sutskever, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Oriol Vinyals, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (2015). "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." In: *arXiv*.

Artetxe, Mikel, Gorka Labaka, and Eneko Agirre (2017). "Learning Bilingual Word Embeddings with (Almost) no Bilingual Data." In: *ACL*, pp. 451–462.

Artetxe, Mikel, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho (2018). "Unsupervised Neural Machine Translation." In: *ICLR*.

Axelrod, Amittai, Xiaodong He, and Jianfeng Gao (2011). "Domain Adaptation via Pseudo In-Domain Data Selection." In: *Computational Linguistics* 23.6, pp. 355–362.

Aziz, Wilker, Sheila Castilho, and Lucia Specia (2012). "PET: a Tool for Post-editing and Assessing Machine Translation." In: *LREC*.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton (2016). "Layer Normalization." In: *arXiv*.

Bahdanau, Dzmitry, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio (2017). "An Actor-Critic Algorithm for Sequence Prediction." In: *ICLR*.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). "Neural Machine Translation by Jointly Learning to Align and Translate." In: *ICLR*. San Diego, California, USA, pp. 3104–3112.

Bahdanau, Dzmitry, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio (2016). "End-to-End Attention-based Large Vocabulary Speech Recognition." In: *ICASSP*, pp. 4945–4949.

Banerjee, Satanjeev and Alon Lavie (2005). "METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments." In: *ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Vol. 29, pp. 65–72.

Baroni, Marco, Georgiana Dinu, and Germán Kruszewski (2014). "Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors." In: *ACL*, pp. 238–247.

Béchara, Hanna, Yanjun Ma, and Josef van Genabith (2011). "Statistical Post-Editing for a Statistical MT System." In: *MT Summit XIII*, pp. 308–315.

Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer (2015). "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks." In: *NIPS*.

Bengio, Yoshua (2012). "Practical Recommendations for Gradient-Based Training of Deep Architectures." In: *Neural Networks: Tricks of the Trade*.

Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Janvin (2003). "A Neural Probabilistic Language Model." In: *Journal of Machine Learning Research* 3, pp. 1137–1155.

Bérard, Alexandre, Laurent Besacier, Ali Can Kocabiyikoglu, and Olivier Pietquin (2018). "End-to-End Automatic Speech Translation of Audiobooks." In: *ICASSP*.

Bérard, Alexandre, Olivier Pietquin, and Laurent Besacier (2017). "LIG-CRIStAL System for the WMT17 Automatic Post-Editing Task." In: *WMT - Shared Task Papers*.

Bérard, Alexandre, Olivier Pietquin, Laurent Besacier, and Christophe Servan (2016a). "Listen and Translate: A Proof of Concept for End-to-End Speech-to-Text Translation." In: *NIPS End-to-end Learning for Speech and Audio Processing Workshop*.

Bérard, Alexandre, Christophe Servan, Olivier Pietquin, and Laurent Besacier (2016b). "MultiVec: a Multilingual and Multilevel Representation Learning Toolkit for NLP." In: *LREC*.

Bojar, Ondřej, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Shujian Huang, Matthias Huck, Philipp Koehn, Qun Liu, and Varvara Logacheva (2017). "Findings of the 2017 Conference on Machine Translation (WMT17)." In: *WMT - Shared Task Papers*.

Bojar, Ondřej, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, and Christof Monz (2016). "Findings of the 2016 Conference on Machine Translation (WMT16)." In: *WMT - Shared Task Papers*.

Bojar, Ondřej, Rajen Chatterjee, Christian Federmann, Barry Haddow, Matthias Huck, Chris Hokamp, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Matt Post, Carolina Scarton, Lucia Specia, and Marco Turchi (2015). "Findings of the 2015 Workshop on Statistical Machine Translation (WMT15)." In: *WMT - Shared Task Papers*.

Brown, Peter F, Stephen A Della Pietra, Vincent J Della Pietra, and Robert L Mercer (1993). "The Mathematics of Statistical Machine Translation: Parameter Estimation." In: *Computational Linguistics* 19.2, pp. 263–311.

Brown, Peter F, Vincent J Della Pietra, Robert L Mercer, Stephen A Della Pietra, and Jennifer C Lai (1992). "An Estimate of an Upper Bound for the Entropy of English." In: *Computational Linguistics* 18.

Cettolo, Mauro, Christian Girardi, and Marcello Federico (2012). "Web Inventory of Transcribed and Translated Talks." In: *EAMT*.

Chan, William, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals (2016). "Listen, Attend and Spell." In: *ICASSP*.

Chatterjee, Rajen, M Amin Farajian, Matteo Negri, Marco Turchi, Ankit Srivastava, and Santanu Pal (2017). "Multi-source Neural Automatic Post-Editing: FBK's participation in the WMT 2017 APE shared task." In: *WMT - Shared Task Papers*. Vol. 2, pp. 630–638.

Chatterjee, Rajen, José G de Souza, Matteo Negri, and Marco Turchi (2016). "The FBK Participation in the WMT 2016 Automatic Post-Editing Shared Task." In: *WMT - Shared Task Papers*. Berlin, Germany: Association for Computational Linguistics, pp. 745–750.

Chatterjee, Rajen, Marco Turchi, and Matteo Negri (2015a). "The FBK Participation in the WMT15 Automatic Post-editing Shared Task." In: *WMT - Shared Task Papers*, pp. 210–215.

Chatterjee, Rajen, Marion Weller, Matteo Negri, and Marco Turchi (2015b). "Exploring the Planet of the APEs: a Comparative Study of State-of-the-art Methods for MT Automatic Post-Editing." In: *ACL* 3, pp. 156–161.

Cho, Kyunghyun, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio (2014a). "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches." In: *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pp. 103–111.

Cho, Kyunghyun, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (2014b). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." In: *EMNLP*, pp. 1724–1734.

Chorowski, Jan, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio (2015). "Attention-Based Models for Speech Recognition." In: *NIPS*, pp. 577–585.

Chorowski, Jan and Navdeep Jaitly (2016). "Towards better decoding and language model integration in sequence to sequence models." In: *arXiv*.

Cohn, Trevor, Cong Duy Vu Hoang, Ekaterina Vymolova, Kaisheng Yao, Chris Dyer, and Gholamreza Haffari (2016). "Incorporating Structural Alignment Biases into an Attentional Neural Translation Model." In: *NAACL-HLT*. Denver, Colorado, USA, pp. 876–885.

Collobert, Ronan, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa (2011). "Natural Language Processing (Almost) from Scratch." In: *Journal of Machine Learning Research* 12, pp. 2493–2537.

Conneau, Alexis, Douwe Kiela, Holger Schwenk, Loc Barrault, and Antoine Bordes (2017a). "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data." In: *EMNLP*.

Conneau, Alexis, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou (2017b). "Word Translation Without Parallel Data." In: *arXiv*.

Crego, Josep, Jungi Kim, Guillaume Klein, Anabel Rebollo, Kathy Yang, Jean Senellart, Egor Akhanov, Patrice Brunelle, Aurelien Coquard, Yongchao Deng, Satoshi Enoue, Chiyo Geiss, Joshua Johanson, Ardas Khalsa, Raoum Khiari, Byeongil Ko, Catherine Kobus, Jean Lorieux, Leidiana Martins, Dang-Chuan Nguyen, Alexandra Priori, Thomas Riccardi, Natalia Segal, Christophe Servan, Cyril Tiquet, Bo Wang, Jin Yang, Dakun Zhang, Jing Zhou, and Peter Zoldan (2016). "SYSTRAN's Pure Neural Machine Translation Systems." In: *arXiv*.

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." In: *Journal of Machine Learning Research* 12, pp. 2121–2159.

Duong, Long, Antonios Anastasopoulos, David Chiang, Steven Bird, and Trevor Cohn (2016). "An Attentional Model for Speech Translation Without Transcription." In: *NAACL-HLT*. 1, pp. 949–959.

Durrani, Nadir, Barry Haddow, Philipp Koehn, and Kenneth Heafield (2014). "Edinburgh's Phrase-based Machine Translation Systems for WMT-14." In: *WMT*, pp. 97–104.

Dyer, Chris, Victor Chahuneau, and Noah A. Smith (2013). "A Simple, Fast, and Effective Reparameterization of IBM Model 2." In: *NAACL*.

Elman, Jeffrey L (1990). "Finding Structure in Time." In: *Cognitive Science* 14.2, pp. 179–211.

Ferrero, Jérémy, Frédéric Agnes, Laurent Besacier, and Didier Schwab (2017). "Using Word Embeddings for Cross-Language Plagiarism Detection." In: *EACL*.

Ferrero, Jérémy, Frédéric Agnès, Laurent Besacier, and Didier Schwab (2016). "A Multilingual, Multi-Style and Multi-Granularity Dataset for Cross-Language Textual Similarity Detection." In: *LREC*.

Firat, Orhan, Kyunghyun Cho, Baskaran Sankaran, Fatos T. Yarman Vural, and Yoshua Bengio (2017). "Multi-way, multilingual neural machine translation." In: *Computer Speech & Language*.

Gal, Yarin and Zoubin Ghahramani (2016). "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks." In: *NIPS*.

Gale, William A. and Kenneth W. Church (1993). "A Program for Aligning Sentences in Bilingual Corpora." In: *Computational Linguistics*.

Gehring, Jonas, Michael Auli, David Grangier, and Yann N Dauphin (2017a). "A Convolutional Encoder Model for Neural Machine Translation." In: *arXiv*.

Gehring, Jonas, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin (2017b). "Convolutional Sequence to Sequence Learning." In: *arXiv*.

Gers, Felix, Jürgen Schmidhuber, and Fred Cummins (1999). "Learning to Forget: Continual Prediction with LSTM." In: *Neural Computation* 12, pp. 2451–2471.

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Vol. 9, pp. 249–256.

Godard, Pierre, Marcely Zanon Boito, Lucas Ondel, Alexandre Bérard, François Yvon, Aline Villavicencio, and Laurent Besacier (2018). "Unsupervised Word Segmentation from Speech with Attention." In: *Interspeech*.

Gouws, Stephan, Yoshua Bengio, and Greg Corrado (2015). "BilBOWA: Fast Bilingual Distributed Representations without Word Alignments." In: *ICML*.

Goyal, Anirudh (2016). "Professor Forcing: A New Algorithm for Training Recurrent Networks." In: *NIPS*.

Green, Spence, Jeffrey Heer, and Christopher D. Manning (2013). "The Efficacy of Human Post-Editing for Language Translation." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

Gulcehre, Caglar, Orhan Firat, Kelvin Xu, Kyunghyun Cho, Loic Barrault, Huei-Chi Lin, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (2015). "On Using Monolingual Corpora in Neural Machine Translation." In: *arXiv*.

Hardt, Daniel and Jakob Elming (2010). "Incremental Re-training for Post-editing SMT." In: *AMTA*.

Hasler, Eva, Barry Haddow, and Philipp Koehn (2011). "Margin Infused Relaxed Algorithm for Moses." In: *The Prague Bulletin of Mathematical Linguistics* 96, pp. 69–78.

Hassan, Hany, Anthony Aue, Chang Chen, Vishal Chowdhary, Jonathan Clark, Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, William Lewis, Mu Li, et al. (2018). "Achieving Human Parity on Automatic Chinese to English News Translation." In: *arXiv*.

Haykin, Simon (1994). *Neural Networks: a Comprehensive Foundation*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In: *Proceedings of the IEEE International Conference on Computer Vision*.

Hill, Felix, Kyunghyun Cho, and Anna Korhonen (2016). "Learning Distributed Representations of Sentences from Unlabelled Data." In: *NAACL-HLT*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory." In: *Neural Computation* 9.8, pp. 1735–1780.

Hokamp, Chris (2017). "Ensembling Factored Neural Machine Translation Models for Automatic Post-Editing and Quality Estimation." In: *WMT - Shared Task Papers*.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer Feedforward Networks are Universal Approximators." In: *Neural Networks* 2, pp. 359–366.

Huang, Po-Sen, Chong Wang, Dengyong Zhou, and Li Deng (2018). "Towards Neural Phrase-based Machine Translation." In: *ICLR*.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *ICML*.

Isabelle, Pierre, Colin Cherry, and George Foster (2017). "A Challenge Set Approach to Evaluating Machine Translation." In: *EMNLP*.

Jean, Sébastien, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio (2015a). "On Using Very Large Target Vocabulary for Neural Machine Translation." In: *NAACL-HLT*.

Jean, Sébastien, Orhan Firat, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio (2015b). "Montreal Neural Machine Translation Systems for WMT15." In: *WMT*, pp. 134–140.

Johnson, Melvin, Mike Schuster, Quoc V Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean (2016). "Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation." In: *arXiv*.

Junczys-Dowmunt, Marcin, Tomasz Dwojak, and Hieu Hoang (2016a). "Is Neural Machine Translation Ready for Deployment? A Case Study on 30 Translation Directions." In: *IWSLT*.

Junczys-Dowmunt, Marcin and Roman Grundkiewicz (2016b). "Log-linear Combinations of Monolingual and Bilingual Neural Machine Translation Models for Automatic Post-Editing." In: *WMT - Shared Task Papers*. Vol. 2. Berlin, Germany: Association for Computational Linguistics, pp. 751–758.

Junczys-Dowmunt, Marcin and Roman Grundkiewicz (2017a). "An Exploration of Neural Sequence-to-Sequence Architectures for Automatic Post-Editing." In: *arXiv*.

Junczys-Dowmunt, Marcin and Roman Grundkiewicz (2017b). "The AMU-UEdin Submission to the WMT 2017 Shared Task on Automatic Post-Editing." In: *WMT - Shared Task Papers*. Vol. 2, pp. 639–646.

Kalchbrenner, Nal, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu (2016). "Neural Machine Translation in Linear Time." In: *arXiv*.

Keskar, Nitish Shirish, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang (2017). "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." In: *ICLR*.

Kingma, Diederik and Jimmy Ba (2015). "Adam: A method for stochastic optimization." In: *ICLR*.

Kiros, Ryan, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler (2015). "Skip-Thought Vectors." In: *NIPS*. 786.

Klementiev, Alexandre, Ivan Titov, and Binod Bhattarai (2012). "Inducing Crosslingual Distributed Representations of Words." In: *COLING*. December, pp. 1459–1474.

Kocabiyikoglu, Ali Can, Laurent Besacier, and Olivier Kraif (2018). "Augmenting Librispeech with French Translations: A Multimodal Corpus for Direct Speech Translation Evaluation." In: *LREC*.

Koehn, Philipp (2010). *MOSES, Statistical Machine Translation System, User Manual and Code Guide*. Tech. rep., p. 245.

Koehn, Philipp and Rebecca Knowles (2017). "Six Challenges for Neural Machine Translation." In: *ACL Workshop on Neural Machine Translation*.

Koehn, Philipp, Franz Josef Och, and Daniel Marcu (2003). "Statistical phrase-based translation." In: *NAACL-HLT*, pp. 48–54.

Kumar, Gaurav, Graeme Blackwood, Jan Trmal, Daniel Povey, and Sanjeev Khudanpur (2015). "A Coarse-Grained Model for Optimal Coupling of ASR and SMT Systems for Speech Translation." In: *EMNLP*, pp. 1902–1907.

Lample, Guillaume, Ludovic Denoyer, and Marc'Aurelio Ranzato (2017). "Unsupervised Machine Translation Using Monolingual Corpora Only." In: *arXiv*.

Le, Ngoc-Tien, Christophe Servan, Benjamin Lecouteux, and Laurent Besacier (2016). "Better Evaluation of ASR in Speech Translation Context Using Word Embeddings." In: *Interspeech*.

Le, Quoc V. and Tomas Mikolov (2014). "Distributed Representations of Sentences and Documents." In: *ICML*.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning." In: *Nature* 521.7553, pp. 436–444.

LeCun, Yann, Léon Bottou, Genevieve B. Orr, and Klaus Robert Müller (2012). "Efficient backprop." In: *Neural Networks: Tricks of the Trade*.

Lee, Jason, Kyunghyun Cho, and Thomas Hofmann (2016). "Fully Character-Level Neural Machine Translation without Explicit Segmentation." In: *ACL*, pp. 1693–1703.

Levy, Omer and Yoav Goldberg (2014). "Linguistic Regularities in Sparse and Explicit Word Representations." In: *CoNLL*, pp. 171–180.

Levy, Omer, Yoav Goldberg, and Ido Dagan (2015). "Improving Distributional Similarity with Lessons Learned from Word Embeddings." In: *Transactions of the Association for Computational Linguistics* 3, pp. 211–225.

Libovický, Jindřich, Jindřich Helcl, Marek Tlustý, Pavel Pecina, and Ondřej Bojar (2016). "CUNI System for WMT16 Automatic Post-Editing and Multimodal Translation Tasks." In: *WMT - Shared Task Papers*. Vol. 2. 2014, pp. 646–654.

Luong, Minh-Thang, Quoc V Le, Ilya Sutskever, Oriol Vinyals, and Łukasz Kaiser (2016). "Multi-task Sequence to Sequence Learning." In: *ICLR*. San Juan, Puerto Rico.

Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning (2015a). "Bilingual Word Representations with Monolingual Quality in Mind." In: *NAACL Workshop on Vector Modeling for NLP*, pp. 151–159.

Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning (2015b). "Effective Approaches to Attention-based Neural Machine Translation." In: *EMNLP*.

Luong, Minh-Thang, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba (2015c). "Addressing the Rare Word Problem in Neural Machine Translation." In: *ACL*.

Mathieu, Benoit, Slim Essid, Thomas Fillon, Jacques Prado, and Gaël Richard (2010). "YAAFE, an Easy to Use and Efficient Audio Feature Extraction Software." In: *ISMIR (International Society of Music Information Retrieval)*.

Mesnil, Grégoire, Marc'Aurelio Ranzato, Tomas Mikolov, and Yoshua Bengio (2014). "Ensemble of Generative and Discriminative Techniques for Sentiment Analysis of Movie Reviews." In: *arXiv*.

Mikolov, Tomas, Greg Corrado, Kai Chen, and Jeffrey Dean (2013a). "Efficient Estimation of Word Representations in Vector Space." In: *ICLR*.

Mikolov, Tomas, Quoc V. Le, and Ilya Sutskever (2013b). "Exploiting Similarities among Languages for Machine Translation." In: *arXiv*.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean (2013c). "Distributed Representations of Words and Phrases and their Compositionality." In: *NIPS*, pp. 3111–3119.

Mikolov, Tomas, Wen-tau Yih, and Geoffrey Zweig (2013d). "Linguistic regularities in continuous space word representations." In: *NAACL-HLT*. June, pp. 746–751.

Mitchell, Tom M (1997). *Machine learning*.

Moore, Robert C (2002). "Fast and Accurate Sentence Alignment of Bilingual Corpora." In: *AMTA*.

Negri, Matteo, Marco Turchi, Rajen Chatterjee, and Nicola Bertoldi (2018). "eSCAPE: a Large-scale Synthetic Corpus for Automatic Post-Editing." In: *arXiv*.

Ng, Andrew (2017). *Coursera - Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization*.

Och, Franz Josef (2003). "Minimum Error Rate Training in Statistical Machine Translation." In: *ACL*. Vol. 1001. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 160–167.

Och, Franz Josef and Hermann Ney (2003). "A Systematic Comparison of Various Statistical Alignment Models." In: *Computational Linguistics* 29.1, pp. 19–51.

Östling, Robert and Jörg Tiedemann (2017). "Neural machine translation for low-resource languages." In: *arXiv*.

Ott, Myle, Sergey Edunov, David Grangier, and Auli Michael (2018). "Scaling Neural Machine Translation." In: *arXiv*.

Pal, Santanu, Sudip Kumar Naskar, Mihaela Vela, and Josef van Genabith (2016). "A Neural Network Based Approach to Automatic Post-Editing." In: *ACL*. Vol. 2, pp. 281–286.

Pal, Santanu, Mihaela Vela, Sudip Kumar Naskar, and Josef van Genabith (2015). "USAAR-SAPE: An English-Spanish Statistical Automatic Post-Editing System." In: *WMT*, pp. 216–221.

Panayotov, Vassil, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur (2015). "LibriSpeech: an ASR Corpus Based on Public Domain Audio Books." In: *ICASSP*.

Papineni, Kishore, Salim Roukos, Todd Ward, and Wj Zhu (2002). "BLEU: a method for automatic evaluation of machine translation." In: *ACL*. July, pp. 311–318.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). "On the difficulty of training Recurrent Neural Networks." In: *ICML*.

Pennington, Jeffrey, Richard Socher, and Christopher Manning (2014). "Glove: Global Vectors for Word Representation." In: *EMNLP*. Vol. 12, pp. 1532–1543.

Post, Matt, Gaurav Kumar, Adam Lopez, Damianos Karakos, Chris Callison-Burch, and Sanjeev Khudanpur (2013). "Improved Speech-to-Text Translation with the Fisher and Callhome Spanish-English Speech Translation Corpus." In: *IWSLT*.

Potet, Marion, Laurent Besacier, and Hervé Blanchon (2010). "The LIG Machine Translation System for WMT 2010." In: *ACL Workshop on Statistical Machine Translation and Metrics*. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 161–166.

Potet, Marion, Laurent Besacier, Hervé Blanchon, and Marwen Azouzi (2012a). "Towards a Better Understanding of Statistical Post-Edition Usefulness." In: *IWSLT*, pp. 284–291.

Potet, Marion, Emmanuelle Esperança-Rodier, Laurent Besacier, and Hervé Blanchon (2012b). "Collection of a Large Database of French-English SMT Output Corrections." In: *LREC*. Istanbul, Turkey: European Language Resources Association (ELRA), pp. 4043–4048.

Press, Ofir and Lior Wolf (2017). "Using the Output Embedding to Improve Language Models." In: *EACL*, pp. 157–163.

Ranzato, Marc'Aurelio, Sumit Chopra, Michael Auli, and Wojciech Zaremba (2016). "Sequence Level Training with Recurrent Neural Networks." In: *ICLR*.

Rosenblatt, Frank (1957). *The Perceptron, a Perceiving and Recognizing Automaton (Project Para)*.

Rousseau, Anthony, Paul Deléglise, and Yannick Estève (2014). "Enhancing the TED-LIUM Corpus with Selected Data for Language Modeling and More TED Talks." In: *LREC*.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1988). "Learning Representations by Back-Propagating Errors." In: *Cognitive modeling*.

Schmidhuber, Jürgen (2015). "Deep learning in Neural Networks: An Overview." In: *Neural Networks* 61, pp. 85–117.

Schwarz, Diemo (2007). "Corpus-Based Concatenative Synthesis." In: *IEEE Signal Processing Magazine* 24.2, pp. 92–104.

Schwenk, Holger (2012). "Continuous Space Translation Models for Phrase-Based Statistical Machine Translation." In: *COLING*, pp. 1071–1080.

Sennrich, Rico, Orhan Firat, Kyunghyun Cho, Alexandra Birch, Barry Haddow, Julian Hitschler, Marcin Junczys-Dowmunt, Samuel Laeubli, Antonio Valerio, Antonio Valerio Miceli Barone, Jozef Mokry, and Maria Nadejde (2017). "Nematus: a Toolkit for Neural Machine Translation." In: *EACL*.

Sennrich, Rico and Barry Haddow (2016a). "Linguistic Input Features Improve Neural Machine Translation." In: *WMT*. Vol. 1, pp. 83–91.

Sennrich, Rico, Barry Haddow, and Alexandra Birch (2015). "Improving Neural Machine Translation Models with Monolingual Data." In: *arXiv*, pp. 86–96.

Sennrich, Rico, Barry Haddow, and Alexandra Birch (2016b). "Edinburgh Neural Machine Translation Systems for WMT 16." In: *WMT - Shared Task Papers*. Vol. 2, pp. 371–376.

Sennrich, Rico, Barry Haddow, and Alexandra Birch (2016c). "Neural Machine Translation of Rare Words with Subword Units." In: *ACL*.

Servan, Christophe, Alexandre Bérard, Zied Elloumi, Hervé Blanchon, and Laurent Besacier (2016). "Word2Vec vs DBnary: Augmenting METEOR using Vector Representations or Lexical Resources?" In: *COLING*.

Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean (2017). "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." In: *ICLR*.

Simard, Michel, Cyril Goutte, and Pierre Isabelle (2007). "Statistical Phrase-Based Post-Editing." In: *NAACL-HLT*.

Snover, Matthew, Bonnie Dorr, Richard Schwartz, Linnea Micciulla, and John Makhoul (2006). "A Study of Translation Edit Rate with Targeted Human Annotation." In: *AMTA*. August, pp. 223–231.

Snover, Matthew, Nitin Madnani, Bonnie J. Dorr, and Richard Schwartz (2009). "Fluency, Adequacy, or HTER? Exploring Different Human Judgments with a Tunable MT Metric." In: *WMT*. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 259–268.

Specia, Lucia (2011). "Exploiting Objective Annotations for Measuring Translation Post-Editing Effort." In: *EAMT*, pp. 73–80.

Specia, Lucia, Nicola Cancedda, and Marc Dymetman (2010). "A Dataset for Assessing Machine Translation Evaluation Metrics." In: *LREC*.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* 15, pp. 1929–1958.

Sutskever, Ilya, James Martens, George E. Dahl, and Geoffrey Hinton (2013). "On the Importance of Initialization and Momentum in Deep Learning." In: *ICML*, pp. 1139–1147.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to Sequence Learning with Neural Networks." In: *NIPS*. Montréal, Canada, pp. 3104–3112.

Turchi, Marco, Rajen Chatterjee, and Matteo Negri (2016). *WMT16 APE Shared Task Data*.

Turchi, Marco, Rajen Chatterjee, and Matteo Negri (2017). *WMT17 De-En APE Shared Task Data*.

Varga, Dániel, Péter Halácsy, András Kornai, Viktor Nagy, László Németh, and Viktor Trón (2005). "Parallel Corpora for Medium Density Languages." In: *Recent Advances in Natural Language Processing (RANLP)*.

Varis, Dusan and Ondrej Bojar (2017). "CUNI System for WMT17 Automatic Post-Editing Task." In: *WMT - Shared Task Papers* 2, pp. 661–666.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention Is All You Need." In: *arXiv*.

Weiss, Ron J., Jan Chorowski, Navdeep Jaitly, Yonghui Wu, and Zhifeng Chen (2017). "Sequence-to-Sequence Models Can Directly Transcribe Foreign Speech." In: *Interspeech*.

Williams, Ronald J. and David Zipser (1989). "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks." In: *Neural Computation* 1.2, pp. 270–280.

Wisniewski, Guillaume, Nicolas Pécheux, and François Yvon (2015). "Why Predicting Post-Edition is so Hard? Failure Analysis of LIMSI Submission to the APE Shared Task." In: *WMT*, pp. 222–227.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean (2016). "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation." In: *arXiv*.

Zanon Boito, Marcely, Alexandre Bérard, Aline Villavicencio, and Laurent Besacier (2017). "Unwritten Languages Demand Attention Too! Word Discovery with Encoder-Decoder Models." In: *ASRU*.

Zaremba, Wojciech, Ilya Sutskever, and Oriol Vinyals (2014). "Recurrent Neural Network Regularization." In: *ICLR*.

Zeiler, Matthew D. (2012). "AdaDelta: An Adaptive Learning Rate Method." In: *arXiv*.

Zhang, Jiajun, Shujie Liu, Mu Li, Ming Zhou, and Chengqing Zong (2014). "Bilingually-Constrained Phrase Embeddings for Machine Translation." In: *ACL*. Vol. 1, pp. 111–121.

Zhechev, Ventsislav (2012). "Machine Translation Infrastructure and Post-Editing Performance at Autodesk." In: *AMTA Workshop on Post-Editing Technology and Practice (WPTP)*, pp. 87–96.

Zhou, Jie, Ying Cao, Xuguang Wang, Peng Li, and Wei Xu (2016). "Deep Recurrent Models with Fast-Forward Connections for Neural Machine Translation." In: *Transactions of the Association for Computational Linguistics (TACL)* 4, pp. 371–383.

Zoph, Barret and Kevin Knight (2016a). "Multi-Source Neural Translation." In: *NAACL-HLT*. Denver, Colorado, USA.

Zoph, Barret, Deniz Yuret, Jonathan May, and Kevin Knight (2016b). "Transfer Learning for Low-Resource Neural Machine Translation." In: *EMNLP*.

Zou, Will Y, Richard Socher, Daniel Cer, and Christopher D Manning (2013). "Bilingual Word Embeddings for Phrase-Based Machine Translation." In: *EMNLP* October, pp. 1393–1398.

Appendix A

Neural Machine Translation

A.1 MultiVec

A.1.1 Word2Vec Tricks

Word2Vec implements the following tricks, which largely contribute to the training speed and the quality of the trained embeddings:

- Subsampling: each token is discarded from the training set with a probability that depends on its frequency (f):

$$p = \max(0, \frac{f - t}{f} - \sqrt{\frac{t}{f}}) \tag{A.1}$$

$$f = \frac{\#w}{\sum \#w'} \tag{A.2}$$

  γ is the subsampling rate, a hyperparameter of the model whose default value is 10^{-5}. Lower values of γ result in more words being dropped.[1] This subsampling strategy drops more often words that are highly frequent (typically stop words). This decreases training time as less updates need to be done.

- Asynchronous SGD (Y. Bengio et al. 2003): to speed up training on multi-core CPUs, the training set is split into several chunks (as many chunks as there are threads), and each chunk is processed in a separate thread. Each thread reads its own part of the training file and performs SGD updates on shared weight matrices. In most contexts, it would be advisable to lock access to the shared parameters while updating them, in order to avoid race conditions, i.e., inconsistent state caused by several non-atomic operations which happen at the same time and overlap. However, in the context of stochastic gradient descent, such race conditions are not detrimental, and thus thread safety is not necessary. Indeed, SGD is highly stochastic, and random write errors only add up to the stochasticity. By not enforcing thread safety, we can greatly improve training speed.

[1] For example, on the English side of News Commentary, values of 10^{-3}, 10^{-4} and 10^{-5} result respectively in 27%, 49% and 73% of all words being dropped.

- Decreasing learning rate: Word2vec decreases its learning rate linearly using $\alpha = \alpha_0 * (1 - \frac{t}{T})$ where α_0 is the initial learning rate, t is the number of words that have been processed up to the current step, and T is the total number of words to process during training (i.e., the number of words in the train set times the number of epochs). This ensures that the learning rate reaches zero near the end of training (i.e., after a fixed number of epochs). Pure SGD is used for training (with a batch size of 1).
- Dynamically sized window: for each word w_t, a new window size S is uniformly sampled between 1 and S_{max}. This is equivalent to weighting the contribution to the loss of each word in the context window, with larger weights near the center of the window, and smaller weights at the extremities. Intuitively, words that are further away are less related and should contribute less to the loss. This implementation also improves training speed, as training complexity depends on the window size.
- Vocabulary filtering: very infrequent words are removed from the vocabulary at the beginning of training, i.e., words whose count in the training set is below some threshold. Words that do not appear in the vocabulary are just removed from the training sentences. This speeds up training by largely reducing the size of the vocabulary. Words that appear only a couple of times would get very poor embeddings anyway.
- Word2vec also uses a number of low-level speed tricks, like a pre-computed exponential table and a custom random generator.

A.1.2 Architecture and API

Directory substructure The MultiVec toolkit is divided into several directories, whose function is described below. Optionally, a `data/` directory can be used to store the training files (text corpora), and a `models/` directory to store the trained models (word embeddings or full models).

- `multivec/` contains the main source code of the library, which we will detail shortly.
- `word2vec/` contains the source code from the original Word2vec (Mikolov et al. 2013c), which we modified slightly to match our command-line interface.
- `cython/` contains the source code of the Python wrapper. Thanks to this wrapper, the toolkit can be used from Python code without performance loss, as the back end still runs in C++.
- `benchmarks/` contains scripts for running a number of evaluation benchmarks: analogical reasoning, sentiment analysis and crosslingual document classification.
- `scripts/` contains a number of pre-processing and post-processing scripts (e.g., tokenization, lowercasing, etc.)

MultiVec structure The MultiVec toolkit relies on two C++ classes: `MonolingualModel` and `BilingualModel`.
A few utility classes are also used: `Config` and `BilingualConfig` that hold the hyperparameters of the models, `HuffmanNode` that holds a vocabulary item, and `Vec` for vector arithmetic. More precisely, the source code is divided into the following files:

- `vec.hpp` defines the `Vec` class, a wrapper around `std::vector<float>` that supports basic arithmetic operations between vectors. We define a matrix (`mat`) as a simple list of `Vec` instances (alias to `std::vector<Vec>`).

- `utils.hpp` contains the definition of the `Config` and `BilingualConfig` classes, which contain the settings of a model (e.g., dimension, learning rate or window size). It also defines several utility functions (e.g., for random number generation), and the class `HuffmanNode`. This class corresponds to a node in a Huffman Tree, a kind of binary tree where the leaves are words (in our case), and the most frequent words have a shorter path from root to leaf.

  A word can be encoded as a path in the tree (sequence of zeros and ones, indicating right or left branches). Each inner node has its own output embedding (stored in the `MonolingualModel::output_weights` matrix). The probability of following the left path from a given node i is $p_i = \sigma(v_i^\top h)$ where v_i is the embedding of the node, and h the context vector. The probability of taking the right branch is $1-p$. The probability of a word is decomposed as the probability of its path: $\prod_i p_i^{r_i}(1-p_i)^{1-r_i}$. Because the path length is $O(\log_2 |V|)$, this reduces the complexity to $O(n \times \log_2 |V|)$, compared to $O(n \times |V|)$ for regular softmax. Huffman Trees are more efficient than balanced binary trees because they give shorter paths to frequent words, whose probability can be estimated faster than words that are seen more rarely. For example, when building a tree from the English side of News Commentary, the word "the" has an encoding of length 4, while the rare word "zealot" has a path of length 20.

- `monolingual.{hpp,cpp}` define the `MonolingualModel` class. It has a configuration, a vocabulary and weight matrices (input and output weights).

  The vocabulary is stored as a list (`std::vector<HuffmanNode>`) of Huffman nodes. A word is identified uniquely by the position of the corresponding node in this list, which also gives the position of its input and output embeddings inside the `input_weights` and `output_weights` matrices. A hash map is also built that maps each word to the corresponding index (`std::unordered_map<string, int>`).

  The model is initialized either with a call to `train` (which initialize the weights at random and then trains with given text file), or with `load` (which loads an existing model).

  `train` starts by calling `read_vocab`, which reads the training file and builds a vocabulary (i.e., a Huffman Tree and a unigram table used for random sampling of words according to their frequency). Then, `init_net` is called to initialize the weights of the model, and `chunkify` to find the starting position of each file chunk for parallel training. `train_chunk` is called as many times as there are threads (with the corresponding file chunk). This function trains the model by reading its own part of the training corpus line by line, for a fixed number of epochs. `train_sentence` is called for each sentence, and `train_word` for each word in the sentence and its local context. `train_word_CBOW` is called for the CBOW model, or `train_word_skip_gram` when `config.skip_gram = true`.
  Online Paragraph Vector is computed by the `sent_vec` method which also makes use of `train_word` (with additional parameters).

  Models can be serialized with `save` (which saves everything, including the configuration and vocabulary). To save only vectors in the Word2vec format, `save_vectors` can be used. The `policy` parameter controls whether to save the input embeddings, the output embeddings, or a concatenation or sum of both.

- `bilingual.{hpp,cpp}` define the `BilingualModel` class. It has two instances of `MonolingualModel`. Its configuration is an instance of `BilingualConfig`, which has an additional `beta` parameter. This controls the strength of crosslingual updates ($\beta = 1.0$ gives as much strength as monolingual ones). The `train` method takes two files instead of a single one (a parallel corpus). The entire (bilingual) model can be saved with a call to `save`, and its monolingual components can be saved with `src_model.save` and `trg_model.save`. Optionally, a third file can be passed to `train`, which corresponds to a pre-computed word-based alignment of the training corpus, in the `fast_align` format (Dyer et al. 2013). If this file is not provided, MultiVec does a uniform word alignment between sentence pairs.

- `serialization.hpp` is a custom serialization library.[2] It defines functions for saving and loading all types of objects that MultiVec uses (configuration, vocabulary, model parameters) in a binary format, so that entire models can be saved (and not just their embedding vectors). This is useful for accessing the features of a model after training it, like similarity measures, online paragraph vector, or incremental training.

- `distance.cpp` contains the definition of several methods of `MonolingualModel` and `BilingualModel` for computing the similarity (or distance) between words or sequences. These functions can be used thanks to the Python API, or the C++ public API (not the command line). For example, `BilingualModel::similarity` can compute the cosine similarity between a word in the source language and a word in the target language. `MonolingualModel::closest` can retrieve a list of the closest words in the vocabulary to given word w.r.t. cosine similarity. `soft_word_error_rate` computes the Word Error Rate between two sequences, where the cost of the substitution between two words is their cosine distance (instead of 1).

- `main.cpp` and `main-bi.cpp` define the command-line interface of the monolingual and bilingual models.
  They compile as two binaries: `bin/multivec` and `bin/multivec-bi`. We will describe this command-line interface shortly.

- `analogy.cpp` is a standalone program for evaluating word embeddings on the analogical reasoning task. Contrary to Word2vec's `compute-accuracy.c`, it does multi-threaded evaluation (one topic per thread). It also proposes more options (like case-sensitive evaluation) and displays more fine-grained results (e.g., balanced accuracy).

Installation The software dependencies of MultiVec are a recent version of `g++`, `CMake`, and optionally `Cython` and `NumPy` for the Python wrapper. Here is how to install the dependencies (on Ubuntu/Debian), and then download and install the toolkit:

```
sudo apt-get install cmake g++ cython3 python3-numpy
git clone https://github.com/eske/multivec.git
cd multivec
./compile.sh
```

The binaries are then available inside `multivec/bin`. To compile the Python wrapper, run `cd cython && make` (for Python 2: `make python2`). This creates a file named

[2] `boost` seemed unnecessarily heavy, and we wanted MultiVec to have as few dependencies as possible.

`multivec.so` inside `cython/`. To use this library with Python, either run the Python interpreter in this directory, move the file to the working directory, or add the library to the `PYTHONPATH` variable:

```
echo "export PYTHONPATH=`pwd`:\$PYTHONPATH" >> ~/.bashrc
```

Command-line interface Like Word2vec, MultiVec can be used to train new models from the command line. We provide two binaries: `bin/multivec` and `bin/multivec-bi`. The `-v` or `--verbose` option prints more information during training (like training progression).

For example, given a training file `data/news.en`, with one sentence per line, and tokenized at the word level[3] (and optionally lowercased), a monolingual model can be trained as follows:

```
# train and save a monolingual model
bin/multivec --train data/news.en -v --save models/news.bin
```

This uses the default settings: CB-NS model with `dim=100, iter=5, negative=5, win_size=5, min_count=5, threads=4`. A Skip-Gram model can be trained with the option `--sg`, and the Hierarchical Softmax objective (instead of Negative Sampling) can be used with `--hs`. All other options (dimension, iterations, etc.) can be modified thanks to command-line parameters. Run `bin/multivec -h` to get a list of the available options.

If training is interrupted by the user (CTRL+C), the model is saved before exiting. Training can then be resumed by loading this model. However, the right learning rate and number of iterations have to be set manually.

To only save the vectors, use the `--save-vectors` option. This writes the vectors in a text file, following the Word2vec format.[4] By default, MultiVec uses 4 threads that each read their own part of the training file, this can be modified with the `--threads` option.

To use the paragraph vector algorithm in batch mode (i.e., compute embeddings for each sentence in the training set), use the `--sent-vector` flag:

```
# train and save paragraph vectors
bin/multivec --train data/news.en -v --sent-vector \
  --save-sent-vectors models/news.sent.vec \
  --save-vectors models/news.vec
```

This saves one vector per line (aligned with `data/news.en`), with values in text format separated by a whitespace. By default, MultiVec uses the PV-DM algorithm (Paragraph Vector Distributed Memory), which is analogous to CBOW. The `--concat` parameter can be used to concatenate context vectors instead of averaging them. It is possible to use the PV-DBOW algorithm (analogous to Skip-Gram) by combining the `--sent-vector` and `--sg` flags.

Given an existing model (with `--load`), paragraph vector can also be used in online mode to compute representations for unseen sentences. This is done with `--online-sent-vector`. This freezes the other parameters of the model and does SGD on the sentence weights only. For instance:

[3] The scripts `scripts/prepare-data.py` or `scripts/tokenizer.perl` can be used for this purpose.

[4] One-line header with vocabulary size and dimension of the embeddings, then one line for each word, containing the word and the list of embedding values separated by a whitespace.

```
# compute new paragraph vectors using an existing model
bin/multivec --load models/news.bin -v \
  --online-sent-vector data/movie-reviews.en \
  --save-sent-vectors models/movie-reviews.sent.vec
```

Bilingual models are trained with the `bin/multivec-bi` program. Instead of a single training file, it takes a pair of files (a parallel corpus) with the `--train-src` and `--train-trg` options:

```
# train and save bilingual model
bin/multivec-bi --train-src data/news.en --train-trg data/news.de \
  --save models/news.en-de.bin -v
```

By default, MultiVec does a uniform word-based alignment between parallel sentences. This means that in bilingual updates, the source context at position $\frac{A \times i}{B}$ (where A and B are the lengths of the source and target sentences) is used to predict the target word at position i. Optionally, a word alignment of the training corpus by `fast_align` (Dyer et al. 2013) can be provided with the `--alignment` option. The program `scripts/align.sh` can be used to generate such an alignment. The `--beta` option controls the strength of bilingual updates compared to monolingual ones (1 means equal strength).

Source and target models can be saved individually as monolingual models with `--save-src` and `--save-trg`. The options `--save-src-vectors` and `--save-trg-vectors` can be used to only save the word embeddings.

Python API The Python library, named `multivec`, defines `MonolingualModel` and `BilingualModel` classes, with a similar interface as their C++ homonyms.

A new model is created by calling the `MonolingualModel` initializer with the settings of the model as keyword arguments. These settings can then be accessed (read or modified) as attributes of the model. For example:

```
from multivec import MonolingualModel, BilingualModel

mono_model = MonolingualModel(dimension=100, threads=4,
                              alpha=0.1, iterations=10)
mono_model.verbose = True
```

An existing model can be loaded with the `load` method (a shortcut is to directly give the path to the model file to the initializer), or a new model can be trained by calling its `train` method with the path to the training file:

```
mono_model.train('data/news.en')
# or mono_model.load('models/news.bin')
mono_model.save_vectors('models/news.vec')
```

To perform incremental training (and not initialize the model parameters to new values), pass `initialize=False` to the `train` method. A number of methods are available to manipulate a trained model. To get the vector representation of a given word, use `word_vec`. Use the `policy` keyword argument to get the input weights (`policy=0`), a concatenation of the input and output weights, a sum of them, or just the output weights (`1`, `2` or `3`).

`similarity` or `distance` return the cosine similarity or cosine distance between two words. `closest` returns the list of n closest words to given word according to their cosine similarity. Paragraph vector can be computed in an online fashion thanks to `sent_vec`. Similarity measures between sequences are also available, with `similarity_bag_of_words` or `soft_word_error_rate`.

```
print(mono_model.similarity('paris', 'london')) # 0.834
print(mono_model.closest('france', 2))
# [('britain', 0.763), ('germany', 0.754)]
```

Bilingual models can also be trained and manipulated. A Bilingual Model contains two Monolingual Models that can be accessed as attributes (`src_model` and `trg_model`).

`BilingualModel` also has several utility functions to manipulate vectors in a crosslingual way. For example `similarity` computes the cosine similarity between a source language word and a target language word. `trg_closest` looks for the target language words that are most similar to the given source language word.

```
bi_model = BilingualModel(dimension=100, threads=4,
                          iterations=10, alpha=0.1, beta=1.0)

bi_model.train('data/news.en', 'data/news.de')
bi_model.src_model.save_vectors('models/news.en.vec')

print(bi_model.similarity('munich', 'münchen')) # 0.83
print(bi_model.trg_closest('thesis', 3))
# [('these', 0.67), ('ökonomielehre', 0.57), ('theorie', 0.57)]
```

A.2 Seq2seq

A.2.1 TensorFlow

A brief overview of TensorFlow was given in the State of the Art. In this subsection, we give a bit more details and tricks that are useful to understanding our framework.

Graph As a quick reminder, before using a TensorFlow model, one needs to build a static computation graph. To do so, the TensorFlow Python API provides a number of functions for adding new operations to the graph. There are three main types of objects: `tf.Tensor`, `tf.Variable`, and `tf.Operation`. A Tensor is the basic graph unit, which holds the result of a computation. It has a rank (the number of dimensions), a shape (the size of each

dimension), and a type (typically floating point or integer). It can represent scalars, vectors, matrices or higher-order tensors. Generally, the first dimension corresponds to the batch size, as all computations for a single mini-batch are done at once and their results stored into a single tensor.

A Variable is a special kind of Tensor that can store values (a Tensor forgets its value between each execution of the graph). It can be defined thanks to the `tf.get_variable` function. In addition to its shape and type, it also takes an 'initializer' parameter, which defines what its initial value should be (e.g., sampled from a uniform distribution), and a name which identifies it uniquely. Variables are generally used to represent model parameters (weights and biases).

An Operation represents a graph node that performs computation. It takes as input a number of Tensors, and outputs a number of Tensors. For example: `z = tf.add(x, y)` creates a new 'Add' operation (which is automatically included into the graph), whose input Tensors are `x` and `y`, and output Tensor is `z`. Some operations can have side-effects, like printing something to the screen (`tf.Print`) or changing the value of a Variable (e.g., `tf.assign`).

Input tensors Since the graph is directed, it has a number of input Tensors that are not the result of an operation. Input Tensors can be variables, constant tensors (whose value is defined at compilation time), or placeholders (whose value is given by the user when running the graph). For instance, in this code sample we have all three:

```
x = tf.ones(shape=[2], dtype=tf.float32)     # constant [1, 1]
y = tf.placeholder(shape=[None, 3], dtype=tf.float32)
w = tf.get_variable(shape=[3, 2], name='w') # var (rand init)
z = tf.matmul(y, w) + x    # tensor of shape [None, 2]
```

`z` computes a product between an input matrix and a weight matrix and adds one, which results in a Tensor of shape `[None, 2]`. As a side note, adding `x` in the last line is equivalent to adding 1, but more flexible as `x` can be substituted with other values when running the graph.

Tensor shape TensorFlow tries to infer a *static shape* for each tensor from the preceding operations, which is obtained thanks to the `shape` attribute of Tensor objects. In the previous example, `z.shape` would give `TensorShape([Dimension(None), Dimension(2)])`. As can be seen in this example, the static shape is only partially known: TensorFlow was able to infer the second dimension from the 'MatMul' operation, but was unable to infer the first dimension because the shape of the placeholder `y` was not entirely filled in. The *true shape* of a Tensor is available as a one-dimensional Tensor (which also belongs to the graph), thanks to the `tf.shape` function. This dynamic shape can be used in other operations, but its actual value can only be known at run time. The advantage of leaving out some dimensions in a placeholder (with `None`) is that we can feed values of varying size at run time. This is useful for variable-length sequences and dynamic batch size.

Variables and scopes TensorFlow variables can belong to a scope, which defines certain properties of these variables, like their full name or their default initializer. For example:

```
with tf.variable_scope('my_scope'):
    w = get_variable('my_variable', shape=[3, 4])
```

Then the newly created `w` variable will have the unique name 'my_scope/my_variable'. If we try to create a new variable with the same scope and name, then TensorFlow raises an exception, unless we explicitly tell this scope to reuse previously created variables (with a `reuse` parameter). This feature is convenient for accessing existing variables from anywhere in the code, but it can be tedious to keep track of all variables that have already been created.

To handle this reuse situation automatically, we defined a wrapper around `tf.get_variable`, which catches exceptions and calls the function inside a variable scope with `reuse=True` if needed. This is crucial for our implementation of multi-task training, where several models with shared parameters are created. With this trick, we only need to share the scope names and variable names between the different models, and the corresponding variables are automatically shared.

Starting from TensorFlow 1.4, a variable scope can set its `reuse` parameter to `tf.AUTO_REUSE`, which achieves the same effect as our wrapper.

Using GPUs One of the advantages of TensorFlow is that most of its operations can run seamlessly on GPUs. Unless told otherwise, if GPUs are available, TensorFlow will allocate all the remaining memory of the first GPU and store its variables and run its computation on this GPU. Some variables may be too large to fit on GPU memory, in particular embedding matrices. In this case, it can be wise to explicitly tell TensorFlow to store it in RAM (and run this part of the graph on the CPU). If a machine contains several GPUs, we may also wish to use the second GPU, or to distribute the model on several GPUs. This is possible by defining the variables and operations inside a `tf.device` context. For example, to allocate an embedding matrix on the CPU:

```
with tf.device('/cpu:0'):  # or '/gpu:0', '/gpu:1', etc.
    # /cpu:0 = all CPUs
    embedding = tf.get_variable('embedding', shape=[30000, 256])
```

This is actually recommended for embedding matrices, as they can take large amounts of memory, and the operation for looking up the embedding of a particular item does not run on the GPU anyway (`tf.nn.embedding_lookup`).

Running the graph Once the graph has been built, it needs to be instantiated inside a session. A session allocates the resources (e.g., GPU memory) and holds the values of each variable and intermediate results. The variables of the graph can be initialized (generally to random values) by running the operation `tf.global_variable_initializer()` inside a session. This is necessary before running any other operation, otherwise TensorFlow raises an exception.

```
sess = tf.get_default_session()
sess.run(tf.global_variable_initializer())
```

Any operation or tensor expression can be evaluated thanks to a call to `sess.run`. To run operations that depend on placeholders, actual values for these placeholders need to be given as input to the `sess.run` function.

In the earlier example, to evaluate the value of tensor `z`, a value needs to be provided for `y` as follows:

```
y_data = [[3, 2, 0], [5, 7, 1]]  # batch size of 2
z_value = sess.run(z, feed_dict={y: y_data})
```

This gives a NumPy array of shape `[2, 2]` that corresponds to the result of the computation of `z`. An interesting property of this 'feed dict' approach is that we can give a value for any tensor in the graph, not only placeholders, but also tensors that would normally be the result of an operation.

High-level operations Even though TensorFlow is a general purpose symbolic math library, it is primarily intended as a machine learning or deep learning library. Hence, it offers a number of high-level operations that are useful for deep learning. For example, `tf.dense` takes as input a tensor and returns the output of a fully-connected layer whose size and activation function are given as parameters. It automatically creates the corresponding variables: weight matrix, and optional bias vector. TensorFlow also provides operations for dropout: `tf.nn.dropout`, or batch normalization: `tf.layers.batch_norm`. Most of these operations are intended to work with tensors whose first dimension is the batch size.

Control-flow operations Conditional branching in TensorFlow is not straightforward. Python if-statements, because they are executed only once at the creation of the graph, only result in the creation of one branch, which cannot change during the execution of the graph (because the graph is static).

For example, when training RNNs, there is a technique called "teacher forcing" which consists in feeding the RNN with the previous ground truth symbol instead of the token that was generated at the previous time step. However, we want to be able to differentiate between the training phase (where teacher forcing is enabled) and the decoding phase (where ground truth symbols are not available). To do so, we use a `feed_previous` boolean. When true, we want the RNN to take its previous prediction as input, otherwise it should take the target symbol as input.

Here is how we could (wrongly) proceed with a Python if-statement:

```
if feed_previous:
    input_symbol = tf.argmax(output, axis=1)
else:
    input_symbol = target_symbol
```

With this implementation, depending on the truth value of `feed_previous`, a single graph branch will be created. At run time, only this branch will be executed, even when changing the value of `feed_previous`. As a side note, if `feed_previous` is a tensor, Python will evaluate it to `True`, no matter its actual value (according to Python, everything that is not `False`, `0`, `None`, or an empty container, evaluates as `True`).

The correct implementation, which would allow us to choose the correct behavior at run time, is the following:

```
input_symbol = tf.cond(feed_previous,
    lambda: tf.argmax(output, axis=1),
    lambda: target_symbol)
```

where `feed_previous` is a scalar boolean Tensor. Then, by feeding `True` or `False` to `sess.run`, we can control the behavior of the graph (teacher forcing or no teacher forcing).

There is also a `tf.case` operation which can take several conditions and outputs (like an `if` block with `elif` statements, or `switch` statement in C++). `tf.where(b, x, y)` takes three tensors with the same shape, where `b` is a boolean tensor. It outputs values of `x` where `b` is true, or values of `y` where `b` is false. This is equivalent to: `x * tf.to_float(b) + y * (1 - tf.to_float(b))`.

Loops Python for-loops can be used with TensorFlow, but they result in the creation of a statically unrolled graph, with a fixed number of time steps. The following code example unrolls an RNN to a length of `max_steps`. The result is the final state of the RNN (after reading `max_steps` inputs). Here, `inputs` has a shape of `(batch_size, max_steps, input_size)`.

```
cell = GRUCell(state_size)                          # RNN cell
state = tf.zeros(shape=[batch_size, state_size]) # initial state
states = []

for time in range(max_steps):
    state = cell(inputs[:,time], state)  # update RNN state
    states.append(state)

# as tensor of shape (max_steps, batch_size, state_size)
states = states.stack()
# batch_size as 1st dim
states = tf.transpose(states, perm=(1, 0, 2))
```

There are some problems with this version: the maximum length has to be known (or chosen arbitrarily) at compilation time. Shorter input batches have to be padded to the maximum length. This is very inefficient as short inputs will take as much time to process as the longest inputs. A solution is to create several graphs of different lengths, and run the graph whose length matches that of the inputs (bucketing). But this is very application-dependent, and compilation time can be excessively long.

TensorFlow provides access to symbolic loops, whose most general version is `tf.while_loop`. These loops are dynamically unrolled, i.e., only when executing the graph, and the stopping criterion can be dynamic. Here is how to implement the previous algorithm with a symbolic loop:

```
time_steps = tf.shape(inputs)[1]
states = tf.TensorArray(dtype=tf.float32, size=time_steps)

def time_step(time, state, states):
    state = cell(inputs[:,time], state)
    states.write(time, state)
    return time + 1, state, states

_, state, states = tf.while_loop(
    cond=lambda time, *_: time < time_steps,
    body=time_step,
```

```
    loop_vars=(time, state, states))

# as tensor of shape (time_steps, batch_size, state_size)
states = states.stack()
states = tf.transpose(states, perm=(1, 0, 2)) # batch_size as 1st dim
```

Here, the shape of `inputs` is `(batch_size, time_steps, input_size)`, where the second dimension can be dynamic and batch-dependent. In practice, we group sequences of similar length in the same batch, so as to perform as little computation as needed (`time_steps` is the length of the longest sequence in the batch).

The first parameter of the while loop is a function that takes as input the same parameters as `time_step` and returns a boolean scalar tensor, which controls whether the loop should continue or stop. The second parameter is a Python function, which is executed at each time step and whose outputs are used as inputs for the next call. The result of the while loop is the result of the last call. The third parameter is the list of initial values that are given to the first call. The fourth parameter controls how much computation should be run in parallel, which is a time/memory trade-off. When training, TensorFlow needs to store results of the forward pass and backward pass for all time steps, which gives a memory complexity of $O(T \times n)$ where T is the length of the sequence and n the batch size. Because GPUs have a very limited amount of memory (generally between 4 and 12 GB), this can cause training to crash. The `swap_memory` parameter allows TensorFlow to move these tensors to the CPU when needed, to save GPU memory.

The `time_step` function can also take as input and return variable-length sequences of tensors called `TensorArray`, This allows us to store intermediate results (like the output of the RNN at each step).

TensorFlow also provides more specialized operations like `tf.map_fn`, or `tf.dynamic_rnn` (simpler for the example we just described). But the generic while loop provides more control. We use it for our attention-based decoder.

Debugging Because of the distinction between graph creation and graph execution, debugging in TensorFlow can be tough.

Most bugs can be solved before run time, by looking at the static shapes of the tensors. To do so, we only have to set breakpoints (e.g., with `pdb` or `ipdb`) in the definition of the graph where the error happens, and inspect the shapes of the tensors by hand (or use plain old print statements).

Sometimes, the static shape may be fine, but the variable shape components can be faulty (e.g., element-wise addition of two tensors with a different batch size). This can often be fixed by inspecting the value of the dynamic shape at run time.

For this purpose, the operation `tf.Print` is convenient. It takes a tensor as first argument that it returns unchanged (identity function), except for a side-effect that prints the values of the second argument (a list of tensors) whenever the returned tensor is evaluated.

A typical use case is the following:

```
>>> x = tf.Print(x, [tf.shape(x)], message='shape: ')
```

Every time tensor `x` is evaluated, directly or indirectly (e.g., through a call to `sess.run(x)`), TensorFlow outputs its value, and prints the value of `tf.shape(x)` as a side-effect. When printing values, and not shapes, the matrices can be quite large. For debugging, it is more convenient to print only the first element in the batch `x[0]`.

A.2.2 Architecture and API

Package structure Our seq2seq framework is divided into several folders: `translate/`, which contains the main source code, `scripts/` which contains pre-processing, scoring and monitoring scripts, `config/` with the configuration files of the experiments, `data/` which contains the pre-processed training and evaluation data for the experiments, and `models/` where the trained models are saved. We also use a `raw_data/` directory for storing data archives and raw (non-tokenized) data.

The source code (inside `translate/`) is structured into several files:

- `__main__.py` is the public interface of the framework. It parses configuration files and command-line arguments, creates a TensorFlow Session and initiates training or decoding. The script `seq2seq.sh` provides a convenient way of running this module.
- `translation_model.py` defines the `TranslationModel` class which contains the main logic of the program. It is responsible for reading the training data, creating and initializing the model, training, decoding, and saving checkpoints. It defines several public methods: `train`, `decode` and `evaluate` which are called by the main program depending on the user's commands.
- `multitask_model.py` defines the `MultiTaskModel` class. It has several instances of `TranslationModel`, and its `train` function alternately trains each of those models by calling their `train_step` function (multi-task training).
- `seq2seq_model.py` defines the `Seq2SeqModel` class, which is responsible for building the TensorFlow graph, and running the graph for training or decoding.
- `models.py` defines all the functions for creating each block of the graph: multi encoder, attention decoder, global attention, sequence loss, etc. (see below for a detailed overview of the graph).
- `utils.py` contains several utility functions for logging, plotting, reading datasets and iterating over datasets. It also defines global variables such as the default vocabulary ids for special tokens (UNK, EOS and BOS symbols).
- `evaluation.py` defines a number of evaluation metrics, like corpus BLEU, TER or WER, to compare a set of hypotheses against reference translations. These functions are used by `TranslationModel.evaluate`, which is called periodically during training, or when running seq2seq in evaluation mode.
- `beam_search.py` defines the beam search decoder, which takes as input initial state(s) and time step function(s). It outputs tensors that compute the result of beam search decoding by one or several models (ensemble decoding).

The `scripts/` folder contains pre-processing scripts. In particular, `prepare-data.py` is useful to transform raw training data into a form fit for seq2seq: file naming, tokenization, vocabulary creation.[5] Seq2seq expects a single data directory (whose path is defined in a configuration file), containing tokenized files whose name is a corpus prefix followed by an extension. The default prefixes for training and evaluation corpora are `train` and `dev`, and the default prefix for the vocabulary files is `vocab`. The same extension should be used for all files in a given language. For example, for a French to English model whose encoder and decoder are named `fr` and `en`, and whose data directory is `data/`, seq2seq would expect the following set of files: `data/{train,dev,vocab}.{fr,en}`.

We also developed several scripts for monitoring and obtaining various information about data or models:

- `stats.py`: similarly to UNIX's `wc` command, counts lines, words and characters, but also provides additional information: average word or character count, number of unique words, maximum length that covers 90%, 95% or 98% of all lines, etc. It is useful to help choosing a vocabulary size and a maximum sentence length (for efficiency reasons) which cover most of the training set.
- `score.py`: command-line interface to `evaluation.py`'s scoring metrics. It takes as input a hypothesis file (e.g., produced by seq2seq) and a reference file (typically the target of a parallel corpus), and prints the scores produced by BLEU, or other metrics like TER or WER.
- `get-best-score.py` provides a convenient way of looking for information in the log files produced by seq2seq. The script can take a list of model directories and outputs the best score of each model (e.g., best BLEU score or best dev loss).
- `plot-loss.py` takes one or several seq2seq log files and plots the evolution of training loss, dev loss, or dev score (e.g., BLEU) during training w.r.t. SGD steps. It is useful to quickly compare models, or to observe overfitting/underfitting (by comparing train loss with dev loss). It also has a text mode (with `--auto` or `--text` flag), which is useful for plotting on servers than do not have a display (e.g., over SSH).

TensorFlow graph The graph is created by `Seq2SeqModel`'s constructor. It calls several functions inside `models.py`, which are responsible for building each part of the model. Figure A.1 describes the main blocks of the seq2seq model.

Function `multi_encoder` creates one or several encoders (for multi-source scenarios like Zoph et al. (2016a)). Its inputs are two placeholders that hold the input sequences (tensors of shape `[batch_size, time_steps]` containing token ids), and their length (tensors of shape `[batch_size]`). Because all sequences in a batch must have the same length, shorter ones are padded with dummy `</S>` symbols. This tensor stores their true length, before padding. This is used by the bidirectional RNN to know how to reverse the sequences, and by the attention mechanism not to attend dummy states.

First, the input ids are transformed into vectors using an embedding matrix. The function `tf.embedding_lookup` looks for an embedding vector using its index. This is more efficient than doing a dot product between a very large one-hot vector and a dense embedding matrix.

[5] It uses pre-processing scripts distributed with Moses (Koehn 2010), and Byte Pair Encoding scripts from Sennrich et al. (2016c).

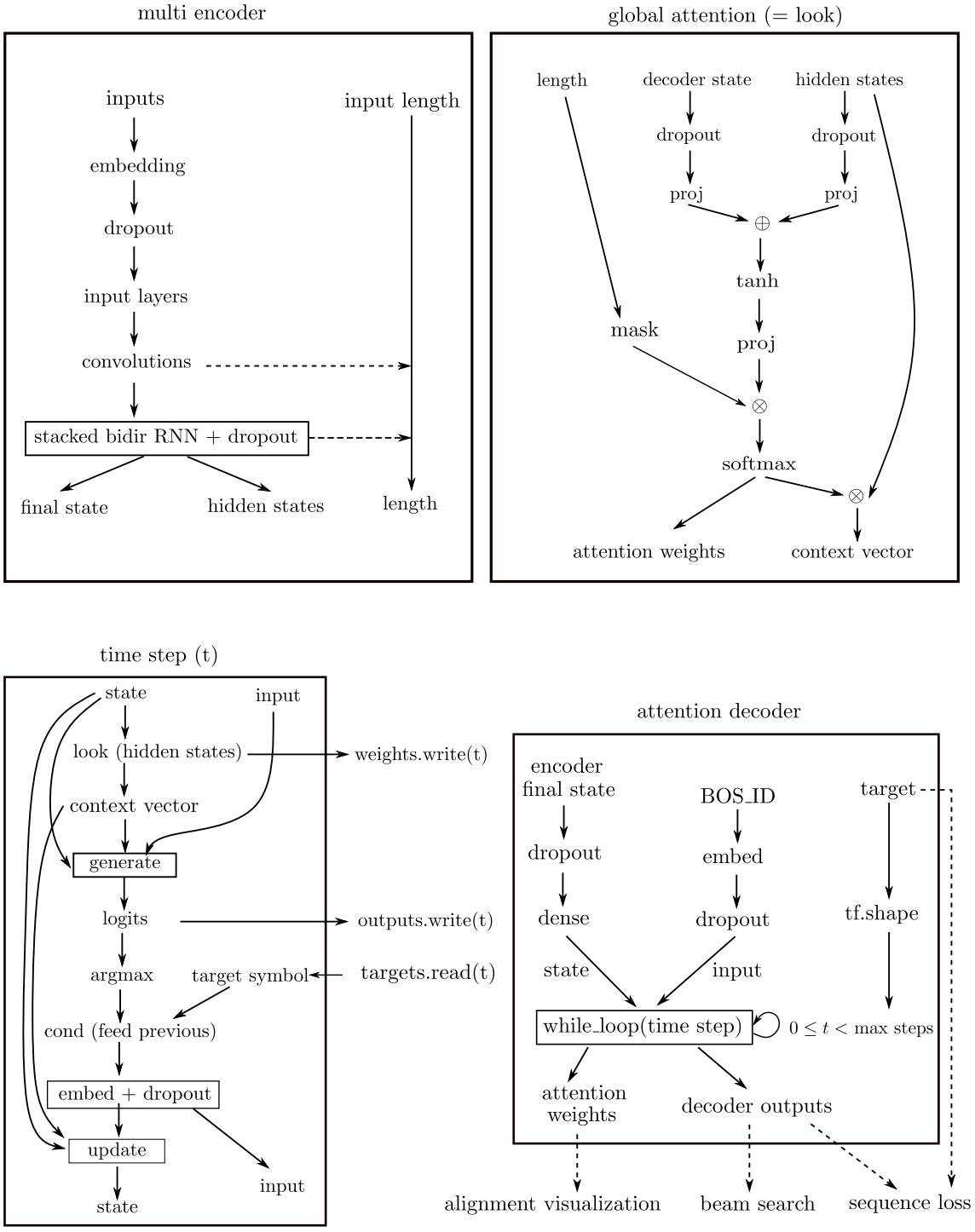


FIGURE A.1: Main blocks in the seq2seq graph

Then, the encoder can perform several transformations of this dense input, like a dropout layer that either drops entire words or embedding values, fully connected input layers, and convolution layers that can reduce the time resolution of the sequence. Finally, this sequence is passed to a recurrent neural network, which can be bidirectional and/or multi-layer (stacked RNNs), and can use LSTM cell(s) or GRU cell(s). This RNN outputs a sequence of hidden states (the output of the cell at each time step) and a final state. In addition to these tensors, the decoder outputs the new length (which can be altered by convolution layers or a pyramidal RNN).

These tensors are taken as input by `attention_decoder`, which also takes a placeholder that holds the reference translations for training (target side of the parallel corpus). At decoding time, this placeholder is fed dummy symbols. It also takes a `feed_previous` boolean tensor which activates teacher forcing (feeding the target symbol to the decoder instead of its own outputs). The decoder's while loop calls a `time_step` function repetitively to generate a sequence of outputs. A similar but more compact function is also used for beam search decoding.

The `time_step` function takes current decoder state and current input (embedded symbol) and computes a new state and output, by using three functions: `look` (a wrapper around `global_attention`), `generate`, and `update`. It can look at the hidden states of several encoders, by using several attention mechanisms and concatenating or summing their context vectors. This context vector is then used to generate a new output (`generate` function), which is stored into a `TensorArray`. This output, which we also call 'logits' contains scores for each token in the target vocabulary. Then, depending on whether we are training or decoding (`feed_previous` parameter), we embed the argmax of this output tensor (i.e., the symbol with the highest score) or the target (ground truth) symbol at this time step, and feed it to the `update` function to update the RNN state. The `time_step` function returns the new state, and new input (embedded symbol), which are used as input for the next time step. The decoder's while loop runs this function as many times as there are time steps in the target placeholder (second dimension). At decoding time, this corresponds to the maximum length set in the configuration file. The first call to `time_step` takes as input an initial state and an initial output. The initial state is a non-linear transformation of the encoder's final state (single dense layer with bias). The initial input is the embedded BOS symbol (Beginning of Sequence), which informs the decoder that it should proceed to decode the first symbol.

The decoder returns a tensor that contains a score for each item in the vocabulary (of shape `[batch_size, max_len, vocab_size]`), as well as the attention weights (useful for visualization), the initial state and a time step function for beam search decoding.

`sequence_loss` computes a cross-entropy loss between the decoder outputs and the reference translation. This loss is optimized thanks to SGD or Adam (`tf.train.AdamOptimizer`).

Most hyperparameters of the model (e.g., RNN cell type and size, embedding size, number of RNN layers, convolutions, dropout rate) are not simply hard-coded but can be configured. We will describe shortly how this configuration is done (with config files and command-line arguments). Seq2seq passes these parameters around as `AttrDict` instances (a kind of dictionary). Each encoder and decoder has its own dictionary of parameters. For example `multi_encoder` takes an `encoders` parameter, which is a list of dictionaries of hyperparameters for each encoder. The first encoder's cell size is `encoders[0].cell_size`. This is convenient, as many new parameters can be added to the encoders and decoders, without cluttering the function signatures too much and without having to explicitly pass these parameters to each new function call.

Configuration Each model's configuration is defined inside a dedicated YAML configuration file. The default configuration (the default value of each hyperparameter) is defined inside the file `config/default.yaml`.[6]

Seq2seq first parses this file, and then reads the experiment's config file. Any parameter that is redefined overrides the default value. Naturally, the user only has to specify the parameters whose value differs from the default value.

Several parameters have no default value and should always be defined: the encoder and decoder names and paths to data and model directories. An example of minimal configuration file is given below.

```
label: 'BTEC baseline'

cell_type: GRU
cell_size: 256
attn_size: 256
embedding_size: 128

data_dir: data/BTEC
model_dir: models/BTEC/baseline
batch_size: 32
max_len: 25

optimizer: adam
learning_rate: 0.001
steps_per_checkpoint: 2000
steps_per_eval: 2000
max_steps: 30000

encoders:
    - name: fr
decoders:
    - name: en
```

Each encoder and decoder can have their own hyperparameters. There can be several encoders and several decoders. For these reasons, encoder parameters are defined as a list of dictionaries, whose name is `encoders`. Each item in the list (which begins with '-' in YAML) has to define a `name` parameter. The `decoders` parameter follows the same format.

The name of an encoder or decoder determines the training file names (unless a different `ext` parameter is specified) and the names of model variables. In the example, the encoder's embedding variable will be named `embedding_fr` and all other encoder variables will be created in the `encoder_fr` scope. The source training file is `train.fr`, unless a different prefix is specified with `train_prefix` (in the main scope), or a different extension with `ext` (in the encoder's scope).

When `encoders` and `decoders` are defined in the main scope of the config file, a single task is created (mono-task training). To create multiple tasks, a `tasks` parameter can be defined,

[6] It is also self-documenting, as a short description of each parameter is given.

which consists in a list of dictionaries, each with a name and its own lists of encoders and decoders.

```
tasks:
    - name: fr_en
      encoders:
        - name: fr
      decoders:
        - name: en

    - name: fr_de
      encoders:
        - name: fr
      decoders:
        - name: de
```

The same parameters can be defined at multiple locations. Local parameter definitions have a higher precedence over global ones. For example, if the main scope contains `cell_size: 256`, and the first encoder contains `cell_size: 128`, then this encoder's cell size is set to `128`, and the decoder's cell size to `256`.

Several of the parameters that we show in the config example (namely `max_len`, `cell_size`, `embedding_size` and `cell_type`) are encoder and decoder dependent. Different values can be specified inside the configuration of each encoder and decoder. `max_len` controls the maximum number of tokens (words or subwords, or characters with a char-based decoder) in each source and target sequences. Seq2seq truncates sequences that exceed this threshold. When decoding, this parameter controls the maximum length of the translation hypotheses. Decoding time is linear with respect to the max length. This parameter is also important to limit training time, and even more importantly the amount of GPU memory that is used when training. Indeed, the time and space complexity of the BPTT algorithm is linear with respect to sequence length.

The `steps_per_checkpoint` and `steps_per_eval` parameters control the number of SGD updates between two checkpoints, and between two evaluations on the dev set. 2000 steps with a batch size of 32 corresponds to 2000×32 examples.

A checkpoint is a file that contains the values of all model variables at some point in time during training. Seq2seq keeps the latest checkpoint, as well as a number of "best checkpoints", i.e., models whose performance on the dev set is the best according to an evaluation metric. This performance is evaluated every `steps_per_eval` steps. To do so, seq2seq temporarily interrupts training, activates decoding mode (teacher forcing off and dropout off), and decodes the entire dev set. Then, it evaluates the hypotheses against the target side using the evaluation metrics defined by the user. The parameter `score_functions` takes a list of evaluation functions which are defined in `evaluation.py`. Existing candidates are: dev loss, BLEU, WER (Word Error Rate), TER, CER (Character Error Rate) and BLEU-1 (unigram BLEU). The first element in the `score_functions` list is the main scoring function which is used for keeping the best checkpoints. This is a way of performing "early stopping", a form of implicit regularization, which stops training when the performance on the dev set starts degrading (overfitting).

`max_steps` interrupts training once this number of SGD updates have been performed. The number of epochs (pass through the entire training set) can be controlled thanks to `max_epochs`. The other parameters are documented inside `default.yaml`.

Command-line arguments The `seq2seq.sh` script takes a number of command-line options that decide what action should be taken exactly. Some of the options can also override parameter values defined in the configuration files. The first argument should always be a path to the model's config file. Then, the user should specify in which mode she wants to run the program: `--train` for training mode, or `--decode`, `--eval` or `--align` for decoding, evaluation or alignment modes. We also provide a `--crash-test` mode which tries to train with the longest sequences in the training set, to test whether there is enough GPU memory for this configuration.

- Training mode: trains a model and saves it in the `model_dir/` directory, with training data stored inside `data_dir/` directory. It resumes training if a model with saved checkpoints exists at this location. The `--purge` flag erases any model that was stored at the same location, which effectively restarts training from scratch.

  Training goes on forever, unless manually interrupted by the user (e.g., with CTRL+C), or after reaching a specified number of epochs or updates. The `--model-dir` option can change the default model directory (useful for training several instances of the same model without having to change the config file). Seq2seq creates a directory, and copies all files that are necessary for using the model once trained, or for replicating the experiment (vocabulary files, config files and current source code). Furthermore, a log file is created, which includes information about model settings (configuration, GPU id), variables (list of variables and their shapes), and training performance (time, training loss, periodic evaluation). The `--verbose` or `-v` flag logs even more information (useful for debugging). When running seq2seq in a distant terminal (e.g., with SSH), the screen command is particularly convenient.[7]

- Decoding mode: if no argument is specified, interactively decodes the user inputs (or anything that is sent to the standard input). Otherwise, the first argument should be the path of the file to decode (several paths in multi-source settings). The `--beam-size` option controls the size of the set of hypotheses for beam search decoding. By default, seq2seq performs greedy decoding (beam size of one). Decoding speed and memory usage are greatly impacted by batch size, which can be changed for this instance with `--batch-size`.

  By default, seq2seq loads the best checkpoint. A custom checkpoint can be chosen with the `--checkpoints` option. If several checkpoints are given, seq2seq loads each of them (in reading order), unless the `--average` or `--ensemble` flags are toggled. In this case, TensorFlow either averages all the checkpoint's variables into a single model, or builds as many models as there are checkpoints and does ensemble decoding (by averaging their log-probabilities). The checkpoints option can also be used in training mode for pre-training.[8] By default, seq2seq outputs the translation hypotheses on the standard output. This behavior can be modified with the `--output` option which gives the path of the file where these outputs should be written. All these options can also be used in evaluation mode.

[7] `screen -S my_screen_name` to launch a new screen, then run any command in it, and type `CTRL+A` then `D` to detach from it. You can go back to a screen anytime with `screen -r my_screen_name`. You can stop a command running inside a screen with `CTRL+C`. Close the screen for good (if no command is running) with `CTRL+D` or `exit`.

[8] For example, train a model with the same parameters on a different data set, and initialize a new model with it, or train a different model on another task that shares some parameters (e.g., the encoder). Special care should be given to the names of the encoders and decoders in the configuration files (which controls how the parameters should be shared).

- Evaluation mode: very similar to decoding mode, except that it provides a shortcut for evaluation. If no argument is specified, it evaluates on the dev corpus (specified in the config file). Otherwise, a list of files can be given, or a corpus prefix (which should be located in the data directory). At the end of decoding, seq2seq runs the chosen evaluation metric(s) (`score_functions`) to compare the translation output with the target side of the evaluation corpus. Contrary to decoding mode, the hypotheses are not printed on the standard output.
- Alignment mode: like evaluation mode, but instead of evaluating it decodes line by line and shows the alignment performed by the attention mechanism, with forced decoding (i.e., teacher forcing). To perform unconstrained alignment between the input and the MT output (not the target), the `--align` option (without any argument) can be combined with the `--decode` option.