Improving methods to learn word representations for efficient semantic similarites computations

Julien Tissier

THESE DE DOCTORAT DE L'UNIVERSITE DE LYON

opérée au sein de
l'**Université de Saint-Etienne**
N° d'ordre NNT : 2020LYSES008

Ecole Doctorale **ED SIS 488**
Science, Ingénierie et Santé

Spécialité de doctorat : **Informatique**

Thèse préparée par **Julien Tissier**
Soutenue le 4 mai 2020

Improving methods to learn word representations for efficient semantic similarities computations

---

Devant le jury composé de :

Massih-Reza AMINI	Professeur	Université Grenobles Alpes	Rapporteur
Julien VELCIN	Professeur	Université de Lyon 2	Rapporteur
Elisa FROMONT	Professeure	Université de Rennes 1	Examinatrice
Laure SOULIER	Maître de Conférences	Sorbonne Université	Examinatrice
Christophe GRAVIER	Maître de Conférences	Université de Saint-Etienne	Directeur de thèse
Amaury HABRARD	Professeur	Université de Saint-Etienne	Co-directeur de thèse

Contents

Introduction

Natural Language Processing (NLP) is a branch of Artificial Intelligence that aims to make machines able to process and understand information from the human language in order to solve some specific tasks, like translating a document or automatically writing reports from market data. In the context of this thesis, we focus on textual information, which can be found in webpages, books, communication applications, etc. With the democratization of the Internet, the amount of written and shared textual information has never been so important. For example, each second[1], almost 9,000 tweets[2] are published and almost 3 million emails are sent. Being able to process this large amount of data to extract meaningful information can no longer be achieved manually, and for this reason, the development of NLP models have rapidly gained interest within the last couple of years to do it automatically. These models have become more and more complex in order to solve more and more difficult tasks, like with self-generating dialog systems (chatbots) or with personal assistants (Siri, Alexa, etc.). Unlike humans, these models do not directly process words to solve a given task. Most of the NLP models are based on Machine Learning (a subfield of Artificial Intelligence) models which commonly use vectors as a representation of data in order to apply some algorithms and solve the task. Therefore, before solving a task, a common preliminary step of NLP models is to represent the elements one can find in textual information as vector representations.

In this thesis, we are interested in methods used to generate those vector representations of textual information, and more specifically, representations of words since they are the most basic unit one can find in a text. These word representations are called *word embeddings* and are typically represented as an array of values. Word embeddings are then used in downstream NLP models to solve tasks. The values in word embeddings have to be set such that they reflect the linguistic properties of words and the relations between them. Indeed, they are the main source of information the downstream models have access to in order to solve the task. Without linguistic information encoded into word embeddings, the models would not have any knowledge to know how to use the word vectors and give the expected answer or the correct prediction for a given task. The first works on word embeddings manually created word vectors by using specific linguistic properties of words, generated by linguistic experts. However, these methods are not scalable when the number of words to embed is in the order of millions, which is commonly the case for popular languages like English and the massive amount of textual data available on the Web. Therefore, to overcome this shortcoming, some methods have been developed to automatically learn word embeddings. They are usually based on the statistics of the occurrences of words in texts to learn word representations that convey linguistic information. These methods are mostly *unsupervised*: there are no clues or hints given to the method to know which linguistic information should be encoded into the embeddings, or which word vectors should be related.

[1] As of March 2020, according to `https://www.internetlivestats.com/one-second/` .

[2] A *tweet* is a short message (up to 280 characters) posted on the website `https://www.twitter.com` .

During the last decade, many different methods have been created to learn word embeddings that capture linguistic information in order to be used in downstream NLP models to solve tasks. Among these methods, two main limitations can be observed:

- Most of the methods learn word embeddings with statistics from large training texts extracted from the Web. These texts are, for the majority, generic and do not contain many specific linguistic information, which is therefore not captured into the word embeddings learned by these methods.

- As the tasks to solve become more difficult, the linguistic knowledge required to solve them increases, and so is the complexity and the size of the methods that learn to encode this additional knowledge into word embeddings. With the democratization of NLP applications running on low-resource devices like smartphones, such complex and large word representations models cannot be used on those devices.

These two limitations of current word embeddings are complementary: if more linguistic information is encoded into word vectors with other sources of information, the representations would become larger and would not be able to run on low-resource devices. On the other hand, if the word vectors are reduced by removing some of their values to be able to run on low-resource devices, the amount of information they encode would also be reduced. The contributions presented in this thesis address each one of these limitations.

Contributions

This thesis presents two contributions to learn word embeddings which address the two aforementionned limitations of current word embeddings. The first contribution is a novel method that uses lexical dictionaries to extract semantic relations between words and incorporate this additional information in a model to learn word embeddings. The process of extracting information from dictionaries is based on the cooccurrence of words in their dictionary definitions and is completely automatic. The word embeddings learned with this method exhibit significant improvements in word semantic similarity tasks compared to other common word embeddings learning methods. This first contribution has been accepted and presented at the EMNLP 2017 conference [Tissier et al., 2017]. The second contribution is a method to transform common word embeddings (which usually use real values to encode information) into binary word vectors. The binary vectors are much smaller in memory than the real-valued ones (more than 30 times smaller) and have the advantage of accelerating vector operations, which are two of the main characteristics required to be used in downstream models on low-resource devices. This second contribution has been accepted and presented at the AAAI 2019 conference [Tissier et al., 2019].

Outline

This thesis is divided into three main parts: Chapter 1 and Chapter 2 introduce the main notions and concepts used throughout this thesis; Chapter 3 and Chapter 4 give an overview of the existing methods to learn word embeddings and reduce the size of representations; Chapter 5 and Chapter 6 present the two contributions of this thesis.

Chapter 1 introduces the main notions used in this thesis, the most important one being the definition of *word embeddings*. It also details tasks to evaluate the quality of word embeddings, which are used in the chapters of contributions (Chapter 5 and Chapter 6).

Chapter 2 presents the main concepts of machine learning: supervised learning, unsupervised learning and semi-supervised learning. It also explains how can machine learning

models learn from data and presents two models (neural network and autoencoder) that are commonly used in the word embeddings literature.

Chapter 3 is an overview of the most common existing methods to learn word embeddings. It details different models, from the first works on word embeddings to the latest ones which are, as we said before, complex and large. Some of the methods presented in this chapter are used in Chapter 5 to compare the performances of the word embeddings learned by the model of the first contribution.

Chapter 4 is an overview of existing methods to reduce the size of vector representations so they can be smaller in memory or speed up vector computations. Some of the methods presented in this chapter are used in Chapter 6 to compare the performances of the binary word vectors learned by the model of the second contribution.

Chapter 5 presents the first contribution of this thesis. It details how additional semantic information is extracted from the dictionary definitions of words and how this information is used to learn semantically-richer word embeddings. Several tasks like word semantic similarity or document classification are used to evaluate the quality of those word representations and their performances in downstream models.

Chapter 6 presents the second contribution of this thesis. It details the model and its architecture used to transform learned word embeddings into binary word vectors of any size. Several evaluations are performed to measure the quality of those binary vectors and an additional task is performed to evaluate the computational benefits of binary vectors for semantic similarity computations.

Part I

Background

Chapter 1

Preliminaries on Word Representations

1.1 Introduction

A language is a concept allowing one to express some meaning and communicate. For this thesis, we will consider languages that can be expressed by means of a text. A text can be viewed as a simple entity: a sequence of letters, numbers or symbols. But to work with texts and being able to create algorithms that can process them, we need to define some terms and notions. Let us define what are the main components of a text. The first unit that can be found in a text is defined by the notion of *words*.

Definition 1.1 (A word)**.** Let Σ be a a finite alphabet representing a non-empty finite set of symbols. A word is a finite sequence of symbols from Σ that carries a meaning.

In this thesis, we are only interested in texts of occidental languages like English. Any further mention to *a word* will refer to a sequence of letters (from the Roman alphabet), digits or symbols (like punctuation marks such as dash "-") that carries a meaning. Then, we can assemble words to form *sentences*.

Definition 1.2 (A sentence)**.** A sentence is a sequence of words, separated by spaces or punctuation symbols. The first word of a sentence starts with an uppercase letter and the last word of the sentence is followed by a period (.), a question mark (?) or an exclamation mark (!).

Sentences can be then be assembled to form *a text*.

Definition 1.3 (A text)**.** A text is a sequence of sentences.

Another important notion when working with text is the *vocabulary*.

Definition 1.4 (A vocabulary)**.** The vocabulary of a text is the set of all the unique words which occur in this text.

The length of a text should not be confused with the size of its vocabulary. Indeed, if a word appears several times in a text, it will be present only once in its vocabulary. Therefore, the length of a text is usually way larger than the size of its vocabulary. Let us take an example of two short texts to illustrate this. Table 1.1 indicates the vocabulary of two short texts.

- Text 1 has a vocabulary size of 17 words but a text length of 23 words.
- Text 2 has a vocabulary size of 17 words but a text length of 20 words.

	Text	Vocabulary
1	Add the milk to the flour and egg preparation and mix it. Do not mix too quickly to avoid lumps in the preparation.	Add, the, milk, to, flour, and, egg, preparation, mix, it, Do, not, too, quickly, avoid, lumps, in
2	Liverpool won the final against Tottenham. Salah scored after 1 minute and Origi scored another goal at the 87th minute.	Liverpool, won, the, final, against, Tottenham, Salah, scored, after, 1, minute, and, Origi, another, goal, at, 87th

Table 1.1: Examples of short texts and their vocabularies.

When we want to talk about the length of a text, we usually use the term "token" over the term "word". For example, we can say that Text 1 contains 23 tokens (because its text length is 23 words).

In the Natural Language Processing (NLP) domain, the final objective is usually to solve a given task. This includes for instance text classification which consists in finding the correct category of a text. Another task is information retrieval which focuses on extracting the correct sentences or documents the most relevant to a question. In order to show how the vocabulary of a text can be used to solve a given task, let us pick up the example of the text classification task. In this task, a naive way to find the correct category of a text would be to look at its tokens or at the content of its vocabulary and assign the category with the greatest amount of specific words (*e.g.* the words "won", "scored" and "goal" of Text 2 in Table 1.1 probably indicate that the text is about *sport*). However, it is difficult to compare two texts based solely on their vocabulary. Some words are different but have the same meaning, so two texts can have completely different vocabularies but still discuss the same subject or have the same information. The same goes for information retrieval, as a sentence can be the perfect answer to a question without using any of the words of the question. Therefore, although tokens and vocabularies are the essence of texts, there are not sufficient to represent the semantic information of the text or to process the content of the document. In order to solve this issue, we need additional concepts or representations that can be used to work with texts or documents. The next section presents some notions that are commonly used in the NLP literature.

1.2 From words to numerical representations

In the vocabulary of a text, each word only appears once even if it has several occurrences in the text. Therefore the vocabulary representation of a text is lacking some information, which is a shortcoming to be able to process it. If one text contains 20 times the word "football" and another text contains it only once, there is no way to know which text speaks the most about *football* by only looking at their vocabulary because both text vocabularies have only a single occurrence of "football". One solution to this problem is the *Bag-of-Words (BoW)* representation [Lang, 1995, Joachims, 1997].

Definition 1.5 (Bag-of-Words)**.** The Bag-of-Words representation of a text is the set of all the unique words within it, and the associated number of occurrence of each word.

For the two texts of Table 1.1, their Bag-of-Words are:

- Text 1:
  { Add: 1, the: 3, milk: 1, to: 2, flour: 1, and: 2, egg: 1, preparation: 2, mix: 2, it: 1, Do: 1, not: 1, too: 1, quickly: 1, avoid: 1, lumps: 1, in: 1 }

- Text 2:
  { Liverpool: 1, won: 1, the: 2, final: 1, against: 1, Tottenham: 1, Salah: 1, scored: 2, after: 1, 1: 1, minute: 2, and: 1, Origi: 1, another: 1, goal: 1, at: 1, 87th: 1 }

With Bag-of-Words representations, one can tell if among two texts using the same words, one contains more terms related to a subject. But solving this problem also introduces another one. In a text, most of the words are irrelevant to understand its topic. For example, in Text 2, the words "another" or "after" are less important than "scored" or "goal" to know that the text deals about *football*. Moreover, if a word appears in a lot of texts, then it is probably not discriminative to know the topic of a text. For instance, the word "minute" in Text 2 can also be found in other cooking recipes (which is the topic of Text 1) so this word is not indicative to differentiate the two different topics. To take into account the fact that less important words are usually way more frequent than more specific ones (it has been shown empirically that the frequency of words in a text follows a Zipf's law) and therefore that the number of occurrences of a word is not an indicative information, the values in BoW need to be replaced. The values need to measure the importance of a term in a text, but also among a set of texts. Words that are frequent among all the texts should have a smaller value because they are not discriminative. On the other hand, less frequent words should have a higher value because they better characterize a text. One solution is to replace the number of occurrences by the *TF-IDF value*.

Definition 1.6 (TF-IDF)**.** The TF-IDF of a word is a statistical value that indicates the importance of a word within a text among a collection of texts. It is defined as:

$$\text{TF-IDF}(word, text) = tf_{word,text} \times \log(\frac{N}{D_{word}}) \tag{1.1}$$

where $tf_{word,text}$ is the number of occurrences of $word$ in $text$, N is the total number of texts and D_{word} is the number of texts which contain $word$. With the same texts as in Table 1.1, we have:

- TF-IDF("the", Text 1) $= 3 \times \log(\frac{2}{2}) = 0$
- TF-IDF("the", Text 2) $= 2 \times \log(\frac{2}{2}) = 0$
- TF-IDF("milk", Text 1) $= 1 \times \log(\frac{2}{1}) = 0.693$
- TF-IDF("scored", Text 2) $= 2 \times \log(\frac{2}{1}) = 1.39$

Now that we have introduced the concept of TF-IDF, we can have a representation for each document based on the words it contains and the importance of each word. This type of representation has been proved to be useful for information retrieval [Ramos et al., 2003] or document classification [Zhang et al., 2011]. However, this representation has some severe limitations:

1. In order to compute the similarity of two given texts or to find the most specific words in a BoW representation, one has to go through each element and compare the words or their value. This becomes a problem when the size of the vocabulary is in the order of thousands and the number of texts reach several millions.

2. These representations can be viewed as *manually* generated. While the process of finding the unique words and counting their number of occurrences can be automated, the definition used to compute the weights (*i.e.* the values) in the BoW representations are defined by a human, which implies the need of an expert who knows what values would be useful for processing texts.

3. The BoW representation has zero understanding of the semantic and syntactic properties of words. Indeed, with the BoW representation, there is no way to know that "car" and "vehicle" are related words, unless a human annotator has explicitely indicated it (which brings us to point 2.).

To overcome these limitations, a solution is to find another way to represent texts, sentences and words. Moreover, with the increasing power of computers and the development of new models, these representations should have some properties to perform computations on them. Therefore, this brings **the need a numerical representation of the language**, which is **capable of encoding the linguistic information** inside it.

1.3 Word vectors

In classic Machine Learning (this notion will be defined in more details in Chapter 2), a standard approach consists in encoding information as numerical vectors. A vector can be viewed as an array of numerical values. For example, if a person is 33 years old, has 2 cars and lives in a 1000 square feet house, then it can be represented by the vector: $[33, 2, 1000]$. The values in the vectors are the characteristics of this person. Following this idea, NLP scientists have proposed to represent words of a language also as arrays of values. This notion of arrays of values comes from a mathematical background, so let us first define some concepts we will use throughout this thesis.

Definition 1.7 (A vector space, simplified version)**.** A vector space $\mathcal{X}$ over $\mathbb{R}$ is a collection of vectors and a set of two operations with specific properties which can be used on vectors:

1. vector addition: adding two vectors $\mathbf{u}, \mathbf{v} \in \mathcal{X}$ produces another vector which is also in $\mathcal{X}$.

2. scalar multiplication: multiplying a vector $\mathbf{u} \in \mathcal{X}$ with a scalar $\lambda \in \mathbb{R}$ produces another vector which also belongs to $\mathcal{X}$.

A *vector space* is a generic concept and it can be composed of different kind of objects like matrices or functions (but all the elements of a vector space must be of the same nature). For the rest of this thesis, we will consider vector spaces which are either $\mathbb{R}^d$ or a subspace of it. Another important notion is the concept of *vectors* and more precisely *d-dimensional* vectors.

Definition 1.8 (A vector)**.** A vector is an element of a vector space $\mathcal{X}$. In this thesis, we will consider vectors of $\mathbb{R}^d$ (or vectors from $\mathcal{X} \subseteq \mathbb{R}^d$). In this case, we say that $\mathbf{v} \in \mathbb{R}^d$ is a *d-dimensional* vector because it is composed of a sequence of d values, each one belonging to $\mathbb{R}$. From a practical point of view, we will write the values of a *d-dimensional* vector as $[\mathbf{v}_{[1]},\ \mathbf{v}_{[2]},\ ...,\ \mathbf{v}_{[d]}]$.

In classic Machine Learning, vectors are used as numerical representations of data. In the small example above, the vector $[33, 2, 1000] \in \mathbb{R}^3$ is the numerical representation of the person. In natural language processing, the problem is to find a numerical representation of the language which encodes linguistic information. As we have seen in Section 1.1, the smallest units of meaning in a text are words, so first we need to find a numerical representation for each word of a vocabulary. A naive way to find such representations is to associate a *one-hot vector* to each word. In Table 1.2, we have associated a one-hot vector to each word of a simplified vocabulary (which comes from Table 1.1). In one-hot vectors, all values are 0 except for one value which is equal to 1. The length of one-hot vectors is the same as the size of the vocabulary, which allows to associate a vector different from the others to each word by placing the 1 at a position which is specific to each word.

	Word	One-hot vector
1	Add	{1, 0, 0, 0, 0, 0, 0, 0, 0, 0}
2	the	{0, 1, 0, 0, 0, 0, 0, 0, 0, 0}
3	milk	{0, 0, 1, 0, 0, 0, 0, 0, 0, 0}
4	to	{0, 0, 0, 1, 0, 0, 0, 0, 0, 0}
5	flour	{0, 0, 0, 0, 1, 0, 0, 0, 0, 0}
6	and	{0, 0, 0, 0, 0, 1, 0, 0, 0, 0}
7	egg	{0, 0, 0, 0, 0, 0, 1, 0, 0, 0}
8	preparation	{0, 0, 0, 0, 0, 0, 0, 1, 0, 0}
9	mix	{0, 0, 0, 0, 0, 0, 0, 0, 1, 0}
10	it	{0, 0, 0, 0, 0, 0, 0, 0, 0, 1}

Table 1.2: Examples of one-hot vectors associated to each word of a small vocabulary (10 words). Vectors have 10 dimensions because the vocabulary is composed of 10 words.

More formally, one-hot vectors are defined as follows.

Definition 1.9 (A one-hot vector)**.** A one-hot vector of d dimensions is an element $\mathbf{v}$ of the vector space $\mathbb{B}^d \subseteq \mathbb{R}^d$ where $\mathbb{B} = \{0, 1\}$ and:

$$\sum_{i=1}^{d} \mathbf{v}_{[i]} = 1 \tag{1.2}$$

As said before, the dimension d is set to be equal to the size of the vocabulary. In the rest of this thesis, we will write **onehot(k)** to identify the one-hot vector where the k-th value is the one equal to 1. Then we can associate a unique vector to each one of the words of the vocabulary based on their position in this vocabulary. For example, in Table 1.2, the word "preparation" is at the 8-th position in the vocabulary, so it is associated with the vector $\mathbf{onehot}(\mathbf{8}) = \{0, 0, 0, 0, 0, 0, 0, 1, 0, 0\}$. Each word will therefore have a different vector, because $\mathbf{onehot}(\mathbf{i}) \neq \mathbf{onehot}(\mathbf{j})$ if $i \neq j$.

BoW representations have some limitations, one of them being that it is not straightforward to compute the similarity of two different texts with their BoW representations. Indeed, words in BoW of two texts are not necessarily the same so one has to compare words one by one to know what are the common ones. When the words or sentences of a document or the document itself are represented with elements of a vector space, like one-hot vectors which are elements of a boolean vector space, the mathematical properties of vector spaces allow one to use vector operations and similarity functions defined over the vector spaces. The mathematical properties of vector spaces solve some issues raised by the BoW representations, mainly the computation of the distance between two representations to compare them (which can be calculated with a function defined over the vector space like the Euclidean distance for $\mathbb{R}^d$). Moreover, one-hot vectors also solve the BoW representations issue of handcrafted vector values like with TF-IDF because values in one-hot vectors are independent of the properties of the words like its number of occurrences in texts. However, one-hot vectors do not solve the problem of encoding linguistic information. Indeed, there is no way to tell if two words are related or not by looking at their one-hot vector representations, because the position of the value 1 does not depend on the properties of words but on their position in the vocabulary, which is not representative of their linguistic properties. Furthermore, the length of one-hot vectors can also become a problem as their length should be the same as the size of the vocabulary, which can be in the order of millions.

A better numerical representation of words than one-hot vectors is to associate each word with a vector of real values, *i.e.* associating it with a vector $\mathbf{v} \in \mathbb{R}^d$. This introduces the notion of *word embeddings*.

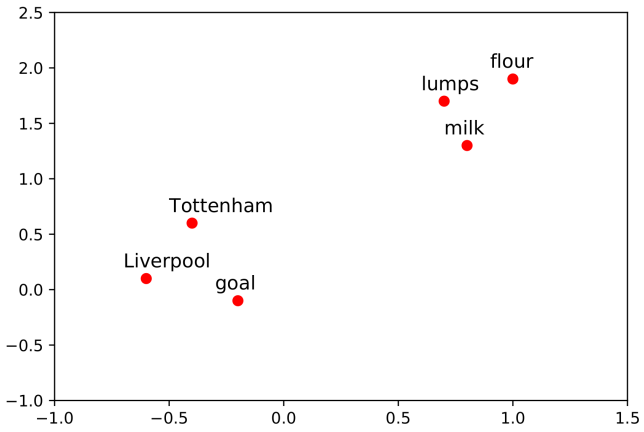
Definition 1.10 (Word embeddings)**.** The embedding of a word w is a vector $\mathbf{v}_w \in \mathbb{R}^d$ where d is the dimension of the embedding. The embedding matrix $\mathbf{M} \in \mathbb{R}^{|\mathcal{V}| \times d}$ (where $|\mathcal{V}|$ is the size of the vocabulary $\mathcal{V}$) is a matrix composed of the stacked up embeddings of all the words of the vocabulary.

This real-valued numerical representation addresses the first two of the three issues of BoW representations: (i) a representation that can be used to compute vector operations like similarity and distances calculations; (ii) it does not need handcrafted expressions to be defined. However, for the last of the three issues (encoding linguistic information), we need to add more constraints. Indeed, if the values of the embeddings are chosen randomly for each word, then there is no way to tell if two words are related or not because the distance between their vectors would also be random. Therefore the values in the embeddings have to be carefully designed to reflect the linguistic properties of words.

As an example, a common practice in the NLP literature [Mikolov et al., 2013a, Pennington et al., 2014] is to set the values of the vectors so that words that are related or have similar meanings also have similar word embeddings. In Figure 1.1a, word embeddings of 2 dimensions (*i.e.* vectors of 2 values) are reported for six words of the vocabularies from Text 1 and Text 2 of Table 1.1. The embeddings are plotted on a 2D plan in Figure 1.1b with the respective word they represent. The words "flour" and "milk" do not have the same meaning but they belong to the same semantic field (cooking ingredients) so their values are similar. The word "lumps" also have similar values because it is related to the word "flour". However, the words "Liverpool" and "Tottenham" are far from the words "flour" and "milk" on the 2D plan and their vector values are also quite different because they are not related to the semantic field of cooking and food. The word "goal" is closer to the two cities (Euclidean distance between the vectors of "goal" and "Liverpool" is 0.447; distance between the vectors of "goal" and "flour" is 2.332) because it is more related to the football teams of the two cities than to some cooking ingredients.

milk →	0.8	1.3
flour →	1.0	1.9
lumps →	0.7	1.7
Liverpool →	-0.6	0.1
Tottenham →	-0.4	0.6
goal →	-0.2	-0.1

(a) Values of 2-dimensional word embeddings of some words.



(b) Visual representation on a 2D plan of the word embeddings of some words.

Figure 1.1: Values (on left subfigure) and visual representations (on right subfigure) of 2-dimensional word embeddings of 6 words. Values of the vectors have been chosen so that words that are related have similar values and thus are close in the 2D plan representation.

In this section, we have introduced basic ways to encode words with numerical representations which are commonly used in NLP models. However, there exist others ways to encode words like with word-cluster representations [Bekkerman et al., 2003] or using n-grams feature vectors [Cavnar, 1995]. We have seen that word embeddings have the ability to encode linguistic information, which is a crucial lack of one-hot vector representations. Other types of representations for texts (like sentence or document representations) can be computed either by concatenating or by averaging the embeddings of the words that compose it. The vector space $\mathbb{R}^d$ to which the word embeddings belong have some mathematical properties that allow one to perform vector operations like summing or computing the distance between two vectors. However, the vector space $\mathbb{R}^d$ does not have a finite size and the number of possible values for each word embeddings is infinite. This raises another question: among several word embedding representations, how to compare them and select the most appropriate one for our final goal, that is to solve a NLP task?

1.4 Evaluation of word embeddings

The final objective in NLP is to solve tasks such as text classification, machine translation or information retrieval. Learning word embeddings is not a part of this final objective but rather an intermediate step to obtain semantic-preserving word representations that can be leveraged to solve downstream NLP tasks. Indeed, if the representations are able to better capture linguistic information into word embeddings, then downstream NLP models which use them to solve tasks have access to more knowledge to make predictions, which increases the performances on tasks [Kim et al., 2016, Bojanowski et al., 2017]. Word embeddings can either be evaluated when they are created or when they are used into other models, which separates the evaluations into two different categories:

1. How much linguistic information is encoded into word embeddings? This evaluation depends only on the values in the embeddings, not on their use in downstream NLP tasks. We call it the **intrinsic evaluation**.

2. How helpful are word embeddings when they are used to solve an NLP task? We call this the **extrinsic evaluation**.

While these two types of evaluation are related, the performances of word embeddings on one of them do not reflect the performances on the other one. Chiu et al. [Chiu et al., 2016] showed that scores on datasets used for intrinsic evaluation are, for the majority, negatively correlated with scores on tasks of extrinsic evaluation which means that both types of evaluation are needed to evaluate word embeddings. In this section, we present the most common tasks and datasets used for intrinsic evaluation in Subsection 1.4.2 and for extrinsic evaluation in Subsection 1.4.3. Most of these tasks require to compute the similarity or the distance between pairs of vectors, so let us first define in Subsection 1.4.1 what are the main similarity or distance functions used in the literature.

1.4.1 Similarity and distance functions for vectors

Cosine similarity

Cosine similarity is computed on real-valued vectors from $\mathbb{R}^d$. It is the most commonly used similarity function in NLP models, and is also the most used one in the context of this thesis.

Definition 1.11 (Cosine similarity)**.** The cosine similarity between two d-dimensional vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$ is defined as:

$$\text{cosine}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \, \|\mathbf{v}\|} = \frac{\sum_{i=1}^{d} \mathbf{u}_{[i]} \times \mathbf{v}_{[i]}}{\sqrt{\sum_{i=1}^{d} \mathbf{u}_{[i]}^2} \times \sqrt{\sum_{i=1}^{d} \mathbf{v}_{[i]}^2}} \tag{1.3}$$

Jaccard similarity and Jaccard distance

Jaccard similarity is computed on binary vectors from $\mathbb{B}^d$ and has a value between 0 and 1.

Definition 1.12 (Jaccard similarity)**.** The Jaccard similarity between two binary d-dimensional vectors $\mathbf{u}, \mathbf{v} \in \mathbb{B}^d$ is defined as:

$$\text{Jaccard}_{sim}(\mathbf{u}, \mathbf{v}) = \frac{M_{11}}{M_{01} + M_{10} + M_{11}} \tag{1.4}$$

where M_{11} is the number of dimensions simultaneously set to 1 in $\mathbf{u}$ and $\mathbf{v}$ and M_{01} (resp. M_{10}) is the number of dimensions set to 0 (resp. 1) in $\mathbf{u}$ and set to 1 (resp. 0) in $\mathbf{v}$. Jaccard distance is closely related and is defined as:

$$\text{Jaccard}_{dist}(\mathbf{u}, \mathbf{v}) = 1 - \text{Jaccard}_{sim}(\mathbf{u}, \mathbf{v}) \tag{1.5}$$

Euclidean distance

Euclidean distance is not a similarity function but a distance function. It measures how close (or how far) two vectors are.

Definition 1.13 (Euclidean distance)**.** The Euclidean distance between two d-dimensional vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$ is defined as:

$$\text{Euclidean}_{dist}(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{d} (\mathbf{u}_{[i]} - \mathbf{v}_{[i]})^2} \tag{1.6}$$

L_1 distance

L_1 distance is also a distance function like the Euclidean distance. It can be applied to real-valued or binary vectors. However, this function is not differentiable which makes it hard to use in NLP models.

Definition 1.14 (L_1 distance)**.** L_1 distance between two d-dimensional vectors either $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$ or $\mathbf{u}, \mathbf{v} \in \mathbb{B}^d$ is defined as:

$$\text{L}_1(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{d} |\mathbf{u}_{[i]} - \mathbf{v}_{[i]}|} \tag{1.7}$$

1.4.2 Intrinsic evaluation

With the defined similarity and distance functions described in the previous subsection, word embeddings can be evaluated in intrinsic evaluation tasks to measure how much linguistic information is encoded into the vectors. The two main existing intrinsic evaluation tasks are word semantic similarity and word analogy.

Word semantic similarity

As explained in the previous section, word embeddings should encode linguistic information. Words with similar meaning should have close vector representations. The word semantic similarity task evaluates how well the embeddings reflect the semantic similarity of words. It is based on datasets composed of pairs of words. Many datasets exist, the most commonly used ones being WordSim-353 [Finkelstein et al., 2001] and Rare Words (RW) [Luong et al., 2013]. A more extensive list of available datasets will be presented in the contributions of this thesis (Chapter 5 and Chapter 6) where they are used to evaluate the word embeddings learned by the models proposed in the contributions. For each pair of words of the datasets, a few human annotators have been asked to rate the similarity between the words on a scale of 0 to 10. Scores from all the annotators are validated with an inter-annotator agreement threshold and are then averaged for each pair. In Table 1.3, some examples of such pairs and scores are reported.

Pairs of words	Human score
Maradona – football	8.62
month – hotel	1.81
problem – airport	2.38
money – cash	9.15

Table 1.3: Examples of pairs of words and their similarity scores rated by human annotators on a scale of 0 to 10, from the WordSim-353 dataset [Finkelstein et al., 2001].

Word	Embedding			
Maradona	[0.73,	0.55,	0.12,	0.33]
football	[0.62,	0.42,	0.07,	0.79]
month	[0.01,	-0.22,	0.76,	-0.94]
hotel	[0.25,	0.62,	0.37,	0.23]
problem	[-0.87,	-0.83,	-0.97,	0.89]
airport	[0.22,	-0.18,	-0.19,	-0.08]
money	[-0.09,	0.85,	0.20,	0.35]
cash	[0.74,	0.72,	-0.30,	0.26]

(a) Examples of vector values associated to some words of the WordSim-353 dataset [Finkelstein et al., 2001].

Pairs of words	Human score	Cosine similarity
Maradona – football	8.62	0.892
month – hotel	1.81	-0.070
problem – airport	2.38	0.114
money – cash	9.15	0.551

(b) Examples of pairs of words and their cosine similarity scores computed with the cosine similarity between their respective embeddings.

Table 1.4: Cosine similarity scores (Table 1.4b) computed with the vector values (Table 1.4a) of the respective embedding of the words of each pair.

The ideal situation would be when the similarity score between the respective embeddings of the words of a pair (like "money"–"cash") has the exact same value as its human score. However, this is very unlikely to happen for all the pairs of a dataset (there are typically more than 100 pairs in those datasets). Therefore, we need to evaluate how "close" word embeddings are from the human judgement. For each pair of words, the cosine similarity between their respective word embeddings is computed with the values of the vectors (reported in Table 1.4a). This adds a new column in the table of scores (see Table 1.4b).

Comparing the similarity scores of pairs of vectors to the values assigned by human annotators is not a good solution mainly because the similarity scores are usually small (between -1 and 1 for the cosine similarity function) whereas the values of human annotations can be much larger (up to 10 in the WordSim-353 dataset). Instead of directly comparing the vector similarities and the human scores, a better solution is to evaluate if the order of values is preserved. Indeed, if a pair has a high score assigned by annotators and a high score obtained by computing the cosine similarity of the vectors, then the embeddings are reflecting the human judgement (the words are strongly related). To evaluate if the order is preserved, the common method is to use the Spearman's rank correlation coefficient [Spearman, 1904]. It measures the correlation between the ranks of two lists of values. It is defined as:

$$\rho = 1 - \frac{6 \times \sum_{i=1}^{n} d_i^2}{n \times (n^2 - 1)} \tag{1.8}$$

where n is the number of values in the two lists and d_i are the differences of ranks in the two lists for each value. The Spearman's rank correlation coefficient is a real value in the [-1, 1] range. A coefficient of 1 (resp. -1) indicates that the values of the two lists are ranked in the same order (resp. in reverse order).

Let us illustrate the computation of this coefficient with an example. Table 1.5 is almost the same as Table 1.4b except it has three new columns:

- Rank_{Human} is the rank each value of "Human score" would have if the list was sorted in decreasing order: 9.15 is the highest value of the column "Human score" so its rank is 1; 8.62 is the second highest value so its rank is 2, etc.
- Rank_{Cosine} is the rank each value of "Cosine similarity" would have if the list was sorted in decreasing order: 0.892 is the highest value of the column "Cosine similarity" so its rank is 1; 0.551 is the second highest value so its rank is 2, etc.
- Rank difference (d_i) is the difference of Rank_{Human} and Rank_{Cosine} for each value.

Pairs of words	Human score	Rank_{Human}	Cosine similarity	Rank_{Cosine}	Rank difference (d_i)
Maradona – football	8.62	2	0.892	1	2 - 1 = 1
month – hotel	1.81	4	-0.070	4	4 - 4 = 0
problem – airport	2.38	3	0.114	3	3 - 3 = 0
money – cash	9.15	1	0.551	2	1 - 2 = -1

Table 1.5: Rank differences used to compute the Spearman's rank correlation coefficient.

Given the formula of the Spearman's rank correlation coefficient (Equation 1.8), we have:

$$\rho = 1 - \frac{6 \times (1^2 + 0^2 + 0^2 + (\text{-}1)^2)}{4 \times (4^2 - 1)} = 1 - \frac{6 \times (1 + 1)}{4 \times (16 - 1)} = 1 - \frac{12}{60} = 1 - 0.2 = 0.8$$

A Spearman's rank correlation coefficient close to 1 indicates that the embeddings are close to the human semantic knowledge because pairs of words evaluated as highly related by humans are also highly related with the cosine similarity of their embeddings values. Similarly, pairs of words rated as not related by humans also have word embeddings values reflecting that they are not related (low values for their cosine similarity). The Spearman's coefficient for this small example is 0.8, which indicates that the word embeddings values are pretty good to encode semantic (dis)similarity of words.

Word analogy

In a language, some relations or notions between words can be common. For example, the words "Berlin" and "Germany" can be linked by the relation *capital of*. The same goes for the words "Budapest" and "Hungary" and many others. In 2013, Mikolov et al. [Mikolov et al., 2013a] analyzed some word embeddings and saw that they are capable of encoding such relations with vector operations. Relations between words such as *capital of* can also be encoded as vectors, and adding it to the embedding of a capital city can produce the embedding of the related country. In Figure 1.2, the black arrows represent the *analogy*, *i.e.* the vector one needs to add to a capital city to obtain the embedding of the related country. These vectors are almost identical for the three examples, which suggests that word embeddings are able to encode the relation between a country and its capital city.

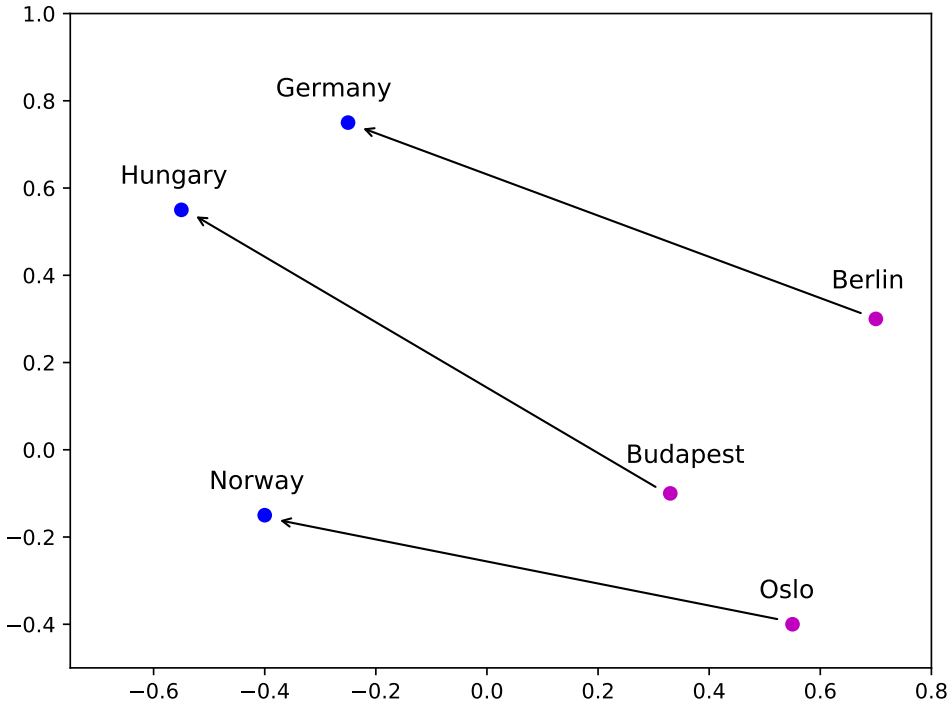


Figure 1.2: Representation on a 2D plan of word embeddings and the analogy *capital of*.

Mikolov et al. introduced a new task to evaluate how well word embeddings can predict linguistic relations like "a is to b as c is to d". Their datasets [Mikolov et al., 2013a] are composed of lines containing four words, such as "Berlin Germany Budapest Hungary". The task consists in computing the vector $\mathbf{v} = \mathbf{v}_{Germany} - \mathbf{v}_{Berlin} + \mathbf{v}_{Budapest}$ and see if its closest vector is the embedding of the word "Hungary". If so, the analogy is said to be correct. The final evaluation score for this task is the percentage of correctly predicted analogies. Analogies are separated into two groups: one focuses on semantic analogies and the other one on syntactic analogies, which are respectively split into 5 and 9 categories. Table 1.6 reports the different categories and an example for each one of them.

	Category	Example (a b c d)			
Semantic					
	capital-common-countries	Berlin	Germany	Budapest	Hungary
	capital-world	Tehran	Iran	Islamabad	Pakistan
	currency	Europe	euro	Japan	yen
	city-in-state	Phoenix	Arizona	Honolulu	Hawaii
	family	king	queen	man	woman
Syntactic					
	adjective-to-adverb	calm	calmly	swift	swiftly
	opposite	informed	uninformed	known	unknown
	comparative	cold	colder	large	larger
	superlative	cold	coldest	long	longest
	present-participle	debug	debugging	read	reading
	nationality-adjective	Colombia	Colombian	Malta	Maltese
	past-tense	flying	flew	going	went
	plural	snake	snakes	pineapple	pineapples
	plural-verbs	think	thinks	swim	swims

Table 1.6: Categories and examples of semantic and syntactic word analogies taken from the datasets of [Mikolov et al., 2013a].

1.4.3 Extrinsic evaluation

The two intrinsic evaluations (word semantic similarity and word analogy tasks) described in Subsection 1.4.2 measure how well the linguistic information is encoded into word embeddings. But word embeddings are usually learned for another purpose, which is to be used as a language representation to solve a NLP task. After being learned, word embeddings are then fed as the input of other mathematical models that perform computations with them to produce representations for sentences or texts, and then achieve a specific goal like predicting the category of a text or translating a sentence. As the performance of the model to solve the task is dependent on the amount and the quality of linguistic information encoded into the word embeddings, it can be used to compare different word embeddings (word embeddings with more encoded information gives better results on the downstream tasks). The variety of tasks using word embeddings is large, like name entity recognition [Turian et al., 2010, Pennington et al., 2014], question answering [Bordes et al., 2014] or machine translation [Zou et al., 2013, Lample et al., 2018]. Presenting an exhaustive list of all the possible tasks would be too long so we mainly focus on describing tasks related to document classification, which are the tasks used in the evaluation of the word embedding learned in the contributions of this thesis (see Chapter 5 and Chapter 6).

Text classification

This task aims to assign the category of a text. The text can come from different sources (a Wikipedia page, a newspaper, a blog article etc.) and the category can be of different forms (a class like "economy" or "politics" for a newspaper article, a hierarchical category like "musician" or "vertebrate animal" for a Wikipedia page, etc.). For this task, a model combines the embeddings of the words from the text to predict the correct category. The model is usually a neural network (neural networks will be described in Subsection 2.4.1 of Chapter 2). After the model has learned how to predict the correct category given a certain amount of examples (composed of texts and their associated category), it is evaluated on a set of texts not seen during the learning phase. The percentage of correctly predicted categories gives the score of the word embeddings for this task.

Sentiment analysis

This task is very similar to the text classification task. In this task, the objective is to predict a value given an input text, most often a short text like a couple of sentences. We find in this task different examples like predicting the number of stars (between 1 and 5) a customer has given to a product based on the written review of the product (multi-class classification tasks) or predicting if a comment about a service is good or bad (positive and negative *i.e.* binary classification tasks). Like with the text classification task, another mathematical model is fed with word embeddings and then learns how to predict the correct value based on the input text. The final score for this task is the percentage of correctly predicted values on a set of unseen texts.

GLUE/SuperGLUE

Word embeddings are used in downstream models to solve NLP tasks. But as the recent downstream models become more and more complex [Peters et al., 2018, Devlin et al., 2019], they are able to process and understand more complex relations between words and sentences. Tasks like document classifications or sentiment analysis are not sufficient to evaluate the performances of the representations of words learned by these complex models (because their scores on those tasks reach the human baseline). Wang et al. [Wang et al., 2019a] have created `GLUE` (**G**eneral **L**anguage **U**nderstanding **E**valuation benchmark), a resource of datasets to evaluate the performances of models on 9 downstream tasks. `GLUE` also proposes tools to standardize the evaluation of models and a leaderboard[1] to compare the models proposed by different people. The nine tasks in `GLUE` are:

- `CoLA`: a set of sequences of English words from books and journal articles where the objective of the task is to predict if the sequence is grammaticaly correct or not.
- `SST-2`: a set of sentences from movie reviews. The objective is to predict if the review is positive or negative (like in the sentiment analysis task).
- `MRPC`: a set of pairs of sentences extracted from online news sources. The objective of the task is to predict if the two sentences are semantically equivalent or not.
- `QQP`: a set of pairs of questions extracted from the question-answering website Quora. Like in `MRPC`, the objective of the task is to predict if the two questions are semantically equivalent or not.
- `STS-B`: a set of pairs of sentences extracted from news headlines or video/images captions. The objective is to predict the similarity of the two sentences on a scale of 1 to 5 (the correct value to predict has been evaluated by human annotators).
- `MNLI`: a set of pairs of sentences (an hypothesis and a premise) from different sources like fictions or government reports. The objective of the tasks is to predict if the premise entails or contradicts the hypothesis, or if it is neutral to it.
- `QNLI`: a set of (paragraph, question) pairs where the objective is to extract the sentence from the paragraph that answers the question.
- `RTE`: a set of pairs of sentences (an hypothesis and a premise, like in the `MNLI` task) extracted from news and Wikipedia. The objective is the same as in `MNLI`.

[1] `https://gluebenchmark.com/leaderboard`

- `WNLI`: a set of pairs of sequences where one of the sentence use the specific name of something or someone (like the firstname of a person or a brand name) and the second sentence uses a pronoun (like he, she, it, etc.). The objective of the task is to predict if the sentence with the pronoun is an entailment or not of the first sentence.

Recent models like `ELMo` [Peters et al., 2018] or `BERT` [Devlin et al., 2019] are achieving near human performances in the `GLUE` benchmark dataset. Even more recent models [Raffel et al., 2019, Wang et al., 2020] surpass the human baseline of `GLUE`. Wang et al. [Wang et al., 2019b] introduced a new benchmark dataset named `SuperGLUE` which contains 8 different tasks, more difficult than the ones in the `GLUE` benchmark dataset. `SuperGlue` also proposes tools to standardize the evaluation on the tasks and a leaderboard[2] to compare different models. These evaluation benchmarks are mainly used to evaluate downstream complex models that use representations of the language like word embeddings, not really the performances of the embeddings themselves. Furthermore, they are very recent (2019a for `GLUE`, 2019b for `SuperGLUE`). Word embeddings proposed in the contributions of this thesis (in Chapter 5 and Chapter 6) have therefore not been evaluated with these benchmark datasets but on classic tasks like document classification or sentiment analysis.

1.5 Conclusion

In this chapter, we have presented the notions which will be used in this thesis, mainly representations of words and how to evaluate those representations.

To process texts in order to solve tasks like machine translation or document classification, models need to have a numerical representation of texts or a numerical representation of smaller units of meaning like words to be able to perform some computations. Early NLP models were only using words and the number of occurrences of words in texts. This kind of representations, named *Bag-of-Words*, showed some limitations in the amount of encoded linguistic information as well as for the computations on those representations. Several methods like *TF-IDF* or one-hot vectors have been proposed to overcome these limitations but they are still lacking some features to fully solve all the limitations. A method commonly used in the literature to represent linguistic information contained in texts are *word embeddings*. Word embeddings associate a real-valued vector $\mathbf{v} \in \mathbb{R}^d$ to each word of the vocabulary of a corpus. The vectors have many dimensions (typically in the [100, 300] range) and values of vectors are set to represent linguistic properties of words: for example, words with similar meaning have similar vector values.

In this chapter, we also have presented different methods to evaluate word embeddings. The methods can be separated into two categories: intrinsic evaluations measure how much linguistic information is encoded into the vectors with tasks like word semantic similarity or word analogy; extrinsic evaluations measure how useful the embeddings are when they are used into downtream models. Tasks like document classification or sentiment analysis are commonly used for extrinsic evaluations.

While we have introduced the notion of word embeddings and different methods to evaluate them, we have not explained how the values of vectors are set or how downstream models can use them to solve tasks. For a small vocabulary and a small number of dimensions, vector values can be chose manually. But it becomes a problem when the number of words in the vocabulary or the number of dimensions are large. We need tools to let a machine learn word embeddings automatically without having to explicitely tell it linguistic rules. In the next chapter (Chapter 2), we will describe *Machine Learning* (a general method used so machines can learn automatically a solution to a problem) with a

[2] `https://super.gluebenchmark.com/leaderboard`

focus on how to apply Machine Learning to learn word embeddings and a presentation of different models we will use throughout this thesis to learn word embeddings.

Chapter 2

Machine Learning for Word Embeddings

2.1 Introduction

In Chapter 1, we have defined the main notions we will use thoughout this thesis and we have presented common ways to numerically represent linguistic information. A common method used in the literature is *word embeddings*: it consists of associating a real-valued vector to each word of a vocabulary. This vectorial representation of words benefits from the mathematical properties of vector spaces. It allows one to perform operations on vectors and compute distances between them, which are the main requirements for downstream models to be able to use word embeddings to solve NLP tasks. We have seen that word embeddings need to have vector values which reflect the linguistic properties of words so downstream models can have access to linguistic knowledge through the use of word embeddings, but we have not yet explained how the values of word embeddings are chosen neither how word embeddings are actually used to solve downstream NLP tasks.

In the early works on word representations, values of vectors were chosen based on linguistic and lexical properties of words [Osgood, 1964, Bierwisch, 1970] . For example, the first value of each word embedding could be an integer representing the nature of the word (1 for nouns, 2 for verbs, 3 for adjectives, etc.), the second value could be the ratio between the number of times a word is used as the subject and as a complement in a text, etc. While those choices are appropriate from a linguistic point of view because it makes sense to directly incorporate properties of words into their vector representations, it is not appropriate from a scaling point of view. Indeed, the number of words in a language is large (it can reach several millions when we consider that entities like cities, regions, brands, etc. can occur in a text and therefore can be an element of the vocabulary of a language), so the process of identifying and generating handcrafted linguistic properties of words to incorporate them into their word vector becomes a long and complex task. Moreover, this process requires the knowledge of linguistic experts, which is expensive and time consuming because it requires human manual annotations.

To overcome the problem of manual labor required by handcrafted features, NLP scientists sought to use methods from the field of *Machine Learning* (ML) to produce them automatically. Machine learning is a subfield of artificial intelligence that aims to create algorithms or statistical models which can identify and extract information from data without being explicitly programmed to. Machines use data to *learn* rules about the information hidden inside it. Machine learning algorithms and models are then used in different kinds of task like identifying the model of a car from a picture of it, detecting abnormal behavior of people in videos or predicting the power consumption of a city for the

next week. Advances in computing power available in machines have allowed to build more complex models over the years [Taigman et al., 2014, Silver et al., 2016, Devlin et al., 2019].

In this chapter, Section 2.2 presents the different types of machine learning algorithm that exist while Section 2.3 explains how can a machine *learn* from data. Lastly, Section 2.4 details some of the main machine learning models used for dealing with word embeddings. This section (Section 2.4) will also serve as a foundation for the rest of this thesis as the contributions described in this thesis use those machine learning models.

2.2 Supervised and unsupervised learning

As said in the previous section, machine learning algorithms use data to extract, by themselves, rules about the information contained inside data in order to solve a given problem. These algorithms are able to perform on their own the information extraction operation to solve the problem, without needing to follow a computer program that indicates precisely where is the information contained inside data which is relevant to the given task. However, some machine learning models need some help at the beginning of the extraction process to *learn*, *i.e.* to understand what information is required to solve the task and what rules need to be designed to extract it.

Let us take a small example to illustrate this. Let say we want to know if a mushroom is edible or poisonous based on a picture of it and that we want to use a machine learning algorithm to solve this problem. If enough pictures of mushrooms are given to the algorithm, it will be able to recognize some common patterns in certain types of mushrooms (the color of the cap, the spacing in the gill etc.) and it will be able to group together mushrooms with similar features. However, if no one has indicated for each picture (*i.e.* the data) if it is a poisonous or an edible mushroom, then the algorithm cannot guess the correct answer and therefore cannot differentiate edible from poisonous mushrooms. Thus, without any help, the machine learning algorithm cannot learn to correctly predict if a mushroom is edible or not. The same thing also happens in the NLP domain. For instance, if an algorithm has to predict the subject of a news article (like "sport", "economy" or "politics") and no one has helped the algorithm by associating articles with their subject, then the algorithm has no way to learn what are the common features of sport articles because it cannot know which articles are sport articles. The process of adding a hint to help the algorithm to differentiate edible from poisonous mushrooms is called *annotating* data. Each piece of data (*i.e.* each picture of a mushroom) is associated to a *label* (*i.e.* the hint) which indicates if the mushroom is edible or not. With this example, we can separate machine learning algorithms into two groups:

1. When the algorithm needs a hint or additional knowledge to be able to learn how to differentiate different categories of objects in data (like predicting the toxicity of a mushroom) we call it *supervised learning*.

2. When there are no hints available and the objective of the algorithm is to group together or to find common patterns and features in data, we call it *unsupervised learning*.

This section presents supervised learning in Subsection 2.2.1 and unsupervised learning in Subsection 2.2.2. Another type of machine learning algorithm called *semi-supervised learning* which lies between supervised and unsupervised learning is presented in Subsection 2.2.3.

2.2.1 Supervised learning

In supervised learning, each piece of data is annotated with one or several labels (in the mushroom example, the annotation to indicate if the mushroom is edible or poisonous is a single label). Machine learning algorithms learn how to use the features of data (*i.e.* the information contained inside it) to predict the corresponding label(s). Data can have many forms: an image, a video, a list of numerical values to indicate the properties of a person or an object, etc. Labels can also be of different types: an integer (*e.g.* when predicting an age), a word (*e.g.* when predicting the species of an animal), a real value (*e.g.* when predicting a temperature), etc. In machine learning, we use the notion of *input* to designate data given to algorithms and *output* to designate the prediction of algorithms [Ayodele, 2010]. In the case of supervised learning, the model learns how to map the input to the corresponding correct output. After the *learning phase* (also called the *training phase*) which consists in finding the mapping function that correctly predicts the output given the input, the algorithm is fed with new input data it has never seen (and for which we do not know the corresponding label) and uses what the model has learned to predict the output. This prediction step on new data is the *inference phase*. In the small mushroom example, this step consists in using the trained model to know if a mushroom is edible or not from a new picture of mushroom (*i.e.* a picture not used during the learning phase). We can formalize supervised learning with mathematical terms:

Definition 2.1 (Supervised learning)**.** Let $\mathcal{X} \subseteq \mathbb{R}^d$ be the vector space containing input data, $\mathcal{Y}$ the label space containing output labels and $\mathcal{S}$ a set of annotated examples of the form $(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ where $\mathbf{x}_i$ is the feature vector of the i-th example and y_i its corresponding output label. A supervised learning algorithm searches a function $f : \mathcal{X} \to \mathcal{Y}$ among the space of hypothesis $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$ which gives the same (or the closest) output value $f(\mathbf{x}_i)$ as y_i for all $(\mathbf{x}_i, y_i) \in \mathcal{S}$.

The label space $\mathcal{Y}$ depends on the task solved by the machine learning algorithm. Table 2.1 reports some examples of common label spaces used in supervised learning tasks.

Type of task	Label space	Example of a task
Binary classification	$\mathcal{Y} = \{0, 1\}$ or $\mathcal{Y} = \{-1, +1\}$	Predict if a mushroom is edible (+1) or not (-1).
Multi-class classification	$\mathcal{Y} = \{1, 2, \ \ldots \ , K\}$	Predict the species of an animal among K possibilities (1 = "dog"; 2 = "cat"; ...).
Regression	$\mathcal{Y} = \mathbb{R}$	Predict a temperature.

Table 2.1: Examples of label spaces for different supervised learning tasks.

Many NLP tasks can be solved with supervised learning. Here is a non-exhaustive list of examples:

- Text classification and sentiment analysis (described in Subsection 1.4.3 of Chapter 1) can be both solved with supervised learning. Input data are commonly vector representations of texts or sentences and output labels can either be a word to indicate the category of the text, or an integer to indicate the number of rating stars associated to a comment.
- Automatic translation is another example of task that can be solved with supervised learning. The machine learning algorithm is fed with a sentence from a specific

language, and the output is the sentence translated into another language. The algorithm learns how to map the two sentences together and therefore, learns how to translate sentences.

- In a textual entailment task (which consists in predicting if two sentences are related by a cause/consequence relation or if they are unrelated), input data are the two sentences (or the vector representations of sentences) and output labels are binary values that indicate if the two sentences are related or not.

2.2.2 Unsupervised learning

The main difference between supervised and unsupervised learning is that there are no labels in the case of unsupervised learning (or from a different point of view, because data do not have any annotations nor labels, we cannot use supervised learning algorithms so we have to rely on unsupervised learning methods). Two main tasks can be solved with unsupervised learning: clustering and data transformation. Let us describe and illustrate those tasks with NLP examples.

- **Clustering**. Clustering is the task of grouping similar objects into sets (called *clusters*) such that two objects from the same cluster are more similar to each other than to objects from another cluster. Clustering cannot be solved with supervised learning algorithms because there are no labels to indicate in which cluster each object should belong. Information contained inside data needs to be used to find which objects are similar together.

  Clustering can be used in NLP to analyze comments posted on social media platforms. While each user is unique, we can group together people with similar behavior because they either speak about the same subjects or have the same style of writing. This kind of data do not have any labels: there are no extra information in a comment to indicate in which group a user belongs nor to indicate how many groups may exist. Clustering can be used to group together comments and detect some common patterns between users.

- **Dimensionality reduction**. Dimensionality reduction is the task of transforming data to reduce the number of features or the number of dimensions in vectors. For some data, the number of features or dimensions can be very large (in the order of thousands) which makes them unsuitable for use in supervised learning algorithms. There are no labels or annotations to indicate which dimensions would be the most useful for a given task so unsupervised learning algorithms are used to find them automatically. Chapter 4 focuses on presenting the most common methods that exist to reduce the number of dimensions in word embeddings.

  In document classification, data can be expressed by vectors of thousands of dimensions with many values set to 0 and a few values set to 1 (we call them *sparse vectors*). Such vectors can be obtained by checking the presence/absence of many words in a news article or in a website. Large sparse vectors can be difficult to use in supervised learning models because of the numerous 0 so vectors are usually transformed into *dense vectors*, *i.e.* vectors of continuous real values with a smaller number of dimensions. There are no labels in this kind of data to indicate what are the transformation to apply, so unsupervised learning algorithms have to learn them by themselves.

2.2.3 Semi-supervised learning

Semi-supervised learning is another type of machine learning which is between supervised and unsupervised learning. In semi-supervised learning, only a small amount of data have labels; the majority of data is unlabeled. Semi-supervised learning algorithms use a combination of supervised and unsupervised algorithms to solve a given task. For example, one can use clustering to group together similar objects and then propagate the annotations of labeled objects of a cluster to unlabeled objects of the same cluster.

Semi-supervised learning is often used in NLP. As we have seen in Chapter 1, annotating each text or each word of a large vocabulary with its linguistic properties is a difficult and time-consuming task because it requires the knowledge of linguistic experts. The small amount of labeled data is used to either assume the properties of similar items or to propagate the annotations to unlabeled data. For the task of learning word embeddings which encode linguistic information, vector representations of words are learned with unsupervised learning algorithms which use a small amount of supervision with either a few number of annotated examples of similar words or with an hypothesis about the distribution of semantic knowledge in a language. Chapter 3 presents the most common methods to learn word embeddings; most of them are semi-supervised algorithms.

2.3 Training a machine learning model

Machine learning models learn to identify patterns and infomation inside data and in the case of supervised learning, learn how to predict an output label given input data. Models learn without being explicitly programmed to: information is usually hidden inside the numerical representation of data, which prevents developers to create specific rules that extract only some of the values in data and then combine them with a manually-written formula to make predictions. Yet, machine learning models are able to automatically extract useful information from data needed to solve a given task and learn how to combine these informative values to predict the correct output label. This automatic process of learning is the result of setting an objective for the algorithm (Subsection 2.3.1) and giving it tools to achieve it (Subsection 2.3.2).

2.3.1 Objective function

In supervised learning, since each data is associated with an output label, the goal of the algorithm is pretty intuitive: being able to predict the correct label for each example. We can modelize this goal in mathematical terms with the notion of a *loss function*.

Definition 2.2 (Loss function)**.** Let $\mathcal{X}$ be the vector space containing data, $\mathcal{Y}$ the label space, $\mathcal{S} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ a set of N annotated examples of the form $(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ where $\mathbf{x}_i$ is the feature vector of the i-th example and y_i its corresponding label and $f \in \mathcal{Y}^{\mathcal{X}}$ a mapping function learned by a supervised learning algorithm. A loss function is a function $C : \mathcal{Y}^{\mathcal{X}} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_+$ such that:

$$\forall i \in [1..N],\ C(f, \mathbf{x}_i, y_i) \geq 0 \tag{2.1}$$

The global loss of an algorithm is the average of the loss values on all the examples of $\mathcal{S}$:

$$\text{Global Loss } (f) = \frac{1}{N} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{S}} C(f, \mathbf{x}_i, y_i) = \frac{1}{N} \sum_{i=1}^{N} C(f, \mathbf{x}_i, y_i) \tag{2.2}$$

The loss function must be chosen such that for a given algorithm, the higher the number of incorrect predictions, the higher the value of its global loss. Here are some examples of loss functions commonly used to solve NLP tasks with supervised learning:

- **0/1 loss**. It is the most basic loss function. It returns a value of 0 when the prediction of the algorithm is correct, and a value of 1 otherwise. The global loss of an algorithm that uses this loss is its number of incorrect predictions.

$$\forall f \in \mathcal{Y}^{\mathcal{X}},\ \forall (\mathbf{x}_i, y_i) \in \mathcal{S},\quad C(f, \mathbf{x}_i, y_i) = \begin{cases} 0 & \text{if } f(\mathbf{x}_i) = y_i \\ 1 & \text{otherwise} \end{cases} \tag{2.3}$$

- **Cross-entropy loss (or log loss)**. This loss is mostly used in classification problems, either for binary [Nam et al., 2014] or multi-class classification problems [Kurata et al., 2016], where the outputs of the algorithm are the probabilities of an input instance to belong to each possible class. When the number of possible classes is K ($K = 2$ for binary classification), it is defined as:

$$\forall f \in \mathcal{Y}^{\mathcal{X}},\ \forall (\mathbf{x}_i, y_i) \in \mathcal{S},\quad C(f, \mathbf{x}_i, y_i) = -\sum_{k=1}^{K} y_{i_k} \log(f(\mathbf{x}_i)_k) \tag{2.4}$$

  where $f(\mathbf{x}_i)_k$ is the predicted probability that $\mathbf{x}_i$ belongs to the class k and y_{i_k} is either 1 or 0 depending on whether $\mathbf{x}_i$ actually belongs to the class k or not.

- **Quadratic loss**. This loss is used in many different tasks like document classification [Lodhi et al., 2002] or in unsupervised tasks like clustering [Jacob et al., 2009]. It is defined as:

$$\forall f \in \mathcal{Y}^{\mathcal{X}},\ \forall (\mathbf{x}_i, y_i) \in \mathcal{S},\quad C(f, \mathbf{x}_i, y_i) = (f(\mathbf{x}_i) - y_i)^2 \tag{2.5}$$

The objective to be achieved by an algoritm can be defined by its global loss. In supervised learning, we want the algorithm to make correct predictions so the objective is to minimize the loss function. We can define the *objective function* as the opposite or the inverse of the loss function. For unsupervised learning, the loss function is defined with other terms since there are no labels associated with data. It can be for example a function that measures how similar are data grouped together in clusters, or how much information is lost in dimensionality reduction transformations. The objective is however the same, to minimize the loss so the model gets better [Pearson, 1901, Jacob et al., 2009].

2.3.2 How do models learn?

In Subsection 2.3.1, we have seen that the objective of a machine learning algorithm is to make fewer and fewer incorrect predictions, which is equivalent to reduce the value of its global loss. The algorithm learns to correctly predict output labels by minimizing the global loss, *i.e.* by solving the optimization problem of reducing the global loss. Machine learning algorithms are composed of many parameters (commonly noted as Θ in the literature) which are used by the learned function f_Θ to map input data to output labels. The algorithms *learns* the mapping function f^*_Θ that best predict output labels by finding the parameters which minimize the global loss function, *i.e.* by solving:

$$f^*_\Theta = \arg\min_{\Theta} \quad \text{Global Loss } (f_\Theta) \tag{2.6}$$

To solve this optimization problem, algorithms use data (and labels in the case of supervised learning) and techniques to update the values of the parameters Θ to decrease the

global loss. Many optimization techniques exist, the most common one being the gradient descent which consists in computing the derivative of the global loss with respect to all the parameters Θ and update the value of the parameters with the value of the derivative. This technique (gradient descent) is at the core or many machine learning models like neural networks (presented in Subsection 2.4.1) or autoencoders (presented in Subsection 2.4.2) and also in common models used to learn word embeddings (presented in Chapter 3). One requirement to use the gradient descent technique is that the global loss needs to be differentiable; this is the case for the cross-entropy and the quadratic losses presented in Subsection 2.3.1, but not for the 0/1 loss.

Once the parameters Θ that solve the optimization problem have been found, the algorithm and its learned mapping function f^*_Θ can be used on new unseen data. But it is common to first evaluate the performance of the learned algorithm. Evaluation can only be done for supervised learning algorithms since it consists in comparing the predictions of the algorithm and the correct expected output labels (and there are no labels in the case of unsupervised learning). But evaluating the algorithm with the same labeled data as the one used during training is not good because it cannot evaluate if the algorithm is able to generalize what it has learned (*i.e.* if it can correctly predict labels of new data). A solution is to split data into two sets: one used for the learning phase, another one used for the evaluation phase. With this method, evaluation data is not used during training and one can evaluate what would be the performance of the algorithm on new data. Another solution is *cross-validation*, which consists in randomly splitting data into train and evaluation tests k times and consecutively train/evaluate the algorithm k times. The final evaluation performance of the algorithm is the average of its k evaluations.

When the algorithm has poor performances on evaluation data but good performances on train data, we say that it is *overfitting*: it is when the algorithm has learned by heart to correctly predict the label of train data, but it cannot generalize to unseen data. In this case, the model has not learned the underlying rules or patterns in data to solve the given task. One solution to prevent overfitting is to modify the optimization problem and add a *regularization* term on the parameters Θ. The problem to solve becomes:

$$f^*_\Theta = \arg\min_\Theta \quad \Big(\text{Global Loss}(f_\Theta) + \text{Regularization}(\Theta)\Big) \tag{2.7}$$

Regularization is a loss on the parameters Θ that penalizes the algorithm if the values of parameters are too high. Most commonly used regularizations in machine learning algorithms are ℓ_1-norm [Schmidt, 2005] and ℓ_2-norm [Cortes et al., 2012].

2.4 Machine learning models in word embeddings learning

Machine learning models car be very diverse. Besides being either trained with supervised or unsupervised learning, they can be used to solve a myriad of different tasks: information retrieval [Manning et al., 2008], machine translation [Hunsicker et al., 2012], text classification [Zhang et al., 2015], etc. Sometimes, models are specifically designed to solve a particular task. For example, in a question answering task where the chronological order of words and sentences is important, recurrent models are used to take into account additional temporal information [Yin et al., 2015]. Presenting all the existing machine learning models would be too long so in this subsection, we focus on the most common models found in the word embedding learning literature. We will see in Chapter 3 and Chapter 4 how those models are used to learn word representations and in the contributions of this thesis (Chapter 5 and Chapter 6) how we modified and improved them for our specific needs.

2.4.1 Neural network

First neural network models began to be developed during the 1950s [Rosenblatt, 1958] but they only started to gain popularity in machine learning approaches with the advances in computing resources during the 1980s [Reilly et al., 1982, Rumelhart et al., 1986]. A second rise in popularity and interest happened during the 2010s with the advances in parallel computing units, especially with GPUs [Krizhevsky et al., 2012, Parkhi et al., 2015, Silver et al., 2016]. Neural networks can be divided in several parts. The smallest unit in a neural network is a *neuron*.

Definition 2.3 (A neuron)**.** A neuron is a unit that receives different input values $\{\mathbf{x}_{[i]}\}_{i=1}^{m}$ from a vector $\mathbf{x} \in \mathbb{R}^m$ and combines them to produce an output value $\hat{y} \in \mathbb{R}$. A neuron is defined by its parameters:

- its weights: $\{\mathbf{w}_{[i]}\}_{i=1}^{m} \in \mathbb{R}$
- its bias: $b \in \mathbb{R}$
- its activation function: f

The output value $\hat{y}$ of a neuron is computed as follows:

$$\hat{y} = f(\mathbf{w} \cdot \mathbf{x} + b) = f\Big((\sum_{i=1}^{m} \mathbf{w}_{[i]} \times \mathbf{x}_{[i]}) + b\Big) \tag{2.8}$$

A visual representation of a neuron is shown in Figure 2.1.

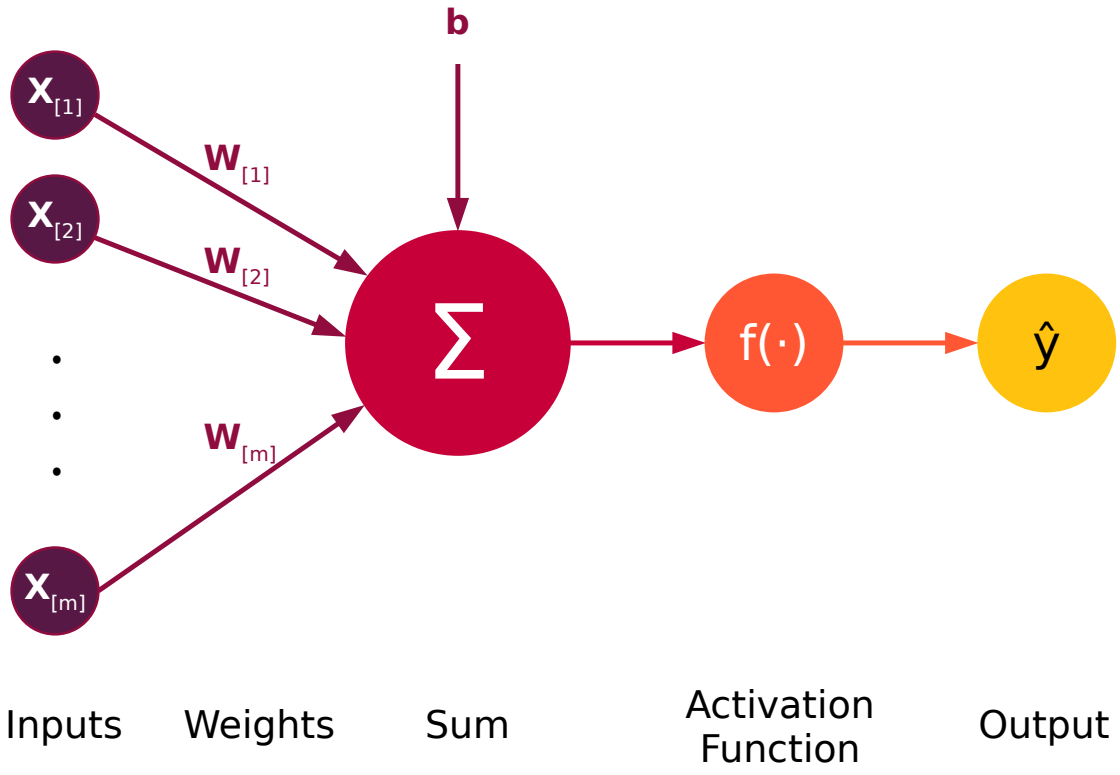


Figure 2.1: Illustration of a neuron: each input value $\mathbf{x}_{[i]}$ is mapped to its corresponding weight $\mathbf{w}_{[i]}$.

The activation function of a neuron usually depends on the task solved by the neural network model. Some functions are more suited for certain kinds of problems because they introduce non-linearity in the combination of the input values $\mathbf{x}_{[i]}$. Most commonly used activation function are plotted in Figure 2.2. They are:

- Identity function: $f(x) = x$
- Sigmoid function: $f(x) = \frac{1}{1 + e^{-x}}$
- Hyperbolic tangent function: $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- ReLU function: $f(x) = \max(0, x)$

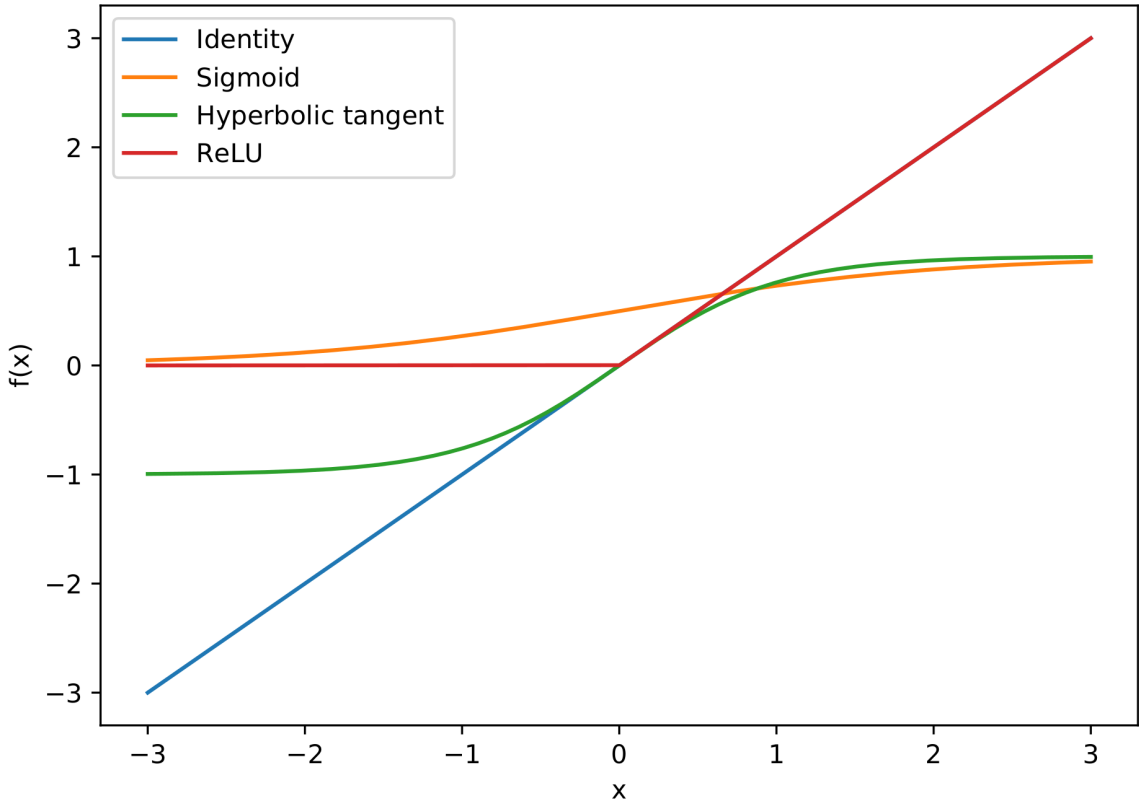


Figure 2.2: Examples of activation functions found in a neuron.

In a neural network, neurons are stacked up and linked together to form *layers*.

Definition 2.4 (A layer)**.** A layer in a neural network is a group of neurons receiving the same input values. Layers are then cascaded so that the output values of a layer are the input values of the next layer. The first layer, which receives input values from data is the *input layer*. The last layer which produces output value(s) of the network is the *output layer*. Any layers between the input and the output layer are called *hidden layers*.

In Figure 2.3, a multi-layer neural network is represented. It contains an input layer, an output layer and multiple hidden layers. Each layer has a specific number of neurons (the i-th layer has d^i neurons). The neuron $\mathbf{n}_{i,j}$ is the j-th neuron of the i-th layer.

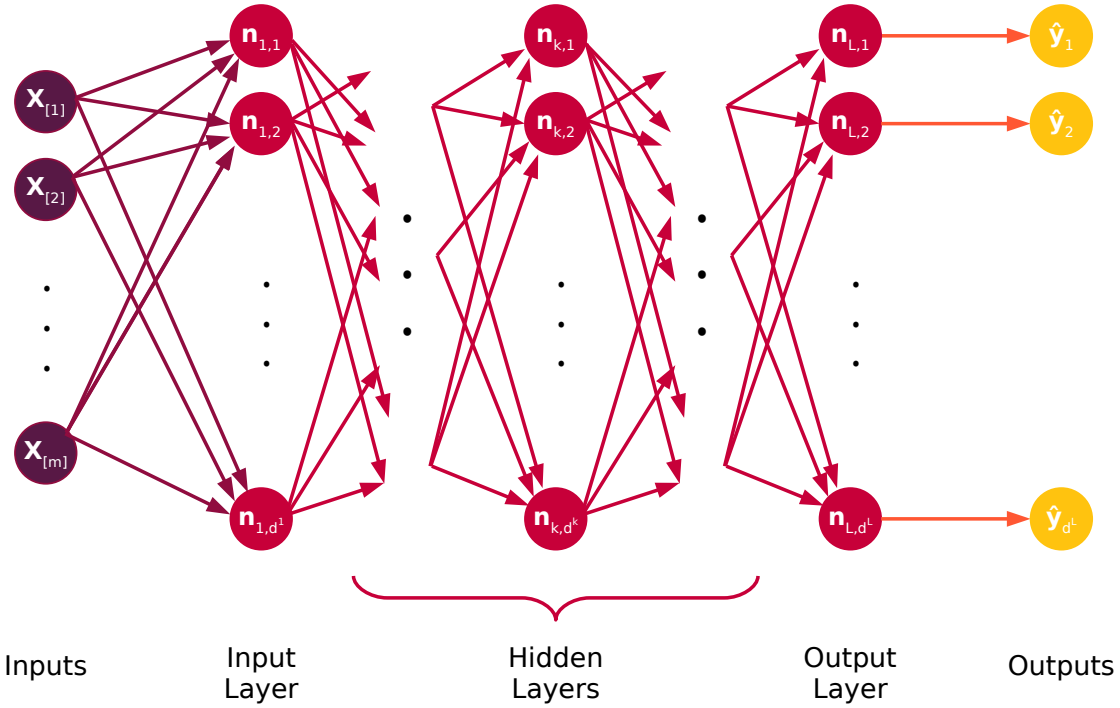


Figure 2.3: Illustration of a multi-layer neural network.

Neural networks are trained with supervised learning. They compute output values of given input labeled data and use a loss function (like the cross-entropy loss if the task is a multi-class classification problem) and the global loss on all the annotated examples to know if the predictions are correct or not. In a neural network, only the weights and biases are variable parameters (the activation function is chosen at the initialization of the neural network and is fixed during training). The optimal variable parameters Θ^* (weights and biases of neurons) that best solve the task are learned by the neural network by solving the following optimization problem:

$$\Theta^* = \arg\min_{\Theta} \quad \text{Global Loss } (\Theta) \tag{2.9}$$

The most common technique used to solve this optimization problem in neural networks is the gradient descent. Each parameter of the network has an influence on the value of the global loss because each variable parameter is used in the computation of the output value of its respective neuron and by forward propagation, used in the computation of the output values of the neural network. Each variable parameter is updated with the value of the derivative of the global loss with respect to this parameter.

Neural networks are often used in NLP [Zhang et al., 2015, Xu et al., 2015, Conneau et al., 2017]. For example, in a classification task, input vectors can be the one-hot vectors of the words of a text and output values the probabilities of the text to belong to each possible class. The neural network *learns* how to combine and weight each input word vector in order to predict the correct class. In Chapter 3, we will see how the architecture of neural networks can be modfied for the specific task of learning word embeddings.

2.4.2 Autoencoder

Autoencoders are special kinds of neural networks. Similarly, they are also composed of an input layer and an output layer, but hidden layers have various sizes. Figure 2.4 represents the architecture of an autoencoder.

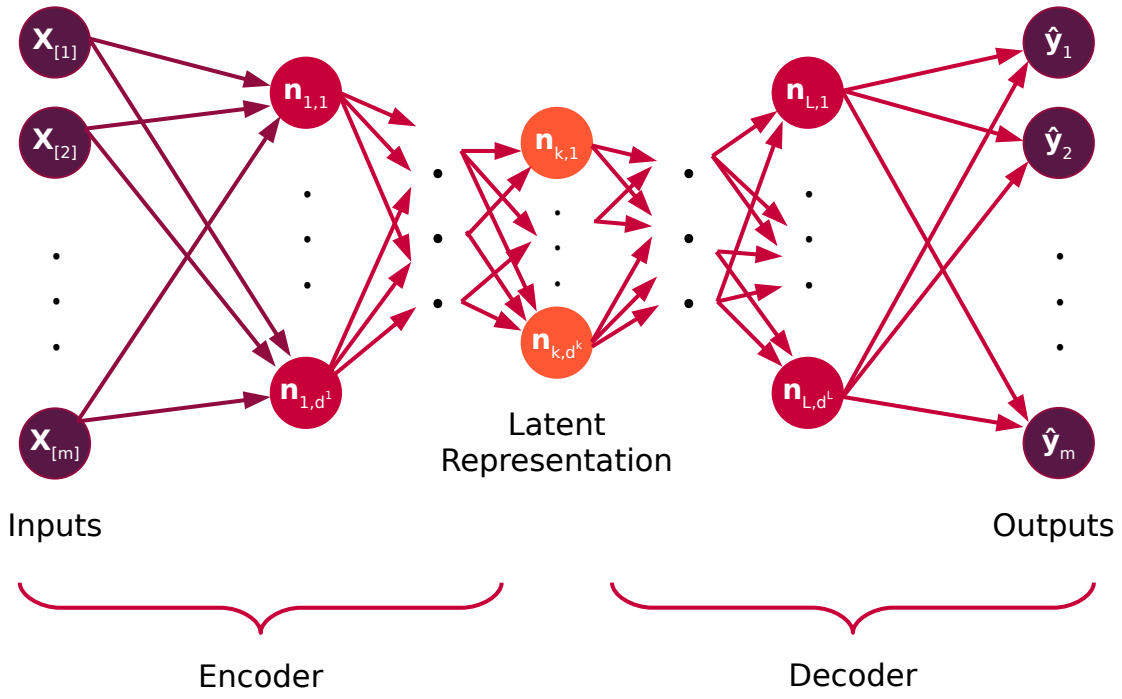


Figure 2.4: Illustration of an autoencoder. The decoder learns the latent representation which is used by the decoder to compute output values similar to input values.

First half of the layers of the autoencoder have a decreasing layer size, until the middle layer of the autoencoder. Second half of the layers have an increasing layer size, until the output layer. Output values $\hat{y}_k$ of an autoencoder are not predictions but values that aims to be as close as possible as the input vector values $\mathbf{x}_{[k]}$. Therefore, input and output layers must have identical size. Autoencoders are unsupervised models but trained like supervised models. However, they do not need any labels: they use a loss which compares output values to input vector values. A common loss used to train autoencoders is the quadratic loss, which is defined for each input vector as:

$$\text{Loss}(\mathbf{x}, \hat{\mathbf{y}}) = \|\mathbf{x} - \hat{\mathbf{y}}\|^2 = \sum_{i=1}^{m} (\mathbf{x}_{[i]} - \hat{y}_i)^2 \tag{2.10}$$

Autoencoders are generally trained with the gradient descent technique to minimize the global loss on all the examples. The value of the derivative of the global loss with respect to each parameter of the model is used to update the value of parameters.

Autoencoders are commonly used for dimensionality reduction (we will see in Chapter 4 different methods to reduce the size of word embeddings, autoencoders are one of them). They are trained to learn a smaller representation of input data, but which contains enough information to *reconstruct* the input data. Autoencoders are composed of two parts:

1. an encoder: composed of all the layers from the input layer to the middle hidden layer (the orange neurons in Figure 2.4). In the encoder part, the size of layers is only decreasing. The encoder can be seen as a function ϕ that transforms an input vector $\mathbf{x} \in \mathbb{R}^m$ into a smaller vector $\ell \in \mathbb{R}^k$ which is called the *latent representation*:

$$\phi : \mathbb{R}^m \to \mathbb{R}^k, m > k; \quad \ell = \phi(\mathbf{x}) \tag{2.11}$$

2. a decoder: composed of all the layers from the middle hidden layer (the orange neurons in Figure 2.4) until the output layer. In the decoder part, the size of layers is only increasing. The decoder can be seen as a function ψ that transforms a hidden representation $\ell \in \mathbb{R}^k$ into a vector $\hat{\mathbf{y}} \in \mathbb{R}^m$:

$$\psi : \mathbb{R}^k \to \mathbb{R}^m, k < m; \quad \hat{\mathbf{y}} = \psi(\ell) \tag{2.12}$$

The objective of an autoencoder is to compute output vectors $\hat{\mathbf{y}}$ as close as possible as input vectors $\mathbf{x}$, *i.e.* to learn ϕ and ψ that mimimize $\|\mathbf{x} - \hat{\mathbf{y}}\|$ for all the data vectors $\mathbf{x} \in \mathcal{S}$. In other terms, to solve the following optimization problem:

$$\arg\min_{\phi,\psi} \sum_{\mathbf{x}\in\mathcal{S}} \|\mathbf{x} - \psi(\phi(\mathbf{x}))\| \tag{2.13}$$

The hidden representation ℓ, called the *latent representation*, has less dimensions than the input vector $\mathbf{x}$, but the decoder part only uses this latent representation to produce $\hat{\mathbf{y}}$. Therefore, the autoencoder has to learn a latent representation which contains enough information to produce a vector $\hat{\mathbf{y}}$ close to the input vector $\mathbf{x}$. In an autoencoder, both parts work jointly: the encoder learns which values of the input vector are the most important to keep and transform into the latent representation while the decoder tells the encoder if its choices are good enough to reconstruct the input vector.

2.5 Conclusion

Machine learning models are great tools to solve problems automatically. Their main advantage is that they can find by themselves what is the information encoded in data representations required to solve the problem, without any human interactions to indicate what are the steps to follow. When they are used in the NLP domain, they can solve problems like translating documents, classifying texts into the correct category or grouping together people with similar style of writing.

In this chapter, we have described what is machine learning and what are the differences between the two main types of machine learning algorithms: supervised learning uses labeled data to learn how to map inputs to outputs while unsupervised learning learns about the distribution of information in data. We have also presented the most common models used to learn word embeddings, which are neural networks and autoencoders, and we have explained how they learn from data to solve a given problem.

However, in Chapter 1, we have explained that finding numerical representations for words is an important preliminary step when solving NLP tasks, because good representations that encode a large amount of linguistic information is useful for the downstream machine learning model: if this downstream model has access to more relevant information, it can better understand data and make better predictions. Which raises the following question: how can machine learning models be used to learn good word representations? This is the main topic of the next chapter. In Chapter 3, we present a detailed review of the existing methods to learn word embeddings and the characteristics of each method.

Part II

Learning *"Good"* Word Representations

Chapter 3

Encoding Linguistic Information into Word Embeddings

3.1 Introduction

In Chapter 1, we have explained why finding numerical representations of words, called *word embeddings*, can help downstream models that use them to solve NLP tasks. Word embeddings can also be used as a starting point for more complex models. The structure of the embedding vector space should reflect the linguistic information and properties of words so that downstream fine-tuned models can learn about the relations between words and make better predictions. Therefore, word embeddings should be "good" representations of the language in order to be helpful for the downstream models using them.

The notion of "good" word representations is vague, so let us define what good word embeddings mean. A "good" word embedding is able to encode relations and linguistic properties of words. First, semantic information about the meaning of words should be present within word embeddings. For example, it needs to encode that a keyboard is more related to a computer than to a spoon. Second, syntactic information should also be present: word embeddings should encode that verbs are different from nouns or adverbs. Third, relations between words also need to be encoded, such as the relation between a country and its capital city or the relation between masculine nouns and their femininine equivalent. Lastly, "good" word representations should give good performances when they are used into another model to solve a NLP task. We can measure how good is the encoded linguistic information in word embedding with evaluation tasks, like intrinsic or extrinsic evaluations (see Section 1.4 in Chapter 1). Intrinsic evaluations measure how much linguistic information is encoded in word embeddings while extrinsic evaluations measure how useful are the embeddings when they are used by downstream models.

Finding "good" word representations is a difficult problem because linguistic information can be of different nature (semantic, syntactic) and the vector space of word embeddings is infinite so there are no perfect representations nor a unique solution to this problem. Seminal works on word vectors [Osgood, 1964, Bierwisch, 1970] tried to find the embeddings values manually by setting vector values with common information (like values indicating if a word belongs or not to a specific semantic group). But manually setting embeddings values is a complex and expensive task because the vocabulary size can be very large and the linguistic expertise required to know which information should be incorporated into word embeddings is costly and time-consuming. NLP scientists turned to machine learning algorithms and statistical models to automatically solve this problem and learn word embeddings. This chapter describes the most common models used to learn word representations and discusses the strenghts and weaknesses of each one.

3.2 General methods to learn word embeddings

Different models have been created to learn word embeddings. They can be divided into three main categories:

1. statistical language modeling,
2. neural network approaches,
3. matrix factorization approaches.

The three categories are described in Subsection 3.2.1 for statistical language models, Subsection 3.2.2 for neural networks and Subsection 3.2.3 for matrix factorization.

3.2.1 Statistical language modeling

A statistical language model learns a probability distribution over all possible sentences of a language. It also learns the probability of a word to appear in a sentence if the previous words of the sentence are known. More formally, let S be a sentence composed of n words $S = w_1,\ w_2,\ \ldots,\ w_k,\ \ldots,\ w_n$. A statistical language model learns the conditional probability that a word w_i occurs given all the previous ones:

$$P(w_i \mid w_1,\ w_2,\ \ldots,\ w_{i-1}) \tag{3.1}$$

The model also learns the probability of each sentence S with:

$$P(S) = P(w_1,\ \ldots,\ w_n) = \prod_{i=1}^{n} P(w_i \mid w_1,\ \ldots,\ w_{i-1}) \tag{3.2}$$

These models are mostly used for speech recognition [Bahl et al., 1989] or machine translation [Brown et al., 1990] because in those tasks words are processed sequentially and are strongly related to the ones already processed. A speech recognition system cannot use the next words of the sentence to predict the current one because they have not been spoken yet. Note here that it is different from a text classification system. In a classification system, all the words of the text are simultaneously available to the model to make its predictions. While statistical language modeling are able to encode linguistic information (some words are more related to others because their probability to appear in a given context is higher), it fails to model relations between words that are in consecutive sentences and have poor generalization properties (they are highly dependent on the training data).

3.2.2 Neural network approaches

Seminal works

Statistical language modeling learns a probability distribution of sequences of words in a language. This probability distribution is learned over a large corpus of text. The model is then used to predict words of a sentence, *e.g.* for language translation or speech recognition tasks. But sentences used for testing or for inference are likely to be different from sentences used during training because, for example, people want to translate texts that have not been translated yet. Therefore, learning a probability distribution that can generalize well on unseen sentences is a difficult task as the dimension of the probability distribution grows exponentially with the number of words in the vocabulary and the length of sequences. Bengio et al. [Bengio et al., 2003] propose to solve this problem by learning a distributed representations of words. They associate a d-dimensional vector $\mathbf{v}_w \in \mathbb{R}^d$ to each word w of the vocabulary and learn simultaneously the probability

distribution of word sequences as a function of those vectors. The dimension of the vector space is fixed and is no longer dependent on the vocabulary size or the lengths of sequences. The probability distribution is learned with a neural network architecture where the inputs are the word vectors of the sequence $w_1, \ldots, w_{k-1}$ and the output is a vector whose i-th value is the probability that the word w_i is the next word in the sequence.

In 2008, Collobert and Weston [Collobert and Weston, 2008] proposed a novel architecture that jointly learns representations of words and how to use them in different tasks. The model aims to solve simultaneously part-of-speech (POS) tagging, chunking, semantic role labeling, name entity recognition and predicting if two words are related or not with a multitask learning scheme. The model they use is a single convolutional neural network that takes a sentence as input and outputs a vector of probabilities of belonging to each possible class for a given NLP task. Since neural networks cannot handle inputs of variable size, the authors rely on the *window approach* which consists in selecting and giving the network the m words around the target word to classify. Figure 3.1 represents their model architecture. The output layer is dependent on the selected task to solve but the representations of words found in the lookup tables are shared for all the tasks. Word embeddings should therefore encode many different types of linguistic information and not only those required to solve a single specific task. During training, a single example from one of the tasks is selected and the parameters of the network are updated with the gradient descent on the loss of this example. While some labels help the model to learn (*e.g.* to indicate the part-of-speech tag of words for the POS tagging task), there are no labels to help the model to know what the values of word embeddings should be. The model has to learn the values that best solve all the tasks. This type of learning is called **semi-supervised learning** (see Subsection 2.2.3 in Chapter 1).

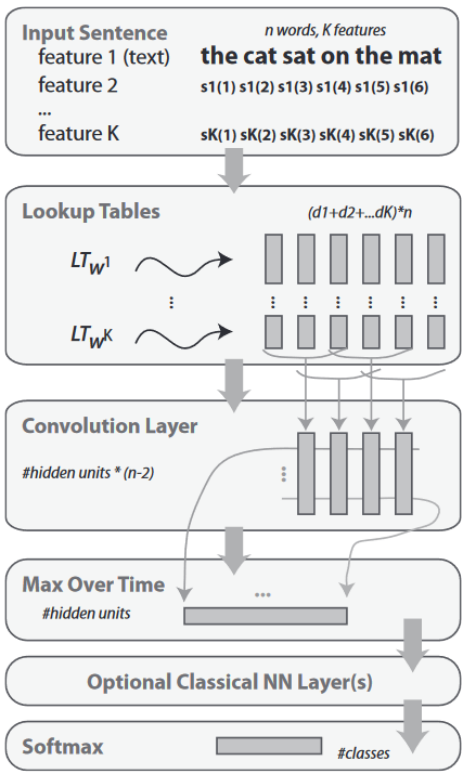
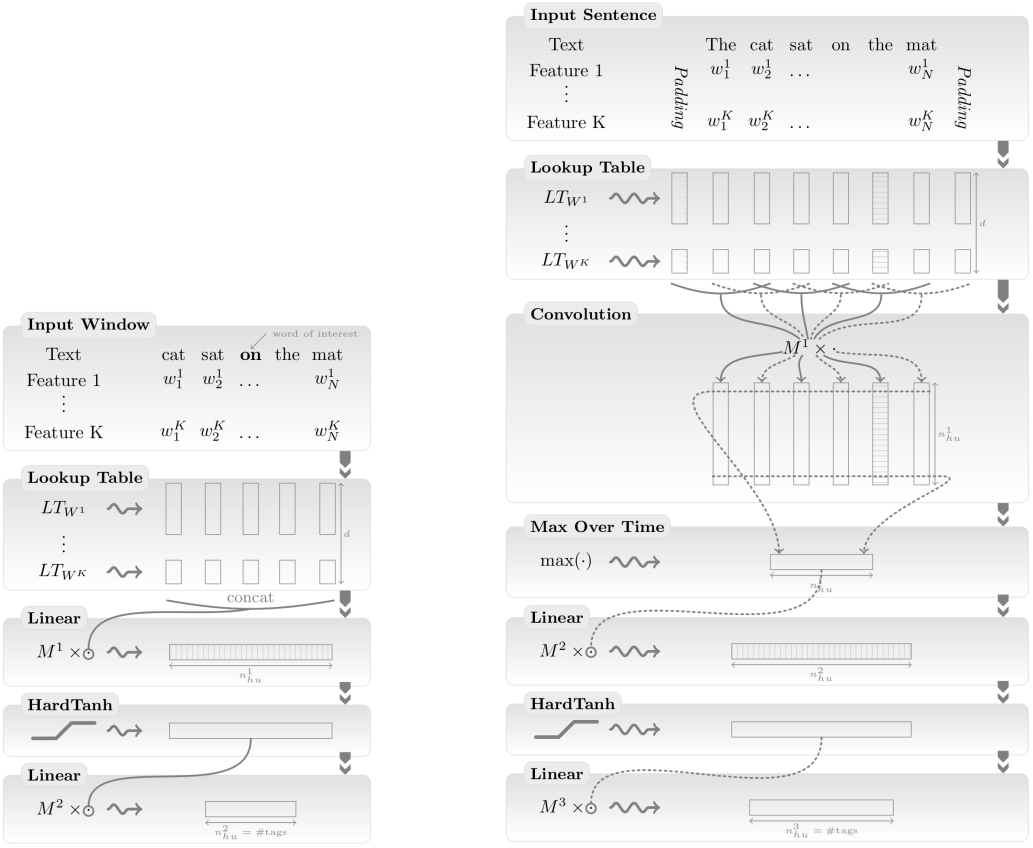


Figure 3.1: The neural network architecture used to learn word representations in [Collobert and Weston, 2008]. Word representations are fed into a convolutional layer to output a vector of probabilities of belonging to certain classes.

In 2011, Collobert et al. [Collobert et al., 2011] extended their neural network architecture to also feed the network with a complete sentence (Figure 3.2b), not only with a window of words (Figure 3.2a). The window approach was limiting the performance of the system in the semantic role labeling task because the information needed to correctly predict the answer is often outside the window. To feed the network with a sentence, Collobert et al. start by using the lookup tables to get the embeddings of each word and then use some convolutions to get a vector of fixed size. This vector is then fowarded to hidden layers of the network to produce an output vector which is, like in the window approach method, composed of the probabilities to belong to each possible class for a given task.



(a) Neural network with a window approach.

(b) Neural network with a sentence approach.

Figure 3.2: Neural network architectures used by [Collobert et al., 2011]. The lookup tables contain the word embeddings, which are then used to compute an output vector representing the predictions of the network for a given NLP task.

Word2vec

The architectures proposed by Collobert et al. [Collobert and Weston, 2008, Collobert et al., 2011] allow one to learn word embeddings but they still need some labeled data for training. Labels are not used to help the model to know what the values of the embeddings should be but to help the model to learn representations which contain information that is useful to predict the correct output and solve a given NLP task. Therefore, this semi-supervised learning still requires some human labor to annotate examples and get labels. Mikolov et al. [Mikolov et al., 2013a] proposed the novel `word2vec` model to overcome this problem. Instead of using NLP tasks as a proxy to know which information should be encoded into word embeddings, Mikolov et al. learn the distribution of embeddings

across the vectorial space $\mathbb{R}^d$ with the hypothesis of John Rupert Firth [Firth, 1957]: *"You should know a word by the company it keeps."*. Their model considers that the vectors of the words in a window should be enough to predict what are the other words in the window. Let $(w_{t-n},\ \ldots,\ w_{t-1},\ w_t,\ w_{t+1},\ \ldots,\ w_{t+n})$ be a sequence of words composed of a central word w_t and the $2n$ words around it. This sequence of words is called *a window*. The authors actually propose two different architectures of their model:

- Continous Bag-of-Words (CBOW): in this architecture, the goal is to predict the central word w_t given all the other words of the window. This is represented in the left part of Figure 3.3.
- Skip-gram: in this architecture, the goal is to predict all the words of the sequence except the central word w_t, which is used as the input. This is represented in the right part of Figure 3.3.

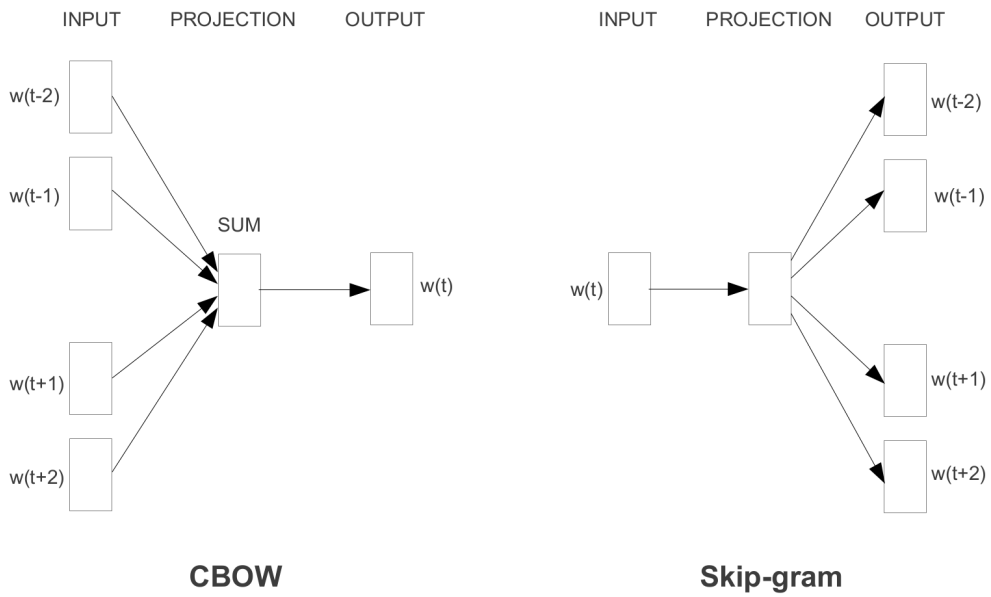


Figure 3.3: Continuous Bag-of-Words (CBOW) and Skip-gram models, respectively on left and right part of the image, used in [Mikolov et al., 2013a]

Both architectures are trained to maximize the probability of predicting the correct words (in fact, they maximize the *log* probability because it simplifies the computations of the derivatives for a gradient descent training). Their objective is therefore slightly different:

- in CBOW, the objective is to maximize the probability of predicting the central word w_t given the context sequence, *i.e.* to maximize:

$$\log\big(P(w_t \mid w_{t-n},\ \ldots,\ w_{t-1},\ w_{t+1},\ \ldots,\ w_{t+n})\big) = \log\big(P(w_t \mid \theta)\big) \tag{3.3}$$

  where θ is the average vector of the vectors of words in $(w_{t-n},\ \ldots,\ w_{t-1},\ w_{t+1},\ \ldots,\ w_{t+n})$. With the embedding vector of the word w_j noted as $\mathbf{v}_j$, it is defined as:

$$\theta = \frac{1}{2n} \sum_{\substack{-n \leq j \leq n \\ j \neq 0}} \mathbf{v}_{t+j} \tag{3.4}$$

- in Skip-gram, the objective is to maximize the probability of predicting each one of the different words of the context given the central word w_t, *i.e.* to maximize:

$$\sum_{\substack{-n \leq j \leq n \\ j \neq 0}} \log\big(P(w_{t+j} \mid w_t)\big) \tag{3.5}$$

In both models, the probability of predicting a word is computed with the embeddings of the words involved. In [Mikolov et al., 2013a], there are two embedding matrices (**WI** and **WO**, respectively for *Input* and *Output*) which both contain the embedding of each word of a vocabulary so all words are actually represented with two different vectors (one from **WI** and another one from **WO**). When a word is used as the input of the neural network (in CBOW, input words are the context words; in Skip-gram it is the central word), the corresponding word embedding is selected from the **WI** matrix. When a word is used as the output of the neural network (in CBOW, the output word is the central word; in Skip-gram they are the context words), word embeddings from the **WO** matrix are selected. The probability of predicting a word given another word is expressed with the softmax function and is defined as:

$$P(w_i \mid w_j) = \frac{\exp({\mathbf{v'}_i}^\top \mathbf{v}_j)}{\sum_{w \in \mathcal{V}} \exp({\mathbf{v'}_w}^\top \mathbf{v}_j)} \tag{3.6}$$

where $\mathbf{v}$ (resp. $\mathbf{v'}$) refers to embeddings selected in the **WI** (resp. **WO**) matrix, and $\mathcal{V}$ is the vocabulary containing all words. Computing the softmax function is computationally expensive because it needs to iterate over all the words in the vocabulary $\mathcal{V}$, which can contain millions of words. Mikolov et al. [Mikolov et al., 2013a] introduced the negative sampling to simplify this expression, which is based on the noise-contrastive estimation [Gutmann and Hyvärinen, 2012, Mnih and Teh, 2012]. The idea is to randomly select only a fraction of words from the vocabulary and constrain the model to predict the expected correct word with a higher probability than for those randomly selected words. The probability of predicting a word becomes:

$$\log\big(P(w_i \mid w_j)\big) = \log\big(\sigma({\mathbf{v'}_i}^\top \mathbf{v}_j)\big) + \sum_{l=1}^{k} \mathbb{E}_{w_l \sim P_k(w_j)} \log\big(\sigma(-{\mathbf{v'}_l}^\top \mathbf{v}_j)\big) \tag{3.7}$$

where σ is the sigmoid function defined as $\sigma(x) = \frac{1}{1+e^{-x}}$ and $P_k(w_j)$ is a random noisy distribution created by selecting k random words different from w_j from the vocabulary $\mathcal{V}$. The final objective function of CBOW and Skip-gram models is obtained by maximizing the log probability over all the possible context/central word pairs of windows of size n across a corpus of size T. For Skip-gram, the objective of the model is to maximize:

$$\begin{aligned} &\frac{1}{T}\sum_{t=1}^{T} \sum_{\substack{-n \le j \le n \\ j \neq 0}} \log\big(P(w_{t+j}|w_t)\big) \\ =&\frac{1}{T}\sum_{t=1}^{T} \sum_{\substack{-n \le j \le n \\ j \neq 0}} \Big(\log\big(\sigma({\mathbf{v'}_{t+j}}^\top \mathbf{v}_t)\big) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_k(w_t)} \log\big(\sigma(-{\mathbf{v'}_i}^\top \mathbf{v}_t)\big)\Big) \end{aligned} \tag{3.8}$$

Models are trained by iterating through the corpus several times (each full iteration is called an *epoch*). After a certain number of epoch are done, final word embedding for each word of the vocabulary is obtained by averaging its two embeddings from **WI** and **WO**.

CBOW and Skip-gram models are unsupervised machine learning models. They do not use any labels during training but only consider cooccurrences of words inside windows to learn the values of word embeddings. However, they are able to encode linguistic information. Mikolov et al. [2013a] show that embeddings learned by their `word2vec` model perform well in the word analogy task (0.53 for Skip-gram vs. 0.11 for the neural network model of [Collobert and Weston, 2008]) as well as in a sentence completion challenge (0.64 for CBOW vs. 0.51 for the model of [Bengio et al., 2003]). Arora et al. [Arora

et al., 2015] have proposed to explain the reasons of the existence of linear relations in word embeddings. Starting with a generative model, they perform a random walk on word vectors instead of sequentially looking at the word windows of a large corpus. They compute the probability that two words appear in the same window with the random walk and demonstrate theoretically the emergence of relationships between words and why one can find linear relationships in word embeddings learned with the `word2vec` model.

Fasttext

While word embeddings learned by the CBOW or the Skip-gram objective functions are able to capture semantic properties of words like the feminine attribute of the word "queen" compared to the word "king", they are lacking some syntactic properties. Indeed, in those models, words are considered as global independent entities but in languages like English, words that share common letters or subgroups of letters are often related. For example, in English, adverbs typically end with the suffix *-ly*. Similarly, prefixes can also indicate that words are related like for the words "possible" and "impossible", and common etymological roots also tell information about similar words like for the words "telephone" and "microphone". This observation is at the origin of the idea developed by Bojanowski et al. [Bojanowski et al., 2017]. Instead of considering a word as a single entity, they consider it as a set of *n-grams*, *i.e.* groups of subwords with length between 2 and 6 letters. For example, the word "badly" is decomposed as: {"ba", "ad", "dl", "ly", "bad", "adl", "dly", "badl", "adly", "badly"}. This information is plugged into a `word2vec` Skip-gram model to produce a novel model called `fasttext` where words are replaced by combination of n-grams. Each n-gram is associated to its own vectors in the embedding matrices **WI** and **WO**, so the model learns simultaneously embeddings of words and embeddings of n-grams. After training, only word embeddings are kept for future use but embeddings of n-grams help the model to understand the morphological similarities between words.

When the `fasttext` model is evaluated on the word analogy task, results are greatly improved especially on the syntactic evaluation dataset because embeddings are able to capture linguistic information to learn that "great" and "greatly" have the same relation as "bad" and "badly". Moreover, the `fasttext` model presents other great advantages. First, in CBOW and Skip-gram models, words that are not in the training corpus are not seen and therefore do not have a learned word embedding associated to them. With `fasttext`, it is possible to construct a word embedding for such words (called *out-of-vocabulary* words) because the vector of a word can be seen as the sum of the vectors of its n-grams. The n-grams solution proposed by Bojanowski et al. [Bojanowski et al., 2017] allows one to create embeddings for new unseen words without having to re-train the model on a new corpus containing those new words. Second, words occuring rarely in a text are less frequently seen in approaches which iterate through the corpus with a sliding window of words like in `word2vec` or in `fasttext`. Therefore, representations of rare words is often not as good as the representations of frequent words. However, `fasttext` also learns a representation for n-grams. A rare word like "interconnectedness" which shares common n-grams with the words "connect" or "connection" can still capture semantic information in its embedding because the words "connect" or "connection" are more frequently seen in different contexts and the representations of their n-grams are more frequently updated to encode this information. `fasttext` is therefore better than `word2vec` to learn the representations of rare words because of the use of n-grams in the model (on a word semantic similarity task, word embeddings learned by `fasttext` obtain a score of 0.47 on the Rare-Words dataset [Luong et al., 2013] while vectors learned by the `word2vec` Skip-gram and CBOW models obtain 0.43).

3.2.3 Matrix factorization

The methods presented in the previous subsection rely on a neural network architecture and a sliding window approach to learn word embeddings. These methods use the cooccurrence information between words to train the vectors of words such that words cooccurring frequently together in context windows have similar word vectors. However, the information they use to learn word embeddings is only local information (context windows). Matrix factorization methods use different statistical information of the training corpus by considering the global cooccurrence count of words instead of their local ones as in context windows. Since the vocabulary of the corpus can be in the order of millions, using the global word cooccurrences matrix as the embedding matrix is not feasible, so these approaches factorize the global cooccurrence information matrix with singular value decomposition (SVD) and truncate it to keep only the first k dimensions in order to obtain a smaller matrix with approximated properties, which is used as the embedding matrix.

One of the first work in global matrix factorization is *Latent Semantic Analysis* (LSA) [Deerwester et al., 1990] which uses a matrix containing the number of occurrences of words in several documents. Another similar method [Lund and Burgess, 1996] factorizes a matrix containing the information about the cooccurrence of words in context windows. A formalization of a more general matrix factorization approach has been described in [Turney, 2012, Levy and Goldberg, 2014]. Let $\widetilde{\mathbf{M}}$ be a matrix containing cooccurrence information of words obtained from a text corpus and let $\widetilde{\mathbf{M}} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$ be its singular value decomposition. A word embedding matrix $\mathbf{M}$ containing k-dimensional word vectors is obtained with:

$$\mathbf{M} = \mathbf{U}_{1:k}\mathbf{D}^{\alpha}_{1:k,1:k} \tag{3.9}$$

where $\mathbf{U}_{1:k}$ is the matrix $\mathbf{U}$ truncated to keep only the first k columns, $\mathbf{D}_{1:k,1:k}$ is the matrix $\mathbf{D}$ truncated to keep only the first k rows and k columns and $\alpha \in [0, 1]$.

Another common method used to learn word embeddings is `GloVe` [Pennington et al., 2014]. While `GloVe` is not based on SVD like the other methods, it has been shown that the objective function of `GloVe` is an implicit factorization of the logarithm of the global word cooccurrence count matrix [Levy et al., 2015]. For each possible pair of words (w_i, w_j) with $w_i, w_j \in \mathcal{V}$ (where $\mathcal{V}$ is the vocabulary of the text corpus), the `GloVe` model aims to learn word vectors such that:

$$\mathbf{v}_i^\top \ \mathbf{v}_j + b_i + b_j = \log(1 + \widetilde{\mathbf{M}_{ij}}) \tag{3.10}$$

where $\mathbf{v}_i$ (resp. $\mathbf{v}_j$) is the word vector of the word w_i (resp. w_j), b_i and b_j are some bias related to the words w_i and w_j and $\widetilde{\mathbf{M}_{ij}}$ is the number of times the word w_i occurs in the context of the word w_j. The vector of each word of the vocabulary $\mathcal{V}$ is obtained by training the model to minimize the global loss function J, defined as:

$$J = \sum_{i,j=1}^{|\mathcal{V}|} \lambda_{ij}\big(\mathbf{v}_i^\top \ \mathbf{v}_j + b_i + b_j - \log(1 + \widetilde{\mathbf{M}_{ij}})\big)^2 \tag{3.11}$$

where λ_{ij} is a weight depending on the value of $\widetilde{\mathbf{M}_{ij}}$ so that pairs of words with a zero or a small cooccurrence count do not have a lot of importance in the loss function.

3.3 Improving word embeddings with external resources

General methods to learn word embeddings, whether by using a neural network approach (presented in Subsection 3.2.2) or a matrix factorization approach (presented in Subsection 3.2.3) mainly use training text corpora like Wikipedia articles or crawled content from the Web. While those corpora contain a lot of information, their content is rather generic, in a sense that specific information about words is not present. Semantic relations like synonymy, antonymy, hypernymy, etc. are not explicitly written in those corpora and it is unlikely that two words linked with such relations cooccur within the same window or at least not often enough together for the model to encode this kind of relation between words into their embeddings. To overcome this problem, several methods have been proposed to improve the linguistic information contained in word embeddings with other additional sources of knowledge. Approaches to improve word embeddings with other sources of knowledge can be separated into two main categories: either by using additional knowledge during training or either by using it after training.

The first category uses additional knowledge **during the training** of word embeddings. Yu and Dredze [Yu and Dredze, 2014] extract synonym information from WordNet [Miller, 1995] and a paraphrase database (PPDB) [Ganitkevitch et al., 2013]. This new information is used in a `word2vec` CBOW model to add new constraints between some related words. Embeddings learned with this method have better results on a word semantic similarity task, demonstrating that using additional information during training can improve word embeddings. Kiela et al. [Kiela et al., 2015] improve both the similarity and the relatedness information encoded into word embeddings by using respectively synoyms and word associations into a `word2vec` Skip-gram model. Embeddings are then evaluated on a word semantic similarity and a synonym detection tasks where the results are improved in both tasks compared to a regular Skip-gram model.

The second category uses additional knowledge **after the word embeddings have been learned**, as a post-training step. Faruqui et al. [Faruqui et al., 2015a] propose to extract pairs of related words from semantic lexicons resources like WordNet [Miller, 1995], Framenet [Baker et al., 1998] or PPDB [Ganitkevitch et al., 2013]. Pairs are used to update the representations and move closer the embeddings of words in those pairs, in a method called `retrofitting`. Like the other methods that use additional knowledge during training, embeddings learned with `retrofitting` give better performances in tasks like word semantic similarity, synonym detection or sentiment analysis. The `retrofitting` method was extended by Jo and Choi [2018] in a novel method called `extrofitting`. In addition to updating word embeddings values with extracted pairs, the authors change the number of dimensions in vectors before doing any modifications and they reorganize the vectors based on the additional knowledge so both word embeddings and the vector space benefit from the additional information found in semantic lexicons. Final `extrofitting` word embeddings perform better than the Faruqui et al. retrofitted vectors on a word semantic similarity task (+3.46% on average on the MEN dataset [Bruni et al., 2014] when improving `GloVe` embeddings of various sizes) which were already better than original vectors learned by the `GloVe` model.

3.4 Contextual word representations

In all the methods presented so far for learning word embeddings, whether by using directly large text corpora like in `word2vec`, `fasttext` or `GloVe` or by using additional external information like in `retrofitting`, each word of the vocabulary is associated to a single word vector. However, some words in a language are *polysemous*: their meaning can be

different depending on the context in which they are used. In the two following sentences, the word "mouse" has different meanings:

1. The *mouse* is chased by the cat.

2. I am using the *mouse* to move the cursor on my screen.

In the first sentence, the word "mouse" refers to the small animal. In the second sentence, it refers to the peripheral device plugged into a computer to perform some actions. If a single unique vector is associated to the word "mouse", should this vector be close to the vectors of other animals or close to the vectors of the words "screen" and "keyboard"? There are no optimal solutions in this case but rather a clear evidence of the shortcoming of methods that associate the same vector to all the occurrences of a polysemous word: they cannot properly encode that words can have multiple meanings and thus different vector representations.

3.4.1 Sense embeddings

Neelakantan et al. [Neelakantan et al., 2014] propose to solve this shortcoming by learning a different vector for each sense of a word. Each word w of a vocabulary $\mathcal{V}$ is assigned to:

- a global vector $\mathbf{v_g}(w)$;
- a sense vector $\mathbf{v_{s,k}}(w)$ for each one of its senses, where $k \in [1, \ \ldots, \ K_w]$ and K_w is the number of senses of the word w.

They use the `word2vec` Skip-gram model of Mikolov et al. [2013b] to learn sense vectors and global vectors. For each central word w_t of a sequence of words and its associated context window $c_t = (w_{t-n}, \ \ldots, \ w_{t-1}, \ w_{t+1}, \ \ldots, \ w_{t+n})$, they compute the context vector of this window by averaging the global vectors of each word c in the context c_t:

$$\mathbf{v_{context}}(c_t) = \frac{1}{2n} \sum_{c \in c_t} \mathbf{v_g}(c) \tag{3.12}$$

They determine which sense s_t of w_t is used in this context c_t by finding the closest sense vector $\mathbf{v_{s,k}}(w_t)$ to the context vector $\mathbf{v_{context}}(c_t)$:

$$s_t = \underset{k \in [1, \ ..., \ K_{w_t}]}{\arg\max} \ sim(\mathbf{v_{s,k}}(w_t), \mathbf{v_{context}}(c_t)) \tag{3.13}$$

where $sim()$ is a similarity function between two vectors like the cosine similarity function. After finding the sense s_t of w_t closest to the context c_t, they use the Skip-gram objective function and the gradient descent technique to update the values of the sense vector $\mathbf{v_{s,s_t}}$ of the word w_t and the values of the context vector $\mathbf{v_{context}}(c_t)$ by maximizing:

$$\log \left(P(\mathbf{v_{s,s_t}}(w_t) \mid \mathbf{v_{context}}(c_t)) \right) \tag{3.14}$$

This approach has the advantage of not needing human labeling on data, because the sense s_t of the word w_t used in the context c_t is found on its own by the method. Other approaches like [Iacobacci et al., 2015] use a word sense disambiguation tool to automatically add a suffix to words with different senses. With the small previous example above, it would annotate the word "mouse" like "$mouse_1$" and "$mouse_2$". The `word2vec` CBOW model [Mikolov et al., 2013a] is then applied on the disambiguated corpus to learn one vector per sense. This approach has the inconvenient of needing a tool to disambiguate the corpus which is often based on a manually generated external source of knowledge.

3.4.2 Contextualized word embeddings

As said in the previous subsection, words can have different meanings depending on the context in which they are used. While some approaches learn to associate a vector to each sense of each word [Neelakantan et al., 2014, Iacobacci et al., 2015], they neglect the information one can found inside words, like in `fasttext` [Bojanowski et al., 2017]which uses subword information to learn morphological properties of words. Moreover, knowing which sense of a word is used in a context can require an external disambiguation tool. Another recent way to tackle the issue of learning representations for polysemous words is to shift from word representations to occurrence representations. That is, instead of associating one embedding per word or per word sense (learned based on all the contexts in which they occur in a corpus), it could be interesting to have one context-specific embedding for each occurrence of a word in a corpus. This would relax the constraint of optimizing the representations for all the occurrences of a word (or its senses) while providing a very fine-tuned representation of a word in a given context. Those types of approaches are called *contextualized word embeddings* [Peters et al., 2018, Devlin et al., 2019, Liu et al., 2019] and have been proven to be very popular within the last couple of years. In what follows, we present the main approaches that fall into this category.

Peters et al. [Peters et al., 2018] propose a novel architecture to learn a *contextual representation* of words without requiring any labeled data or external tools for disambiguation. Their architecture is based on bidirectional LSTM [Huang et al., 2015] and is represented in Figure 3.4. Their model is called `ELMo` for **E**mbeddings from **L**anguage **Mo**dels. It relies on the objective function of a language model. As a reminder, in a language model the objective is to be able to successively predict words in a sentence given all the previous words of this sentence. For example, with the sentence "The mouse is chased by the cat.", the model aims to predict the word "chased" given the words "The", "mouse" and "is". More formally, it is equivalent to maximize:

$$P(t_1,\ t_2,\ \ldots,\ t_n) = \prod_{k=1}^{n} P(t_k \mid t_1,\ \ldots,\ t_{k-1}) \tag{3.15}$$

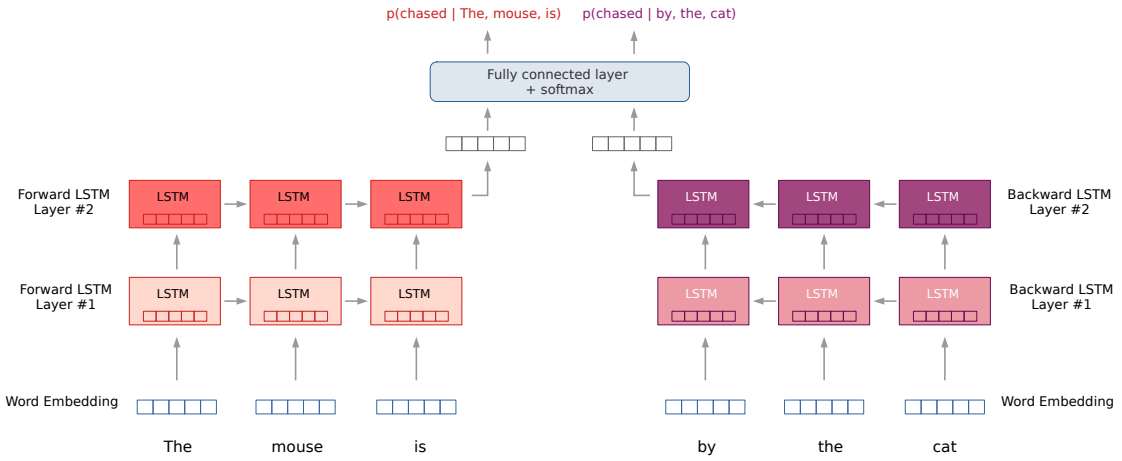


Figure 3.4: Architecture of the `ELMo` model from [Peters et al., 2018]. It is composed of 2 bidirectional LSTM layers (bi-LSTM). Each word is associated to its own word embedding which is used as the input in the first bi-LSTM layer. The outputs of the first bi-LSTM layer are then passed to the second bi-LSTM layer. The final representation of a word in a context is a combination of its own word embedding and the outputs of both bi-LSTM.

ELMo also models the language in the backward direction, *i.e.* to predict a word given all the words that come after it. This is equivalent to maximize:

$$P(t_1,\ t_2,\ \ldots,\ t_n) = \prod_{k=1}^{n} P(t_k \mid t_{k+1},\ \ldots,\ t_n) \tag{3.16}$$

The objective of the ELMo model is to maximize the log likelihood of predicting words in both directions, *i.e.* to maximize:

$$\begin{aligned} \log P(t_1,\ t_2,\ \ldots,\ t_n) = \sum_{k=1}^{n} \Big(& \log \left(P(t_k \mid t_1,\ \ldots,\ t_{k-1}; \overrightarrow{\Theta}_{LSTM}, \Theta) \right) + \\ & \log \left(P(t_k \mid t_{k+1},\ \ldots,\ t_n; \overleftarrow{\Theta}_{LSTM}, \Theta) \right) \Big) \end{aligned} \tag{3.17}$$

where $\overrightarrow{\Theta}_{LSTM}$ (resp. $\overleftarrow{\Theta}_{LSTM}$) are the parameters of the forward (resp. backward) bi-LSTM layers and Θ are the shared parameters between the bi-LSTM (the embeddings of words and the weights of the fully connected and the softmax layers).

Bidirectional LSTM (bi-LSTM) compute an internal representation for each word position k in each layer j. For forward LSTM (red boxes in Figure 3.4), each internal representation is noted as $\overrightarrow{\mathbf{h}}^{LM}_{k,j}$. For backward LSTM (purple boxes in Figure 3.4), each internal representation is noted as $\overleftarrow{\mathbf{h}}^{LM}_{k,j}$. As an example, $\overrightarrow{\mathbf{h}}^{LM}_{3,2}$ is the internal representation of the right dark red box in Figure 3.4, while $\overleftarrow{\mathbf{h}}^{LM}_{2,1}$ is the internal representation of the middle light purple box in Figure 3.4. After the objective function of ELMo has been trained to be maximized over a large text corpus, ELMo is able to compute a unique representation for each word based on their context. To compute the contextual representation of the word w_t from the context $c_t = (w_1,\ \ldots,\ w_{t-1},\ w_{t+1},\ \ldots,\ w_{t+n})$, ELMo computes the internal representations of bi-LSTM layers with the embedding of each word in the context c_t and then sums the internal representations, that is:

$$\texttt{ELMo}(w_t) = \sum_{j=0}^{N} \mathbf{h}^{LM}_{t,j} \tag{3.18}$$

where $\mathbf{h}^{LM}_{t,j}$ is the concatenation of the forward and backward representations of the word w_t in the j-th bi-LSTM layer (*i.e.* $\mathbf{h}^{LM}_{t,j} = [\overrightarrow{\mathbf{h}}^{LM}_{t,j},\ \overleftarrow{\mathbf{h}}^{LM}_{t,j}]$). For $j = 0$, the word embedding of the word w_t is concatenated with itself (*i.e.* $\mathbf{h}^{LM}_{t,0} = [\mathbf{x}^{LM}_t, \mathbf{x}^{LM}_t]$). In the ELMo model, there are only 2 bi-LSTM layers (so $N = 2$ in Equation 3.18).

The approach of ELMo to learn contextualized word representations has led to many other methods which have been proven to bring large increase of performances in downstream NLP tasks [Devlin et al., 2019, Liu et al., 2019]. In 2019, Devlin et al. [Devlin et al., 2019] proposed an architecture deeper than ELMo to model context information into word representations. Their model is called BERT (for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers) and is composed of 12 layers (24 in the larger version of BERT, only 2 in the ELMo model). Each layer is a bidirectional transformer [Vaswani et al., 2017] which uses context information from before and after a token to compute its word representation. During training, BERT randomly masks 15% of the words in sentences (by replacing them with a [MASK] token or with a random word) and tries to predict the original non-masked words given their context instead of predicting the next word of a sequence given all the previous words of this sequence like in a classic language model. BERT has recently been integrated into the Google search engine and has been considered by Jeff Dean as one of the biggest advances in the development of this search engine.

Given the large increase of performances in multiple downstream tasks with the representations learned by `BERT`, many works have extended this architecture (and also the name of the model): to optimize the training time (`ALBERT` [Lan et al., 2019]), to train on larger corpora with more layers (`RoBERTa` [Liu et al., 2019]), to specialize on another language like French (`FlauBERT` [Le et al., 2019], `CamemBERT` [Martin et al., 2019]), etc.

3.5 Overview and recap

In the previous subsections, we have described common methods used to learn word representations. Presented methods can be very diverse: using only cooccurrences of words inside a sliding window, using additional external knowledge or using context information of words. In this subsection, we are going to present an overview of all the methods presented so far and how they compare to each other with their strenghts and weaknesses.

3.5.1 Visualization of methods to learn word embeddings

A simple (and subjective) way to visually classify the different methods to learn word embeddings is to compare them on two criteria:

1. Does the method use a large architecture to learn word embeddings?
2. How much linguistic information is used during the learning of word embeddings?

Figure 3.5 represents the methods described in this chapter along two axes:

- The horizontal axis represents the relative size of the architecture used in different methods. Methods that use small architectures (like a single hidden layer neural network in [Collobert and Weston, 2008]) are placed on the left of this axis. Models with larger architectures like `ELMo` [Peters et al., 2018] or `BERT` [Devlin et al., 2019] are placed on the right of this axis.
- The vertical axis indicates (on a relative scale, not an absolute one) how much linguistic information is used during training to learn word embeddings. The more information a model uses (like external knowledge or context information in addition to solely using a text corpus), the higher it is placed on this axis.

A clear trend can be observed in Figure 3.5: the more linguistic information a model uses, the larger it is. Collobert et al. [Collobert et al., 2011] extend the model of Collobert and Weston [Collobert and Weston, 2008] by using sentences as inputs in their single hidden layer neural network instead of using only context windows. This additional linguistic information used during training (because longer inputs implies that longer dependencies between words can be extracted by the model) slightly increases the size of the model architecture. `fasttext` uses subword characteristics of words during training. This additional linguistic information, compared to models like `word2vec` or `GloVe` which do not use it, also has the consequence of increasing the size of the model because it needs to store and learn embeddings for subwords in addition to the embeddings of words.

`retrofitting` is a special type of learning method. It is used after word vectors have been learned and it incorporates additional information from external sources of knowledge. This is why in Figure 3.5 it is represented as an arrow (not as a point because it is not a learning method on its own) and only increases the quantity of used linguistic information. The size of the model stays the same because the embeddings keep the same number of dimensions. Yu and Dredze [Yu and Dredze, 2014] and Iacobacci et al. [Iacobacci

et al., 2015] both extend the `word2vec` model with a modified objective function which uses additional external knowledge, but Iacobacci et al. use an external tool to disambiguate the training corpus and they learn one vector per sense, hence having a larger model architecture compared to Yu and Dredze [2014].

Lastly, contextual word representations models like `ELMo` and `BERT` use no additional external sources of information but they use longer contexts and longer dependencies between words. While there is no objective measure of the amount of linguistic information brought by external knowledge or by longer context dependencies, I have chosen to place `ELMo` and `BERT` higher than methods which use external sources of information on the vertical axis in Figure 3.5 because embeddings learned by `ELMo` or `BERT` give higher results when they are used in downstream tasks, hence more linguistic information have been encoded into the representations compared to models that use external sources of knowledge. Concerning their model architecture, it is either based on LSTM or on transformers, which are larger and more computationally expensive than a single hidden layer neural network or than a `word2vec` model, hence they are placed on the right part in Figure 3.5.

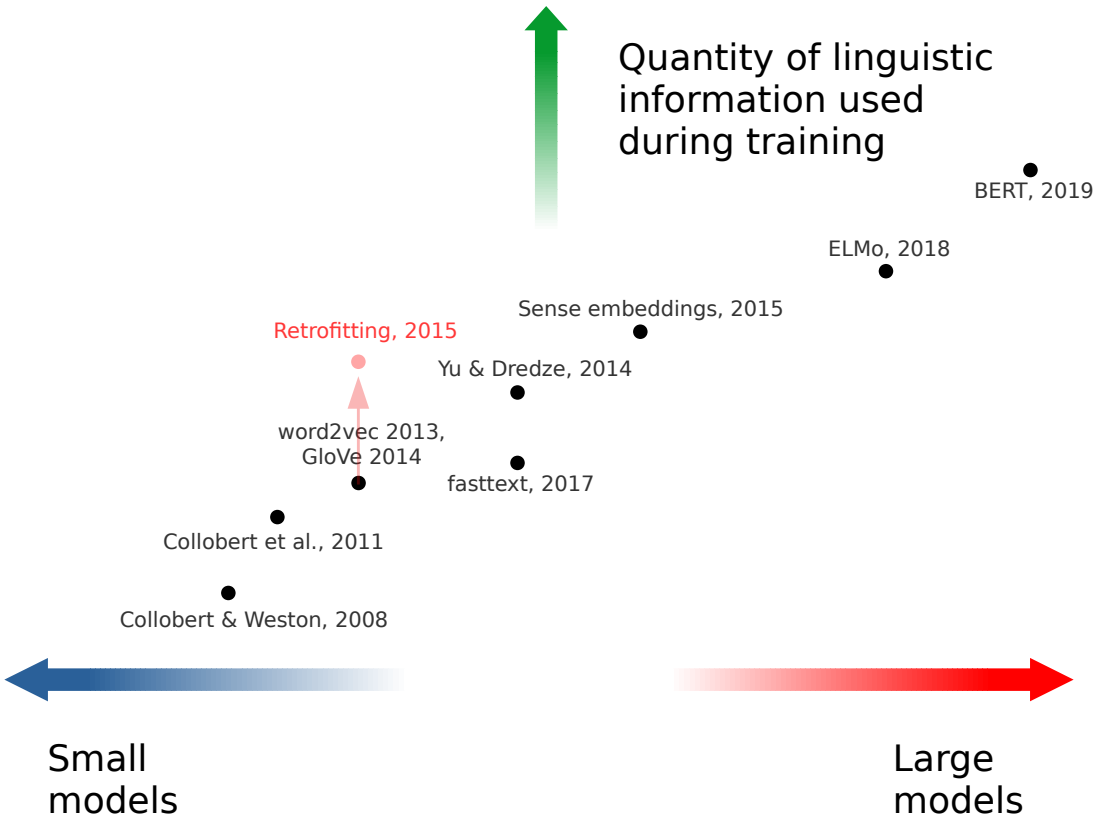


Figure 3.5: Visual representations of methods to learn word embeddings. Methods on the left of the image generally use small architectures while the architecture is larger for methods placed on the right of the image. The vertical axis indicates on a relative scale how much information is used to learn word embeddings in each method.

3.5.2 Strengths and weaknesses of methods to learn word embeddings

Methods presented in this chapter are very different in their learning architecture, in their training method or in the data they use. While each method has its own advantages and inconvenients, there exist common criteria to compare methods and know what are

the strengths and weaknesses of each one. Properties of different methods are reported in Table 3.1, according to four criteria:

- How much information is extracted and used from the training text corpus?
- How much information from external sources of knowledge is used during the training of the model?
- How long does it take to train the model?
- How large is the model architecture used by the method?

The strength of one method is represented by its number or "+" signs where more signs means a greater strength. Minus signs "-" indicate weaknesses where a large number of signs indicates a strong weakness. The number of signs is not based on any objective measure but is the result of my personal view and opinion on each one of those methods.

	Use text corpus	Use external knowledge	Training time	Model size
Collobert and Weston [2008]	+	n/a	++	++
Collobert et al. [2011]	++	n/a	+	++
`word2vec` [2013]	++	n/a	+	+
`GloVe` [2014]	++	n/a	+	+
`fasttext` [2017]	+++	n/a	+	-
Yu and Dredze [2014]	++	+	-	+
Kiela et al. [2015]	++	++	-	+
`retrofitting` [2015]	n/a	++	+++	++
Sense embeddings [2015]	++	+	-	-
`ELMo` [2018]	++++	n/a	---	---
`BERT` [2019]	++++	n/a	----	----

Table 3.1: Strengths and weaknesses of different methods to learn word embeddings. For each criteria, the strength or weakness of one model is represented with plus (+) or minus (-) signs. Most of the values in the "Use external knowledge" column are set to n/a because most of the methods do not use external knowledge during training.

Like in Figure 3.5, similar methods have similar properties. Collobert and Weston [Collobert and Weston, 2008] use only cooccurrence information from windows of words and learn representations with a single hidden layer neural network. Their model is relatively small and fast to train because in 2008, training corpora were not as big as the ones used now and the number of parameters in their neural network is small. Collobert et al. [Collobert et al., 2011] extend this approach by using complete sentences as inputs in the neural network architecture instead of only using windows of words. This allows to capture more linguistic information in word embeddings because longer cooccurrence relations are used but it takes a bit longer to train the model. `word2vec` and `GloVe` are similar because they both use cooccurrence information from the text but they are trained on larger text corpus, so the additional information they get about rare words (which are usually not found in small corpus) is traded for a longer training time. Moreover, their model is bigger than the one used in [Collobert and Weston, 2008, Collobert et al., 2011] because the size of the vocabulary is largely increased as the size of the corpus is bigger. So the number of word embeddings to store during the training phase is also largely increased. `fasttext` has the advantage of using more information from the text corpus by considering morphological information with subwords, but it increases the model size because it needs to also learn one embedding per subword in addition to learning the embeddings of words.

Methods which use additional external sources of information [Yu and Dredze, 2014, Kiela et al., 2015] use the same information from the text corpus (cooccurrence relations) as `word2vec` but they are slower to train because of the additional information used during training. Kiela et al. [2015] has the advantage of using multiple external sources of knowledge, not a single one like in Yu and Dredze [2014]. `retrofitting` is not a learning method but a post-training step so it does not have to iterate over a large text corpus to improve word embeddings and is therefore fast to use. It also uses several sources of external knowledge like Kiela et al. [2015]. Sense embeddings [Iacobacci et al., 2015] use additional information to perform word disambiguation but it makes the model bigger and longer to train as the embeddings for each sense of each word have to be learned.

Deep learning models like `ELMo` or `BERT` use bidirectional LSTM or transformers which are able to capture relations between words that are far apart from each other or that have complex context dependencies. Compared to models like `word2vec` or `GloVe`, they use much more information from the training corpus, which is one great strength of these approaches. Word representations learned by `ELMo` or `BERT` contain a lot of encoded linguistic information and the performances of models which use them in downstream tasks are largely increased. However, their model architecture is large and complex. `ELMo` uses 2 layers of bidirectional LSTM while `BERT` has an even deeper architecture with 12 layers of transformers. The number of computations happening inside bi-LSTM or transformers is also much greater than for a single hidden layer neural network. This has the consequence of making these models much slower to train. The computational cost of `ELMo` or `BERT` is a strong weakness of these approaches because it is difficult to replicate or to use them especially when computing resources are limited.

All the presented methods have their own strengths and weaknesses. While simple models can be fast to train, they do not use a lot of linguistic information during training. Additional knowledge can be added into models to increase the quantity of encoded information (either with external sources of knowledge or by considering subwords or longer contexts) but using too much additional information can lead to large model architectures that are difficult to train because they require a lot of computing power or long training times. To learn word embeddings, one would have to make a choice between having a model that can capture a lot of linguistic information and having a model fast to train. A good trade-off would be to have a model that uses information from the text corpus with cooccurrences relations like in `word2vec` and which also incorporates knowledge from an external source of information but without adding any hard computational overhead so the model is still fast to train. One contribution of this thesis is a novel model to learn word embeddings which answers those requirements (see Chapter 5).

3.6 Conclusion

In this chapter, we have presented common methods to learn word embeddings, and described their main respective strengths and weaknesses.

Basic methods use a sliding context window across a corpus to extract cooccurrence information between words, and update their corresponding word embeddings with a single hidden layer neural network [Collobert and Weston, 2008, Collobert et al., 2011, Mikolov et al., 2013a,b]. These methods have demonstrated the usefulness of learning word vectors to represent linguistic properties of words and the impact that good representations can have when they are used in downstream models to solve NLP tasks. However, these methods are only based on the cooccurrence information of words in texts to learn word embeddings. Specific linguistic knowledge like synonymy is usually not found in training corpus and therefore is not captured by the model and incorporated into the embeddings.

To overcome this shortcoming, several other approaches have been proposed to use additional information during training. Bojanowski et al. [2017] use subword information while Yu and Dredze [2014] and Kiela et al. [2015] use external sources of knowledge to get information about words like hypernyms or synonyms. Additional information is then incorporated into a model like the `word2vec` Skip-gram model. Learned word representations are able to capture this additional linguistic information, which is beneficial for downstream models that use them because their performances on NLP tasks is improved compared to models that use basic word vectors learned from a classic `word2vec` model.

More recent approaches use deep model architectures to learn word representations with either multiple bidirectional LSTM layers [Peters et al., 2018] or multi-layer transformers [Devlin et al., 2019]. These models have the advantage of computing contextual word representations: a word is no longer associated to a single fixed word vector like in the `word2vec` model but a unique vector for each word occurrence is computed by the deep architecture depending on the context in which the word occurs. This approach is closer to what happens in a language since the meaning of a word can vary depending on its context. This has been observed in practice as the results on downstream tasks have been largely improved when models use those contextual word representations. However, their uses come with a rather expensive price to pay. Deep architectures use a lot of computations to train but also to produce contextual word embeddings so it greatly reduces their uses on devices with limited computing resources. This is an important problem as the applications using word embeddings are more and more used on those types of low-resource devices in order to off-load server-side computations or run applications offline.

Two opposite strategies seem to stand out: models that use basic linguistic information like word cooccurrences are simple but lack some information in their learned representations while models that embed large quantity of linguistic information have richer word representations but use large architectures which are usually not able to run on low-resource devices. One may then wonder if the best parts of the two strategies can be combined, that is: is it possible to create a model that use large quantity of information during training but is small enough to be used on low-resource devices? While such a model is yet to be found, there exist some methods to reduce the size and the complexity of large models so they can be used and run on low-resource devices. The next chapter provides on overview of these methods.

Chapter 4

Reducing the Size of Representations

4.1 Introduction

In Chapter 3, we gave an overview of common methods to learn word embeddings. These methods can leverage different kinds of information with diverse model architectures:

- cooccurrence relations of words from a large text corpus to learn word embeddings with a single hidden layer neural network [Collobert and Weston, 2008, Collobert et al., 2011, Mikolov et al., 2013b];
- additional knowledge from external sources of information to capture more linguistic information into word representations [Yu and Dredze, 2014, Kiela et al., 2015, Faruqui et al., 2015a];
- or long contextual dependencies of words with deep architectures [Peters et al., 2018, Devlin et al., 2019].

The objective of these methods is to map each word w of a vocabulary $\mathcal{V}$ to a vector representation $\mathbf{v}_w$:

$$(w, \mathbf{v}_w) \quad \forall w \in \mathcal{V};\ \mathbf{v}_w \in \mathbb{R}^d \tag{4.1}$$

where d is the *dimension* of the word embedding, *i.e.* the number of values in the vector $\mathbf{v}_w$. Word embeddings are usually stored in an *embedding matrix* after they are learned by a model and before they are reused into another model. For a vocabulary $\mathcal{V}$ of $N_\mathcal{V}$ words, the *embedding matrix* $\mathbf{M}$ is:

$$\mathbf{M} \in \mathbb{R}^{N_\mathcal{V} \times d} \tag{4.2}$$

where the i-th row is the embedding of the i-th word of the vocabulary. The size of the embedding matrix is linearly proportional to the size of the vocabulary and the number of dimensions of word vectors. As the training corpora get bigger and bigger over the years, the size of the vocabulary also increases and so is the size of the embedding matrix. As an example, for a vocabulary of $N_\mathcal{V} = 2,000,000$ words and vectors of 300 dimensions (common size in the literature) of 32-bits real values, it requires 2.4 gigabytes in memory space to save the embedding matrix (32 bits = 4 bytes). While this is not a problem when word embeddings are used in downtream models on large computing servers, it becomes difficult to use them directly on devices with limited memory such as smartphones.

Moreover, being able to run models that use word embeddings directly on low-resource devices presents other benefits of practical interest. In general, smartphones running NLP

applications are not powerful enough or do not have enough memory to run the models. So they send the data to process to large computing servers where the models are able to run. The servers perform the calculations and return the results to mobile devices. However, when smartphones send to servers user inputs (which can be confidential), servers can maliciously store those user inputs for later use instead of only storing them to compute the results and returning the results to mobile devices. With a model and an embedding matrix small enough to be run locally, the privacy of the user is preserved because the user input is not being sent anywhere. If the size of word embeddings is reduced, computations can be done locally, directly on the embedded device, without needing to send data to the servers. The benefits are twofold:

1. Embedded devices can run NLP applications offline, which reduces the computing load of servers and can cut down infrastructure costs as less servers are required.

2. Privacy is preserved since no user data is sent to servers.

This chapter provides an overview of methods to reduce the size of word representations used in downstream models so they can be deployed more easily on low-resource devices. It starts with a description of methods which reduce the size of the embedding matrix and then presents methods that change the way word embeddings are encoded in memory, which also results in the embedding matrix taking up less space in memory.

4.2 Reducing the size of the embedding matrix

As defined in Equation 4.2, the embedding matrix is an element of $\mathbb{R}^{N_\mathcal{V} \times d}$. To reduce its size, one can reduce one (or both) of the dimensions: reducing d will reduce the length of each embedding vector (see Subsection 4.2.1) while reducing $N_\mathcal{V}$ means that the number of vectors needed is decreased (see Subsection 4.2.2). Another existing family of methods is *distillation*, which transfers the knowledge contained in large vectors into smaller word vectors (see Subsection 4.2.3).

4.2.1 Reducing the number of vector dimensions

Common word embeddings methods found in the literature usually set a vector size of 300 [Mikolov et al., 2013b, Pennington et al., 2014, Bojanowski et al., 2017]. However, not all of the 300 values carry the same amount of linguistic information. For example, if a certain dimension has almost the same value in all vectors, then this dimension does not bring a lot of information to differentiate vectors or to find which ones are the most similar together. So this dimension is not really useful for any downstream models which use those vectors. The idea of ordering the dimensions by their level of encoded information and discarding the less important ones is at the root of *Principal Component Analysis* (abbreviated `PCA` thereafter), invented by Pearson [Pearson, 1901]. In `PCA`, the level of encoded information in a dimension is computed with its variance. The greater the variance, the more information a dimension encodes. But `PCA` does not only compute the variance for each dimension, it also finds a transformation that maps the original matrix $\mathbf{M} \in \mathbb{R}^{n \times m}$ into a new matrix where the first p dimensions ($p < m$) are the ones with the most encoded information. After the transformation has been learned and applied to the data matrix, only the first p columns are kept. This produces a new data matrix with less dimensions where only the most important ones have been kept. To find this transformation, `PCA` starts to compute the covariance matrix $\mathbf{C} \in \mathbb{R}^{m \times m}$ of the mean centered matrix of $\mathbf{M}$, written as: $\mathbf{M} - \overline{\mathbf{M}}$.

$$\mathbf{C} = cov(\mathbf{M} - \overline{\mathbf{M}}) = \frac{1}{n-1}(\mathbf{M} - \overline{\mathbf{M}})^\top (\mathbf{M} - \overline{\mathbf{M}}) \tag{4.3}$$

Then, eigenvalues and eigenvectors of the covariance matrix $\mathbf{C}$ are computed and only the p eigenvectors with the largest eigenvalues are kept to form a matrix $\mathbf{P} \in \mathbb{R}^{p \times m}$ (each eigenvector has a dimension of m). The new data matrix $\mathbf{Q} \in \mathbb{R}^{n \times p}$ with a reduced number of dimensions is computed as:

$$\mathbf{Q} = \mathbf{M}\mathbf{P}^{\top} \tag{4.4}$$

`PCA` can be applied to any kind of vectors. When applied to word embeddings, the number of dimensions of word vectors is decreased. This has the consequence of producing smaller vectors which speed up computations in downstream models that use them. `PCA` allows one to reduce the number of dimensions while not degrading the performances of word vectors in downstream tasks [Lebret and Collobert, 2014].

While `PCA` is a good method to reduce the number of dimensions of word vectors, it only uses linear projections to find how to transform data into a dimension-reduced vector space. It therefore cannot capture non-linear dependencies of data present in the original space. To overcome this shortcoming, Maaten and Hinton [Maaten and Hinton, 2008] have proposed a new method, called `t-SNE`. It models data of a high-dimensional space with a probability distribution between all pairs of vectors. It captures relations between close vectors, *i.e.* the local structure of data like neighbors, as well as the global structure like clusters. `t-SNE` creates a new vector space for data (with a reduced number of dimensions) that follows this learned probability distribution. `t-SNE` is able to reduce the number of dimensions of word vectors. However, it is mainly used to visualize data, *i.e.* to project high-dimensional vectors into a 2 or a 3 dimensional space. When data is projected into a d-dimensional vector space with $d > 3$, the local structure of data is not well preserved because the probability distribution is heavy-tailed [Maaten and Hinton, 2008].

Raunak et al. [Raunak et al., 2019] propose another method to reduce the number of dimensions of word embeddings. They combine `PCA` with `PPA`, a `P`ost `P`rocessing `A`gorithm applied on learned word embeddings to project the embeddings away from the most dominant directions [Mu and Viswanath, 2018]. `PPA` applies `PCA` to the mean centered embedding matrix to get the D principal component directions and then remove those D directions from all the word vectors. `PPA` does not change the number of dimensions of the word vectors but it makes word embeddings more discriminative as common dominant directions are removed. Indeed, it has been observed that pre-trained word embeddings have usually several common mean vectors that are not informative when computing similarity (or dissimilarity) between vectors. The algorithm developed by Raunak et al. starts to apply `PPA` on a word embedding matrix to remove the non informative directions, then applies `PCA` to reduce the number of dimensions and applies again `PPA` on the reduced vectors. Their final embeddings have 50% less dimensions compared to the original word vectors and perform better on a word semantic similarity task compared to word embeddings that only have been processed with a single `PCA` step (+27% on average).

Reducing the number of dimensions leads to similar or better scores in downstream tasks, which can be counter-intuitive because each dimension in word embeddings contain information learned during the training step so transforming vectors into a lower dimensional space should cause a loss of information and worse results. This behavior can be explained by the Johnson–Lindenstrauss lemma [Johnson and Lindenstrauss, 1984]:

Lemma 4.1 (Johnson–Lindenstrauss lemma)**.** For any $0 < \epsilon < 1$ and any integer $N_{\mathcal{V}}$, let k be a positive integer such that:

$$k \geq \frac{4}{\frac{\epsilon^2}{2} - \frac{\epsilon^3}{3}} \log(N_{\mathcal{V}}) \tag{4.5}$$

Then for any set X of $N_{\mathcal{V}}$ vectors in $\mathbb{R}^d$, there exists a map $f : \mathbb{R}^d \to \mathbb{R}^k$ such that for any vectors $\mathbf{u}, \mathbf{v} \in X$:

$$(1-\epsilon)\left\|\mathbf{u}-\mathbf{v}\right\|^2 \leq \left\|f(\mathbf{u})-f(\mathbf{v})\right\|^2 \leq (1+\epsilon)\left\|\mathbf{u}-\mathbf{v}\right\|^2 \tag{4.6}$$

This lemma shows the existence of a map function f that can project vectors into a lower dimensional space while keeping the distance between vectors as close as the distance between the original vectors by at most ϵ. The distance between vectors is used to compute their similarity, so if the similarity is preserved in the dimensionally reduced space then the performances in tasks like document classification or sentiment analysis which mainly use vector similarities to perform computations are not much affected.

4.2.2 Reducing the number of vectors

As explained at the beginning of this subsection, to reduce the size of the embedding matrix $\mathbf{M} \in \mathbb{R}^{N_{\mathcal{V}} \times d}$, one can either reduce the value of d (the number of dimensions in each word embedding) or reduce the value of $N_{\mathcal{V}}$ (the number of vectors in the embedding matrix). Usually, the number of vectors is much larger than the number of dimensions ($N_{\mathcal{V}} \gg d$). Indeed, it is common to find pre-trained word embeddings with a number of vectors in the order of millions while d is typically between 100 and 300. Moreover, in a pre-trained word embedding matrix, two similar words are mapped to similar vectors (*e.g.* in `fasttext` pre-trained embeddings, vectors of "road" and "highway" have similar values). Therefore, one can wonder if it would be possible to reduce the number of vectors in the embedding matrix by associating a common vector to two similar words instead of associating two similar (but different) vectors to each one of them. This observation is at the root of the method proposed by Shu and Nakayama [Shu and Nakayama, 2018]. Instead of associating each word w to its own embedding vector $\mathbf{v}_w$, it is associated with a code C_w composed of M integers between 1 and K:

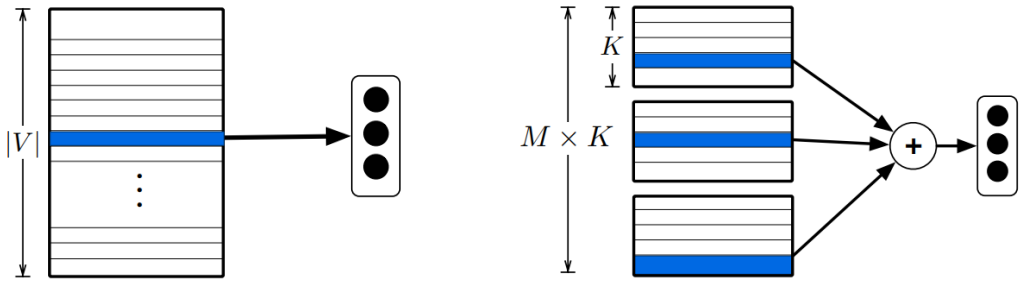
$$C_w = (C_w^1,\ C_w^2,\ \ldots,\ C_w^M);\ \forall i,\ C_w^i \in [1, K] \tag{4.7}$$

Instead of having one vector for each word, the method creates M codebooks (noted as $B_1,\ B_2,\ \ldots,\ B_M$), each one composed of K vectors (so $M \times K$ vectors in total). The vector $\widetilde{\mathbf{v}}_w$ of a word w can be computed by using the integers of its code C_w and summing the C_w^1-th vector of the first codebook B_1, the C_w^2-th vector of the second codebook B_2, etc. This can be written as:

$$\widetilde{\mathbf{v}}_w = \sum_{i=1}^{M} B_i[C_w^i] \tag{4.8}$$

where B_i is the i-th codebook. This method of summing codebook vectors is illustrated in Figure 4.1. Vectors in codebooks as well as the codes C_w composed of the indexes C_w^i are learned with an autoencoder architecture (see Subsection 2.4.2 in Chapter 2 for a description of autoencoders). The objective of the model is to construct compositional vectors $\widetilde{\mathbf{v}}_w$ as close as possible to original vectors $\mathbf{v}_w$ for each word w from the vocabulary $\mathcal{V}$ of all words in the embedding matrix $\mathbf{M}$. The autoencoder model finds the optimal codebook vectors $\hat{B}$ and codes $\hat{C}$ by solving the following optimization problem:

$$(\hat{B}, \hat{C}) = \underset{B,C}{\arg\min} \frac{1}{|\mathcal{V}|} \sum_{w \in \mathcal{V}} \left\|\widetilde{\mathbf{v}}_w - \mathbf{v}_w\right\|^2 = \underset{B,C}{\arg\min} \frac{1}{|\mathcal{V}|} \sum_{w \in \mathcal{V}} \left\|\sum_{i=1}^{M} B_i[C_w^i] - \mathbf{v}_w\right\|^2 \tag{4.9}$$



(a) Classic embbedings: one vector per word.

(b) Codebook embeddings: each word vector is a combination of a certain number of base vectors.

Figure 4.1: Model used by [Shu and Nakayama, 2018] to reduce the number of word vectors. On the left, the classic word embeddings representation where each word is associated to its own vector. On the right, there is not a single vector associated to each word but a bank of $M \times K$ base vectors. The vector of each word is be computed by summing a certain combination of those base vectors.

Storing base vectors (the codebooks B_i) and index codes (the C_w codes) instead of all word vectors can drastically reduce the size of the embedding matrix. Indeed, with 32 codebooks of 16 vectors (so 512 base vectors in total), the size of the `GloVe` embedding matrix can be reduced by 98% (1.23 megabytes vs. 78 megabytes) without any performances loss in sentiment analysis or machine translation tasks [Shu and Nakayama, 2018].

Chen et al. [Chen et al., 2018] propose a similar idea to reduce the number of vectors in the embedding matrix. In the same way, each word w is associated to a code C_w of length M:

$$C_w = (C_w^1,\ C_w^2,\ \ldots,\ C_w^M);\ \forall i,\ C_w^i \in [1, K] \tag{4.10}$$

but instead of summing the base vectors corresponding to each C_w^i like in Shu and Nakayama [2018], a linear transformation is applied to compute the word vectors $\widetilde{\mathbf{v}}_w$:

$$\widetilde{\mathbf{v}}_w = H \cdot \left[\sum_{i=1}^{M} B_i[C_w^i]\right]^\top \tag{4.11}$$

where H is a $\mathbb{R}^{d \times d}$ matrix and d is the dimension of base vectors from the codebooks B_i. Chen et al. [2018] also propose another version where base vectors are first passed to a LSTM and then combined with a linear transformation:

$$\widetilde{\mathbf{v}}_w = H \cdot \left[\sum_{i=1}^{M} h_w^i\right]^\top \tag{4.12}$$

where:

$$(h_w^1,\ h_w^2,\ \ldots,\ h_w^M) = \text{LSTM}(B_1[C_w^1],\ B_2[C_w^2],\ \ldots,\ B_M[C_w^M]) \tag{4.13}$$

Chen et al. [Chen et al., 2018] learn code embeddings in a end-to-end manner: they learn codes and base vectors that best work for a given task. This is the main difference compared to the method proposed by Shu and Nakayama [2018] which first reduces the size of the embeddings and then "freeze" code embeddings in the task learning step. However, the compressing ratio is the same for both methods (size of the embedding matrix is reduced by 98% without any loss of performances in downstream tasks).

4.2.3 Distilling information into smaller embeddings

Deep architectures like `ELMo` [Peters et al., 2018] or `BERT` [Devlin et al., 2019] are able to learn word embeddings that capture many linguistic information such as meanings, syntax or connotation of words thanks to the large number of parameters in their models. However, we have seen in Section 3.5 of Chapter 3 that those models are large and it is difficult to use them on running environments with limited memory space and small computing power. Moreover, large models are often complex and over-parametrized to learn as much as possible linguistic information to solve multiple tasks, but in practice, not all the parameters are required to solve a given task. For example, in a task like sentiment analysis, the connotation of a word is more informative than its syntactic properties. An approach to solve this problem is *distillation*. It consists in distilling the most useful information required to solve the task (*e.g.* the connotation of words in sentiment analysis) into smaller models or smaller representations to remove the information not useful for the task. Distillation can be applied to:

- the entire model: to reduce the number of parameters required to solve the task;
- word representations: to reduce the size of representations and remove non informative values not required to solve the task. This also has the consequence of reducing the size of the downstream model that uses smaller representations (see Figure 4.2).

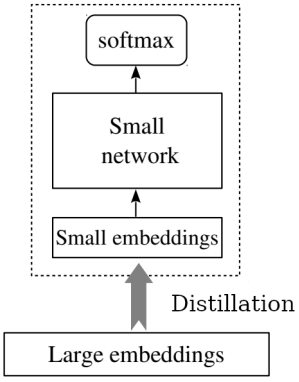


Figure 4.2: Large word embeddings learned by large models are tranformed into smaller embeddings before being used to solve a downstream task. Linguistic information contained in large word embeddings is *distilled* into the smaller ones.

In the context of this thesis, we are more interested in reducing the size of word representations rather than reducing the size of models to solve a given task.

Distillation of trained models. Given a large model (like a neural network) trained on huge datasets, the idea is to train a smaller model that replicates the predictions of the large model. Hinton et al. [Hinton et al., 2015] propose to transfer the knowledge learned by the parameters of a large neural network to a smaller one by minimizing the distances between the output values of the two networks. In this case, output values represent the probability of belonging to each possible class, so close output values means that the predictions of the small network are similar to the predictions of the large network. `DistilBERT` [Sanh et al., 2019] is a distilled version of the `BERT` model. The same architecture is kept but half of the layers are removed. `DistilBERT` is trained to minimize three different objective functions: to minimize distances between output probabilities of `DistilBERT` and `BERT`, to minimize prediction errors on annotated examples and to minimize cosine distances between internal hidden representations of `DistilBERT` and `BERT`.

Distillation of trained word embeddings. High dimensional word vectors trained on a large text corpus with models like `word2vec` or `GloVe` are transformed into embeddings of smaller dimensions, which are then fed into a small model to solve a downstream task (see Figure 4.2). The linguistic knowledge captured by large word embeddings is *distilled* into smaller ones. Mou et al. [Mou et al., 2016] propose to distill the knowledge contained in word vectors $\mathbf{v}_w$ of $d = 300$ dimensions into word vectors $\widetilde{\mathbf{v}}_w$ of $d' = 50$ dimensions. The transformation is done with a non-linear neural network layer:

$$\widetilde{\mathbf{v}}_w = f(\mathbf{W}\mathbf{v}_w{}^\top + \mathbf{b}) \tag{4.14}$$

where $\mathbf{W} \in \mathbb{R}^{d' \times d}$, $\mathbf{b} \in \mathbb{R}^{d'}$ and f is a non-linear function $f : \mathbb{R}^{d'} \to \mathbb{R}^{d'}$. The parameters $\mathbf{W}$, $\mathbf{b}$ and the weights of the small downstream neural network which uses the vectors $\widetilde{\mathbf{v}}_w$ are learned to minimize prediction errors of the output layer on annotated examples. Distilling knowledge into smaller word embeddings is different from learning small word embeddings from the start. With a larger number of dimensions, vectors have more freedom to capture more linguistic information from a text corpus compared to small vectors which have more constraints [Yin and Shen, 2018]. Additional captured information can then be refined with distillation to help solving the downstream task, which is not possible when small vectors have been used from the start as they have not captured as much information.

In this section, we have presented methods that reduce the size of the embedding matrix by reducing its number of parameters, either by reducing the number of dimensions or the number of vectors, or by distilling linguistic information into smaller vectors. The next section presents methods which use a different approach to reduce the memory size of the embedding matrix: by changing the way to encode word vectors.

4.3 Encoding word embeddings as integer vectors

To reduce the size of the embedding matrix and use it more easily in downstream models on memory-limited devices, we have seen that one can reduce its number of dimensions (Subsection 4.2.1) or its number of word vectors (Subsection 4.2.2) or distill the linguistic information into smaller word vectors (Subsection 4.2.3). In those approaches, the size of the embedding matrix can be reduced on average by 98%. However, this size-reduced matrix still uses real-valued vectors. Vector operations done with real values are computationally expensive because in modern processors, real-valued operations require more processor cycles than operations done with integers. Therefore, a size-reduced embedding matrix can be used in a model on a memory-limited device but it will be slow to run as those devices often have low computing power, especially for real-valued vector operations.

Real values (thereafter named *float* or *floating-point*) are generally encoded with 32 bits on common machines. If word embeddings are not composed of real values (on 32-bits) but of small integer values (on 16-bits), the memory size of embeddings is divided by 2 but the computations of vector operations are speeded up by a factor greater than 2 because integer operations are highly optimized on processors. Therefore, changing the representation space of word embeddings and how to encode the information captured by those vectors can provide two benefits at the same time: a smaller need of memory space and faster vector operations. An alternative representation space for word embeddings is the binary space, where each value of word vectors is either 0 or 1 (a bit) instead of a real value. Associating binary vectors to words allows one to speed up computations as vector operations can be done with bitwise operators instead of floating-point arithmetic, *e.g.* computing the distance between two binary vectors only requires a `XOR` and a `popcount()`[1] operations

[1] `popcount(n)` returns the number of bits set to 1 in `n`.

(which are both performed with a single CPU cycle due to hardware optimizations in modern processors) while it requires $\mathcal{O}(n)$ floating-point multiplications and additions for n-dimensional real-valued vectors to compute their cosine distance. To take advantage of fast CPU optimized bitwise operations, the size of binary vectors *has to be in adequacy with register sizes* (64, 128 or 256 bits). When this criteria is met, computations of vector operations are much faster [Norouzi et al., 2012, Subercaze et al., 2015].

This section details several methods used to produce word embeddings where vector values are encoded as integers or as single bits. This has the advantage of both speeding up computations of vector operations as well as reducing the memory size of the embedding matrix. The methods can be separated into two main categories: hashing (described in Subsection 4.3.1) and quantization (described in Subsection 4.3.2).

4.3.1 Hashing of vector representations

Hashing is a family of methods to transform data from a high-dimensional space into a low-dimensional space in order to obtain more compact data such that the similarity between two compact items reflect their similarity in the original space. One would therefore use the similarity of the compact space to find similar items (*e.g.* in a clustering task) because it is much faster to compute than the similarity of the original high-dimensional space. Charikar [Charikar, 2002] propose a *Locality Sensitive Hashing* (`LSH`) method to transform real-valued vectors of $\mathbb{R}^d$ into a binary space (also called the *Hamming space*) such that the similarity between vectors is preserved. They draw a random vector $\mathbf{r}$ from a Gaussian distribution over $\mathbb{R}^d$ (*i.e.* each dimension of $\mathbf{r}$ is drawn from a Gaussian distribution over $\mathbb{R}$) and define a hash function $h_\mathbf{r}$ for each vector $\mathbf{u} \in \mathbb{R}^d$ as follows:

$$h_\mathbf{r} = \begin{cases} 1 & \text{if } \mathbf{r} \cdot \mathbf{u} \geq 0 \\ 0 & \text{if } \mathbf{r} \cdot \mathbf{u} < 0 \end{cases}$$

They draw K random vectors from the Gaussian distribution over $\mathbb{R}^d$. Each random vector $\mathbf{r}_k$ defines a hash function $h_{\mathbf{r}_k}$ and the output of each function are concatenated to produce a binary vector of K bits for each vector $\mathbf{u} \in \mathbb{R}^d$. Several works have improved locality sensitive hashing either by choosing random vectors from a different distribution [Datar et al., 2004] or by using different hashing functions [Andoni and Indyk, 2006].

Salakhutdinov and Hinton [Salakhutdinov and Hinton, 2007] introduce the concept of *Semantic Hashing* where the objective is to find a binary code for each document so semantically similar documents have similar codes. Given a query document, its closest neighbors are found by only looking at the documents with similar codes instead of looking at all the documents. Binary codes are learned with a generative architecture that uses Poisson distributions to map Bag-of-Words representations of documents to binary codes. Binary codes are then fine-tuned to be able to reconstruct the BoW of each document. Weiss et al. [Weiss et al., 2009] propose a method to learn binary codes whose Hamming distances approximate the distance of vectors from the original vector space $\mathbb{R}^d$. Finding binary codes that exactly preserve this distance is a NP-hard problem so the authors relax the problem and remove the constraint of having binary codes during training. They learn real-valued codes with a spectral decomposition (with eigenvectors and eigenvalues) of the similarity matrix $\mathbf{W}$ (where $\mathbf{W}_{i,j}$ is the similarity between the i-th and the j-th vectors from the original vector space). Values of eigenvectors are then tranformed to binary features with threshold functions, which gives a binary code for each vector. `NASH` [Shen et al., 2018] is another architecture used to transform real-valued vectors into binary codes for fast similarity searches of documents. Their model learns to associate the Bag-of-Words representation of a document to a latent vector, which is used to recreate the Bag-of-Words

representation of this document. During training, latent vectors are composed of real values between 0 and 1 such that latent vectors follow a multivariate Bernoulli distribution. Once the latent vectors have been learned, a threshold function is applied to transform them into binary codes. Binary latent vectors contain linguistic information because they have been trained to reconstruct the Bag-of-Words representations of documents.

Hashing methods are good approaches to transform real-valued vectors into binary codes which are both smaller in memory space and allows one to compute vector operations faster. The different models presented in this subsection can either use random projections [Charikar, 2002, Datar et al., 2004] or a whole dedicated architecture [Shen et al., 2018] to find binary codes that preserve the similarities of original vectors. They are mainly used to transform the vectors of documents in order to speed up the search of similar documents but rarely to transform word embeddings for use in downstream tasks because they often fail to fully preserve semantic similarities [Xu et al., 2015].

4.3.2 Word embeddings quantization

Common models to learn word embeddings perform operations with real values encoded with 32 or 64 bits [Mikolov et al., 2013b, Pennington et al., 2014] but other recent methods are able to learn with *half-precision* float *i.e.* real values encoded with 16 bits [Micikevicius et al., 2018, Seznec et al., 2018]. Quantization is a generic method to reduce the number of bits needed to encode real values. For example, if real values have to be encoded with only 4 bits (which gives 16 possible values because $2^4 = 16$), values need to be rescaled or thresholded to be one of the 16 possible values. When the values of vectors are encoded with less bits, they require less memory space to be stored. This subsection presents different methods to quantize word embeddings and therefore reduce their size in memory.

Ling et al. [Ling et al., 2016] shows that word embeddings encoded with 64-bits real values can be quantized with values encoded with 8 bits without any loss of performances in downstream tasks like word semantic similarity. The quantization is achieved by rounding the values of vectors. To round vector values to n-bits precision, values in each vector dimension are first scaled so all the vectors values for this dimension lie between $[-2^{n-1}, 2^{n-1}]$. The following function is then applied to each value $\mathbf{x}_{[i]}$ of each vector $\mathbf{x}$:

$$\text{round}(\mathbf{x}_{[i]}) = \begin{cases} \lfloor \mathbf{x}_{[i]} \rfloor & \text{if } \lfloor \mathbf{x}_{[i]} \rfloor \leq \mathbf{x}_{[i]} \leq \lfloor \mathbf{x}_{[i]} \rfloor + \frac{1}{2} \\ \lfloor \mathbf{x}_{[i]} \rfloor + 1 & \text{if } \lfloor \mathbf{x}_{[i]} \rfloor + \frac{1}{2} < \mathbf{x}_{[i]} \leq \lfloor \mathbf{x}_{[i]} \rfloor + 1 \end{cases}$$

Each rounded value is then encoded with n bits (since each value is between -2^{n-1} and 2^{n-1}, n bits are enough to encode it). This method allows one to choose the value of n, the number of bits needed to encode each value. This method can also be used to produce binary vectors when n is set to 2. However, Ling et al. report that the performances of binary embeddings (when $n = 2$) are worse than the original 64-bits real-valued vectors whereas the performances stay the same when n is set to 8.

Product Quantization [Jegou et al., 2010] is another method to quantize real-valued vectors mainly used for fast similarity searches. Let $\mathbf{v}$ be a d-dimensional vector ($\mathbf{v} \in \mathbb{R}^d$). The vector is split into subvectors $\mathbf{u}_i$ of m dimensions (d needs to be divisible by m).

$$\mathbf{v} = (\underbrace{\mathbf{v}_1,\ \mathbf{v}_2,\ \ldots,\ \mathbf{v}_m}_{\mathbf{u}_1},\ \underbrace{\mathbf{v}_{m+1},\ \ldots,\ \mathbf{v}_{2m}}_{\mathbf{u}_2},\ \ldots,\ \underbrace{\mathbf{v}_{d-m+1},\ \ldots,\ \mathbf{v}_d}_{\mathbf{u}_{d/m}}) \tag{4.15}$$

The split process is repeated for each vector to quantize. Subvectors are quantized separately. For each subvector $\mathbf{u}_i$, a distinct K-means clustering is done on all the corresponding subvectors of all vectors, *i.e.* K-means is applied on all the $\mathbf{u}_1$ subvectors,

another K-means is applied on all the $\mathbf{u}_2$ subvectors, etc. The centroids learned by the i-th K-means algorithm are stored in the i-th codebook B_i. The value of K (the number of centroids used in each K-means) is selected to be $K = 2^b$ where b is the number of bits chosen to encode each subvector. This gives a number of possible centroids equal to 2^b so each centroid can be assigned to a different integer identifier (ID) in $[0, 2^b - 1]$. After all the K-means and the codebooks of centroids have been learned (there are d/m K-means algorithms to perform), each subvector $\mathbf{u}_i$ is assigned to the ID of its closest centroid among the corresponding codebook of centroids B_i. The ID of each subvector are concatenated to obtain a final code for each vector $\mathbf{v}$. Jegou et al. [2010] use the product quantization (PQ) method for fast similarity retrieval among the description vectors of images. Real-valued vectors are transformed to 64-bit binary codes and provide a better recall@1 (*i.e.* the number of times the closest vector in the original space is the closest vector in the binary space) than other methods like spectral hashing [Weiss et al., 2009].

An original use of product quantization is made in `fasttext.zip` [Joulin et al., 2016a] to transform word embedding into compact codes for a more efficient memory text classification architecture. The same method as in product quantization is used to get a binary code for each word, which are then used in a downstream classification model composed of a single hidden layer neural network. However, computations performed inside the downstream model are not done with binary codes. To compute operations between two binary codes, the codes are first *de-concatenated* to get the ID of each centroid used to quantize each subvector $\mathbf{u}_i$. Vector operations are then performed with the respective centroid vector of each subvector $\mathbf{u}_i$. The objective of `fasttext.zip` is to reduce the size of the model for a classification task, not to speed up computations between vectors which is why operations are not done with binary codes but with real-valued vectors (the centroid vectors). With further optimization tricks like keeping only the most discriminative words of the vocabulary for the task, `fasttext.zip` can reduce the size of the classification model by a factor of 1000 with a loss of performances around 0.8% for a document classification task.

In product quantization, vectors are split sequentially: subvectors $\mathbf{u}_1$ use the dimensions from 1 to m, subvectors $\mathbf{u}_2$ use the dimensions from $m + 1$ to $2m$, etc. However, the quantization distortion (the difference between quantized vectors and original ones) can be reduced by reorganizing the dimensions as the distances between vectors and their assigned centroid is dependent on the values of vectors and thus on the range of values in each dimension. Ge et al. [Ge et al., 2013] propose an *Optimized Product Quantization* (OPQ) method in which they learn the way to reorganize the vector space to split subvectors in addition to learning the centroids. Re-ordering the dimensions of vectors $\mathbf{v} \in \mathbb{R}^d$ can be represented by an orthogonal matrix $\mathbf{R} \in \mathbb{R}^{d \times d}$. The authors learn both the matrix $\mathbf{R}$ and the codebooks of centroids B_i that minimize the quantization error:

$$\min_{\mathbf{R}, B_1, \ldots, B^{d/m}} \sum_{\mathbf{v}} \left\| \mathbf{v} - c(\mathbf{u}_1, \ \ldots, \ \mathbf{u}_{d/m}) \right\|^2 \tag{4.16}$$

where $\mathbf{u}_i$ is the i-th subvector of the rotated vector $\mathbf{Rv}$, c is the concatenation of the centroid vectors assigned to each $\mathbf{u}_i$, and $\mathbf{R}$ is an orthogonal matrix such that $\mathbf{R}^\top \mathbf{R} = \mathbf{R}\mathbf{R}^\top = I$. This objective function is minimized by alternatively optimizing two steps: in the first step, the matrix $\mathbf{R}$ is freezed and only the codebooks are learned with the classic product quantization method; in the second step, the codebooks and the centroids are freezed and only $\mathbf{R}$ is learned, which is equivalent to find the rotation of vectors that minimize the distance between their subvectors and their closest respective centroid vector (which are freezed). Compared to product quantization, OPQ greatly increases the recall rate with 64-bit binary codes (0.71 against 0.41 for PQ on recall@100) on MNIST images.

Quantization methods are mainly used for fast retrieval. Indeed, most of the methods presented in this subsection use quantized vectors for fast similarity searches [Jegou et al.,

2010, Ge et al., 2013, Ling et al., 2016]. The compact representations are not learned to be used in downstream models to solve NLP tasks. Even in `fasttext.zip` which uses product quantization to reduce the size of a document classification model, computations are not performed with quantized vectors but with the real-valued cluster vectors used to learn the quantized vectors. Since quantized vectors are not learned to preserve the linguistic properties of original word vectors but rather to preserve the similarity between vectors, some linguistic information is lost during quantization. For example, it has been shown that real-valued word embeddings are able to capture word analogies [Mikolov et al., 2013b] which indicates that vector dimensions can represent specific linguistic relations. But in binary vectors, one bit (or a group of bits) cannot be associated to a specific linguistic feature. Faruqui et al. [Faruqui et al., 2015b] propose a method to quantize word embeddings so dimensions in binary vectors are more interpretable. Binary vectors are generated by learning a linear transformation that maps pre-trained word embeddings into a larger dimensional space (usually with 10 times more dimensions) where the values of vectors are positive and 90% of them are set to 0 (we call them *sparse* vectors). Binary vectors are then obtained by setting all the non-zero values to 1. Faruqui et al. show that sparse binary vectors have bits symbolizing specific linguistic features. For example, binary vectors associated to words of animals have bits in common that other vectors do not have. While sparse binary vectors are useful to find interpretable features, the size reduction ratio is not as high as in other quantization methods. Sparse binary vectors have a length of 3,000 bits while product quantization or optimized product quantization learn binary codes which have a length between 64 and 256 bits. Therefore, the approach of Faruqui et al. is interesting from a linguistic point of view because of the interpretability of dimensions in vectors, but is limited from a memory-efficient storage point of view.

4.4 Conclusion

Word embeddings are real-valued vectors learned to capture linguistic properties of words. When the number of vector dimensions is high (most common methods use vectors of 300 dimensions) or when the number of words to represent is important (vocabulary can contain millions of words), the embedding matrix which contain the embeddings of all the words of the vocabulary becomes large and prevents downstream model which use it to be run on memory-limited devices such as smartphones. In this chapter, we have presented methods to reduce the size of the embedding matrix as well as methods to learn more compact word embeddings with alternative encodings of values which also have the benefit of speeding up vector computations. Methods to reduce the size the embedding matrix can be divided into two main categories:

- by reducing the dimensions of the embedding matrix;
- by changing how the values of word embeddings are stored in memory.

To reduce the dimensions of the embedding matrix, one can either reduce the number of dimensions of word embeddings or reduce the number of vectors in the matrix. `PCA` [Pearson, 1901] or `t-SNE` [Maaten and Hinton, 2008] are common methods to reduce the number of dimensions of vectors by finding a transformation that keeps only the k most informative directions of vectors. Another method to reduce the number of dimensions is to *distill* the knowledge captured by large word embeddings into smaller word embeddings with either a neural network architecture [Mou et al., 2016] or a reduced version of an existing model [Sanh et al., 2019]. To reduce the number of vectors stored in the embedding matrix, a solution is to consider that each word embedding can be written as a combination of a certain amount of base vectors. Instead of storing one vector for each word, only the

base vectors and the list of base vectors to combine for each word are stored, which reduces the number of vectors to store. Shu and Nakayama [2018] propose to sum base vectors while Chen et al. [2018] propose to use linear combinations of base vectors. In both methods, using base vectors (typically 512 base vectors) reduces the size of the embedding matrix by 98% without any loss of performances of word embeddings in downstream tasks.

Another type of approaches to reduce the memory size of word embeddings is to change how their values are encoded. In common word embeddings models, vectors use real values encoded with 32 bits. Ling et al. [2016] propose to round values so only the first 8 bits are kept. This has the consequence of reducing the size of the embedding matrix by 4 without any loss of performances in downstream tasks. Product quantization [Jegou et al., 2010] is another method that transforms real-valued word vectors into binary codes by applying K-means algorithms on separate subvectors and concatenating the ID of the closest centroid of each subvector to form a binary code for each word. This method has been applied to word embeddings to reduce the size of a document classification model [Joulin et al., 2016a]. Binary word embeddings use less memory space and have the advantage of speeding up vector computations as operations on binary vectors are much faster than vector operations performed with real-valued vectors [Norouzi et al., 2012, Subercaze et al., 2015].

We have seen in the previous chapter that *good* word embeddings encode a lot of linguistic information so downstream models which use them have more knowledge about the language and can perform better in NLP tasks. But they require large models and representations that cannot be used in memory-limited environments. Methods presented in this chapter allows one to reduce the size of word representations in memory by trying to minimize the loss of information during the size reduction step. The compact word embeddings are mainly used for retrieval and fast similarity searches, not for use in downstream models to solve NLP tasks. The challenge is therefore to find methods that can transform large word embeddings which contain a lot of linguistic information into compact representations which preserve this linguistic information and are small enough to be used on those limited environments. This challenge is one of the main topic of this thesis and Chapter 6 presents a contribution that answers it with a new method to transform word embeddings into compact binary codes that can be used in downstream NLP tasks.

Part III

Improved Methods to Learn Word Embeddings

Chapter 5

Learning Word Embeddings using Lexical Dictionaries

This chapter is based on the following publication

Julien Tissier, Christophe Gravier, and Amaury Habrard. "Dict2vec: Learning word embeddings using lexical dictionaries". In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2017)*, pages 254–263, 2017.

5.1 Introduction

Learning word embeddings usually relies on the following distributional hypothesis – words appearing in similar contexts must have similar meanings, and thus close representations. As we have seen in Chapter 3, finding such representations for words has been a hot topic of research within the last couple of years in the Natural Language Processing community. Most of the methods presented in Chapter 3 learn in an unsupervised way: there are no labels or clues given to the models to indicate which properties of words or which relations between words the representations should encode. They use the information of cooccurrences of words in a large text corpus, usually Wikipedia, to assess that words should (or should not) be related, following the distributional hypothesis. However, these methods suffer from a classic drawback of unsupervised learning: the lack of supervision, in this case between a word and those cooccurring with it, can lead the model to learn incorrect representations. Indeed, it is likely that some terms cooccurring in the context of a word are not related to it and therefore should not be used to learn its representation. Moreover, the fact that two words do not appear together or more likely not often enough together in any contexts of the training corpus is not a guarantee that these words are not semantically related and that their word representations should not be close.

To tackle the issue of the lack of supervision when learning word embeddings from the cooccurrences of words in a large text corpus, recent approaches have proposed to weight the words used in a context sequence [Vaswani et al., 2017] or to use external sources of information like knowledge graphs in order to improve the embeddings [Faruqui et al., 2015a, Kiela et al., 2015]. Similarities between words derived from such resources are either used in the objective function during the learning phase or used in a retrofitting scheme, as described in Chapter 3. This is beneficial for word embeddings because they can capture more linguistic and semantic information. However, these approaches tend to specialize the embeddings towards the used resource and its associated similarity measures.

Moreover, the construction and maintenance of those knowledge graphs resources are a set of complex, time-consuming, and error-prone tasks.

The first contribution of this thesis answers the problem of improving the semantic information contained in word embeddings, which is often not properly captured by unsupervised methods, by using additional linguistic information from an external source. Instead of using knowledge graphs, the contribution relies on another linguistic resource to improve the semantic information incorporated into word embeddings: lexical dictionaries. Dictionary entries (the definitions of words) contain latent word similarity and relatedness information that can improve language representations. Such entries provide an additional context which describes the meaning and use for most words of a vocabulary. The idea of using dictionaries to learn word embeddings is also motivated by the simple observation that when someone faces an unknown word, one natural reflex is usually to look up its definition in a dictionary, to understand its meaning. These word definitions represent a nice way to get both synonyms and relatedness information without needing a lot of additional manual annotations. The first contribution of this thesis is the development of a complete framework with tools to extract word definitions from online lexical dictionaries and a new model named `dict2vec` which uses the information found in dictionaries to learn word embeddings that incorporate this additional semantic knowledge information.

This chapter describes the architecture of the `dict2vec` model. It adds new cooccurrence information based on the terms occurring in the definitions of a word. This additional information can be used as a supervision when learning word embeddings to improve the semantic information they capture. Two types of word relations can be distinguished from the cooccurrences of words in dictionary definitions: pairs of words that appear mutually in the definition of the other one (we denote them as *strong pairs*); and pairs of words where only one word appears in the definition of the other one (denoted as *weak pairs*). Each type of pair has its own weight as an hyperparameter in the `dict2vec` model during the learning step. Not only the pairs are useful at learning time to control which word vectors should be moved closer, but it also becomes possible to use it to control and prevent words to be moved apart whereas they are actually related. Furthermore, the process of extracting pairs of related words from online lexical dictionaries does not require a human annotator; it can be fully automated. The main results of `dict2vec` are:

- A statistically significant improvement around 12.5% on eleven common evaluation datasets for the word semantic similarity task against classic methods used to learn word embeddings when they are trained on the full Wikipedia corpus.
- The improvement is even larger for small training corpus (first 50 million tokens of Wikipedia) as the average scores increase by around 30%.
- `dict2vec` also exhibits significantly better scores than competitors on a word semantic similarity task when the embeddings have a small number of dimensions (less than 50) and the training corpus is small, which is interesting for reducing the size of representations as described in Chapter 4.
- Word embeddings learned by `dict2vec` are not over-specialized for the word semantic similarity task as their performances are on par with other baselines on extrinsic text classification tasks.

In this chapter, Section 5.2 describes how online lexical dictionaries are processed to extract pairs of related words. Section 5.3 details the architecture of the `dict2vec` model. Section 5.4 describes the experimental settings and the tasks used to evaluate the embeddings learned by `dict2vec` and Section 5.5 analyzes the performances of the learned vectors and compares `dict2vec` to other word embeddings learning models.

5.2 Extracting linguistic information from dictionaries

5.2.1 Definition of a definition

The definition of a given word is composed of words or sentences explaining its meaning. A dictionary is a set of tuples (word, definition) for several words. For example, one may find in a dictionary the following definition for the word "car":

> **car**: A road vehicle, typically with four wheels, powered by an internal combustion engine and able to carry a small number of people.[1]

The presence of words like "vehicle", "road" or "engine" in the definition of "car" illustrates the relevance of using word definitions to obtain semantically-related pairs of words and use them as a supervision when learning word embeddings.

5.2.2 From word definitions to pairs of related words

In a definition, all the words do not have the same semantic relevance. For instance, in the definition of the word "car", the word "vehicle" is more relevant than the words "internal" or "number". To capture this difference of relevance, one contribution of `dict2vec` is the introduction of the concept of strong and weak pairs. They are defined as follows:

Definition 5.1 (A strong pair)**.** If the word w_a and the word w_b are mutually included in the definition of the other one, then (w_a, w_b) forms a strong pair.

Definition 5.2 (A weak pair)**.** If the word w_a occurs in the definition of the word w_b but w_b does not occur in the definition of w_a (*i.e.* the inclusion is not mutual but only one-sided), then (w_a, w_b) forms a weak pair.

Pairs are symmetric, so if (w_a, w_b) forms a strong (resp. weak) pair, then (w_b, w_a) also forms a strong (resp. weak) pair. Let us illustrate this concept with an example. Here are below the definitions of three words, taken from the Oxford English Dictionary:

> **car**: a road vehicle, typically with four wheels, powered by an internal combustion engine and able to carry a small number of people.
>
> **vehicle**: a thing used for transporting people or goods, especially on land, such as a car, lorry or cart.
>
> **wheel**: a circular object that revolves on an axle and is fixed below a vehicle or other object to enable it to move easily over the ground.

- The word "vehicle" is in the definition of the word "car" and "car" is in the definition of "vehicle". Hence ("car", "vehicle") is a strong pair.
- The word "wheel" is in the definition of the word "car" but "car" is not in the definition of "wheel". Therefore ("car", "wheel") is a weak pair.

An attentive reader may have noticed that the weak pair contains the word "wheel" whereas it is written as "wheels" in the definition of the word "car". A pre-processing step is applied to the dictionary definitions before generating the pairs to handle these cases, as they obviously refer to the same word.

Generating strong and weak pairs only from the mutual inclusions in definitions is restrictive because there is no guarantee that if two words are related, they use themselves to describe each other (*e.g.* it is unlikely that the word "car" is in the definition of the word

[1] Definition from the Oxford English Dictionary.

"engine", however the two words are strongly related). To overcome this shortcoming, `dict2vec` artificially generates strong pairs by considering that if the words w_a and w_b form a strong pair, then the neighbors of w_a also form a strong pair with w_b (conversely the neighbors of w_b form a strong pair with w_a). The neighbors of a word w are found by computing the cosine similarity between the vector of w and the vectors of all the words in a vocabulary, where vectors are taken from pre-trained word embeddings. Some pairs of words that are defined as weak pairs can therefore be promoted as strong pairs by considering the neighborhood of words. For instance, in the example, ("car', "wheel") is a weak pair but if "wheel" is a neighbor of "vehicle", then the pair ("car", "wheel") is promoted as a strong pair because ("car", "vehicle") is a strong pair.

5.3 Using strong and weak pairs to learn word embeddings

The `dict2vec` model is based on the `word2vec` Skip-gram model described in Subsection 3.2.2 of Chapter 3. As a reminder, the Skip-gram model aims to predict with a high probability the words of a context window $c_t = (w_{t-n}, \ \ldots, \ w_{t-1}, \ w_t, \ w_{t+1}, \ \ldots, \ w_{t+n})$ given the central word w_t, for all the context windows occurring in a text corpus composed of T tokens. In Skip-gram, the objective function to maximize is:

$$\begin{aligned} &\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-n\leq j\leq n\\ j\neq 0}} \log(P(w_{t+j} \mid w_t)) \\ =&\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-n\leq j\leq n\\ j\neq 0}} \Big(\underbrace{\log\left(\sigma({\mathbf{v'}_{t+j}}^{\top}\ \mathbf{v}_t)\right)}_{\text{(i)}} + \underbrace{\sum_{i=1}^{k}\mathbb{E}_{w_i\sim P_k(w_t)}\log\left(\sigma(-{\mathbf{v'}_i}^{\top}\ \mathbf{v}_t)\right)}_{\text{(ii)}} \Big) \end{aligned} \tag{5.1}$$

where $\mathbf{v}_t$ is the vector associated to the word w_t and σ is the sigmoid function. Two parts can be identified in this objective function: (i) to bring closer the vector representation of one word w_{t+j} from the context and the central word w_t of the window; (ii) to move apart the vector representation of a random word w_i taken from a noise distribution $P_k(w_t)$ and the central word w_t of the window. `dict2vec` improves this model by adding two new elements: a positive sampling and a controlled negative sampling.

5.3.1 Positive sampling

The concept of positive sampling is based on strong and weak pairs. In addition to moving closer the vectors of words cooccurring within the same context window as in `word2vec`, vectors of words forming either a strong or a weak pair are also moved closer in the `dict2vec` model. This new learning mechanism introduced by `dict2vec` is called *positive sampling* in echo to the *negative sampling* of Mikolov et al. [Mikolov et al., 2013a] which has the goal of moving apart vectors of unrelated words. In `dict2vec`, related words are moved closer hence the name of a *positive* sampling. Let $\mathcal{S}(w)$ be the set of all words forming a strong pair with the word w and $\mathcal{W}(w)$ be the set of all words forming a weak pair with w. For each context word w_c from the corpus, two random sets are built:

- $\mathcal{V}_s(w_c)$ a random set of n_s words drawn with replacement from $\mathcal{S}(w_c)$;
- $\mathcal{V}_w(w_c)$ a random set of n_w words drawn with replacement from $\mathcal{W}(w_c)$.

The loss function J_{pos} of the positive sampling for each context word is defined as:

$$J_{pos}(w_c) = \beta_s \sum_{w_i \in \mathcal{V}_s(w_c)} \log(1 + e^{-\mathbf{v'}_i{}^\top \mathbf{v}_c}) + \beta_w \sum_{w_j \in \mathcal{V}_w(w_c)} \log(1 + e^{-\mathbf{v'}_j{}^\top \mathbf{v}_c}) \tag{5.2}$$

where $\mathbf{v}_c$ (resp. $\mathbf{v}_i$ and $\mathbf{v}_j$) is the vector associated to the word w_c (resp. w_i and w_j). This loss function uses terms like $\log(1+e^{-x})$ whereas the objective function of Skip-gram (see Equation 5.1) uses terms like $\log(\sigma(x))$. The two terms are almost equivalent because $\log(\sigma(x)) = \log(\frac{1}{1+e^{-x}}) = -\log(1+e^{-x})$. The reason of this difference is explained shortly after in Subsection 5.3.3. The idea this loss function is to move closer words forming a strong or a weak pair for a given word w_c. The notion of moving closer two words is obtained by increasing the value of the dot product of their respective vectors. Let us suppose that two words form a strong pair. If the dot product of their vectors is high, then $e^{-\mathbf{u}^\top \mathbf{v}} \to 0$ and therefore $\log(1 + e^{-\mathbf{u}^\top \mathbf{v}}) \to 0$. On the other hand, if the dot product is low, $e^{-\mathbf{u}^\top \mathbf{v}} \to 1$ and therefore $\log(1+e^{-\mathbf{u}^\top \mathbf{v}}) \to \log(2)$. The objective of positive sampling is to minimize this loss for all context words w_c of the corpus, thus moving closer words forming a strong or a weak pair. The coefficients β_s and β_w as well as the number of drawn pairs n_s and n_w tune the importance of strong and weak pairs during the learning phase. The influence of these hyperparameters is discussed in Subsection 5.5.5. As a side remark, when $\beta_s = 0$ and $\beta_w = 0$, `dict2vec` is equivalent to the `word2vec` Skip-gram model with negative sampling introduced by Mikolov et al. [Mikolov et al., 2013a].

5.3.2 Controlled negative sampling

Negative sampling, as introduced by Mikolov et al. [2013a], consists in moving away the vector representation of the central word of a context window from the vectors of some randomly selected words from the vocabulary. Indeed, if words are randomly selected and given the large size of the vocabulary, which can be composed of hundreds of thousands of words, it is likely that they are not related so their word representations should be dissimilar. In negative sampling, for each word w of the vocabulary $\mathcal{V}$, a set $\mathcal{F}(w)$ of k randomly selected words from the vocabulary $\mathcal{V}$ (except the word w) is generated:

$$\mathcal{F}(w) = \{w_i\}^k,\ w_i \in \mathcal{V} \setminus \{w\} \tag{5.3}$$

The model is then trained to move apart the vector of the word w and the vectors of words from $\mathcal{F}(w)$. More formally, it is equivalent to minimize the following loss J_{neg} for each central word w_t of a context window:

$$J_{neg}(w_t) = \sum_{w_i \in \mathcal{F}(w_t)} \log(1 + e^{\mathbf{v'}_i{}^\top \mathbf{v}_t}) \tag{5.4}$$

where $\mathbf{v}_t$ (resp. $\mathbf{v}_i$) is the vector associated to the word w_t (resp. w_i). However, the hypothesis that randomly selected words are not related to the central word w_t is not true and there is a non-zero probability that w_i and w_t are in fact related. Therefore, the negative sampling of the `word2vec` model will move the vectors further instead of moving them closer. In `dict2vec`, with the supervision of strong and weak pairs, it becomes possible to better ensure that this is less likely to occur: a word which forms either a strong or a weak pair with w_t is prevented to be used as a negative example. The negative sampling loss from Equation 5.4 becomes:

$$J_{neg}(w_t) = \sum_{\substack{w_i \in \mathcal{F}(w_t) \\ w_i \notin \mathcal{S}(w_t) \\ w_i \notin \mathcal{W}(w_t)}} \log(1 + e^{\mathbf{v'}_i{}^\top \mathbf{v}_t}) \tag{5.5}$$

In the experiments, the controlled negative sampling method discards around 2% of originally generated negative pairs. The influence on the quality of vectors during the evaluation step depends on the nature of the corpus and is discussed in Subsection 5.5.5.

5.3.3 Global objective function

The objective function of `dict2vec` is derived from the objective function of the `word2vec` Skip-gram model with negative sampling, reminded in Equation 5.1. The positive sampling is added to this objective function and the original negative sampling is replaced by the controlled negative sampling, which are both respectively described in Subsection 5.3.1 and in Subsection 5.3.2. The loss function J for each ($target$, $context$) pair of words extracted from context windows of a text corpus is defined as:

$$J(w_t,\ w_c) = \log(1 + e^{-{\mathbf{v'}_c}^\top \mathbf{v}_t}) + J_{pos}(w_c) + J_{neg}(w_t) \tag{5.6}$$

The global objective function J_{global} minimized by `dict2vec` is obtained by summing the loss J of every ($target$, $context$) pair over the entire corpus composed of T tokens:

$$\begin{aligned} J_{global} &= \sum_{t=1}^{T} \sum_{\substack{-n \leq j \leq n \\ j \neq 0}} J(w_t,\ w_{t+c}) \\ &= \sum_{t=1}^{T} \sum_{\substack{-n \leq j \leq n \\ j \neq 0}} \Big(\log\left(1 + e^{-{\mathbf{v'}_c}^\top \mathbf{v}_t}\right) \\ &\qquad + \beta_s \sum_{w_i \in \mathcal{V}_s(w_c)} \log(1 + e^{-{\mathbf{v'}_i}^\top \mathbf{v}_c}) + \beta_w \sum_{w_j \in \mathcal{V}_w(w_c)} \log(1 + e^{-{\mathbf{v'}_j}^\top \mathbf{v}_c}) \\ &\qquad + \sum_{\substack{w_i \in \mathcal{F}(w_t) \\ w_i \notin \mathcal{S}(w_t) \\ w_i \notin \mathcal{W}(w_t)}} \log(1 + e^{{\mathbf{v'}_i}^\top \mathbf{v}_t}) \Big) \end{aligned} \tag{5.7}$$

The objective function of the `word2vec` Skip-gram model (see Equation 5.1) has to be maximized while the objective function of the `dict2vec` model (see Equation 5.7) has to be minimized. This seems counter-intuitive but it can be explained easily. In the objective function of Skip-gram, the terms use the logarithm of the sigmoid function σ, defined as:

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

and therefore:

$$\log\left(\sigma(\mathbf{u}^\top \mathbf{v})\right) = \log\left(\frac{1}{1 + e^{-\mathbf{u}^\top \mathbf{v}}}\right) = -\log\left(1 + e^{-\mathbf{u}^\top \mathbf{v}}\right)$$

Therefore, maximizing $\log\left(\sigma(\mathbf{u}^\top \mathbf{v})\right)$ is equivalent to minimizing $\log\left(1 + e^{-\mathbf{u}^\top \mathbf{v}}\right)$ (because of the minus sign) which is the term used in the `dict2vec` objective function.

5.4 Experimental settings and evaluation tasks

This section focuses on the experimental settings used to train the `dict2vec` model and the tasks to evaluate the learned word embeddings: Subsection 5.4.1 details the process used to obtain the definitions of words from lexical dictionaries, which are required to generate strong and weak pairs; Subsection 5.4.2 describes the different training text

corpora used to learn word embeddings and the values of the optimal hyperparameters of `dict2vec`; Subsection 5.4.3 presents the tasks used to evaluate the word embeddings learned by `dict2vec` as well as the competitor models used to compare them.

5.4.1 Fetching word definitions of online dictionaries

In order to build strong and weak pairs used during the training of the model, `dict2vec` needs the definition of all the words from the vocabulary of the training corpus. The vocabulary is created by extracting all the unique words with more than 5 occurrences from a full English Wikipedia dump. After the extraction step and the creation of the vocabulary, it contains around 2.2M words. Then, for each word of this vocabulary, its definition is extracted from an online lexical dictionary (`dict2vec` uses online dictionaries because their content is more up-to-date than the content of offline dictionaries). Since there are no online dictionaries which contain a definition for all the existing words (the word w might be in the dictionary D_i but not in D_j), several dictionaries are combined to get a definition for almost all of the words of the vocabulary (some words are too rare to have a definition). Four different English dictionaries are used: Cambridge, Oxford, Collins and dictionary.com. For each word, its webpage from the four online dictionaries is downloaded and its word definitions are extracted from the respective HTML source code of each webpage with the use of regular expressions. Each website has its own HTML template where the definition information is placed in specific location. With correct regular expressions, word definitions are accurately and automatically extracted. Below is an example of one of the webpage which contains definitions of the word "car".

Figure 5.1: Example of an HTML page from the Cambridge dictionary.

`dict2vec` does not focus on learning a vector for each sense of polysemous words so the definitions of all the senses of a word are concatenated and considered as a single definition. For each word, its definitions from all the dictionaries are also concatenated. Stop words and punctuation are removed, words are lowercased and a pre-processing step is applied to transform plural words to singular ones. For the word "car", the definition obtained is:

> **car**: road vehicle engine wheel seat small number people part train special purpose separate part train passenger sit part train carrying good animal road vehicle engine wheel seating people car part train abbreviation capital adequacy ratio automobile vehicle running rail streetcar railroad car part elevator balloon modern airship carries passenger freight british wheeled vehicle farm cart wagon literary chariot war triumph archaic cart carriage selfpropelled road vehicle designed carry passenger wheel powered engine modifier conveyance passenger freight cable car carrier airship balloon railway vehicle passenger sleeping car buffet car railway carriage van enclosed platform lift

Among the 2.2M unique words, only 200K have a definition. Strong and weak pairs are generated from the downloaded definitions according to the rules described in Subsection 5.2.2, leading to 417K strong pairs (when the number of close neighbors used to extend strong pairs is set to 5) and 3.9M weak pairs.

5.4.2 Training settings

Training corpora

`dict2vec` model is trained with the generated strong and weak pairs and the November 2016 English dump from Wikipedia. After removing all XML tags found in the Wikipedia dump and converting all words to lowercase (with the help of Mahoney's script[2]), the corpus is separated into 3 training files containing respectively the first 50M tokens, the first 200M tokens, and the full dump. `dict2vec` uses additional knowledge during training so it would not be fair to compare `dict2vec` to other common models which only train word embeddings with a text corpus and no additional knowledge. To overcome this shortcoming and have a fair comparison against other models, the additional information extracted from definitions is also incorporated into the text corpora and two versions of each text training file are created: one containing only text from Wikipedia (noted as "Wikipedia" in Section 5.5) and another one with text from Wikipedia concatenated to the extracted definitions (noted as "Wikipedia + definitions" in Section 5.5).

Hyperparameters of `dict2vec`

Hyperparameters for the general settings of `dict2vec` are set to the ones usually found in the literature, that is: 5 negatives samples, 5 training epochs, a context window composed of 5 words, a vector size of 100 dimensions (resp. 200 and 300) for the 50M tokens training file (resp. the 200M tokens and the full dump training files) and words with less than 5 occurrences are removed from the corpus. For the specific hyperparameters of `dict2vec`, the same tuning protocol as in the `word2vec` and `fasttext` papers is followed to provide the fairest comparison against other competitors, so every other hyperparameters (β_s, β_w, n_s, n_w) are tuned with a grid search to maximize the weighted average score on a word semantic similarity task. For n_s and n_w, values from 0 to 10 with a step of 1 have been tested. For β_s and β_w, values from 0 to 2 with a step of 0.05 have been tested. The optimal hyperparameters for the `dict2vec` model are: $n_s = 4$, $n_w = 5$, $\beta_s = 0.8$ and $\beta_w = 0.45$.

5.4.3 Evaluation protocol

Evaluation tasks

- **Word semantic similarity evaluation.** `dict2vec` follows the standard method for this evaluation task by computing the Spearman's rank correlation coefficient [Spearman, 1904] between human similarity evaluation of pairs of words and the cosine similarity of the corresponding word vectors. This task has been explained in more details in Section 1.4.2 of Chapter 1. The following classic datasets are used:
  - MC-30 [Miller and Charles, 1991]
  - MEN [Bruni et al., 2014]
  - MTurk-287 [Radinsky et al., 2011]
  - MTurk-771 [Halawi et al., 2012]
  - RG-65 [Rubenstein and Goodenough, 1965]
  - RW [Luong et al., 2013]
  - SimVerb-3500 [Gerz et al., 2016]
  - WordSim-353 [Finkelstein et al., 2001]
  - YP-130 [Yang and Powers, 2006]

[2] `http://mattmahoney.net/dc/textdata.html#appendixa`

The same protocol as the one used by `word2vec` and `fasttext` is followed: pairs containing a word which does not occur in the training text corpus are discarded. Since `dict2vec` and the competitor models are all trained on the same corpora, they all have the same out-of-vocabulary (OOV) rate (the ratio of pairs not used to compute the Spearman's rank correlation coefficient). Each training is repeated 3 times and Table 5.2 reports the average score of the 3 training, to minimize the effect of the neural network random initialization. The weighted average score (*W.Average*) on all datasets is computed by weighting each dataset score by the number of pairs that are evaluated in it, in the same way as Iacobacci et al. [Iacobacci et al., 2016].

- **Text classification evaluation.** This task (described in more details in Section 1.4.3 in Chapter 1) follows the same setup as in the `fasttext` paper [Joulin et al., 2016b]. A neural network composed of a single hidden layer is trained, where the input layer corresponds to the Bag-of-Words of a document and the output layer is the probability to belong to each possible class. Weights between the input and the hidden layer are initialized with the learned word embeddings and are freezed during the neural network training so that the evaluation score solely depends on the word embeddings used for the initialization. The weights of the neural network are learned with the stochastic gradient descent technique to minimize the number of prediction errors. The datasets used for this classification task are: AG-News[3], DBpedia [Auer et al., 2007] and Yelp reviews[4] (polarity and full). Each dataset is split into a training and a test file. The same training and test files are used for all models. The classification accuracy obtained on the test file is reported in Table 5.3.

Baselines

Two other models are used to compare the performances of `dict2vec` against competitors: `word2vec`[5] and `fasttext`[6]. Both competitors have been trained on the same 3 corpora as `dict2vec` (50M tokens, 200M tokens and full Wikipedia dump) and their 2 respective versions ("Wikipedia" and "Wikipedia + definitions") described in Section 5.4.2. They use the same general hyperparameters as `dict2vec`, which are detailed in Section 5.4.2 and commonly used in the literature. `word2vec` is trained with its Skip-gram architecture because the `dict2vec` method is based on the Skip-gram model. `GloVe` has also been trained on the same corpora with its respective hyperparameters described in [Pennington et al., 2014] but their results are lower than all the other baselines (weighted averages on the word semantic similarity task are 0.350 for the 50M file, 0.389 for the 200M file and 0.454 for the full dump) so the results of `GloVe` are not reported in Table 5.2.

To compare `dict2vec` against another method which uses additional information, word embeddings learned on the "Wikipedia" corpus are retrofitted with the method of Faruqui et al. [2015a]. `retrofitting` introduces external knowledge from the WordNet semantic lexicon [Miller, 1995]. Word embeddings learned with `dict2vec` and the other competitors are all retrofitted[7] with the WN_{all} semantic lexicon from WordNet and 10 iterations, as advised in the paper of Faruqui et al.. Another comparison is made by evaluating the performances of `dict2vec` using WordNet as an additional resource instead of dictionaries.

[3] `https://www.di.unipi.it/~gulli/AG_corpus_of_news_articles.html`
[4] `https://www.yelp.com/dataset`
[5] `https://github.com/tmikolov/word2vec`
[6] `https://github.com/facebookresearch/fastText`
[7] `https://github.com/mfaruqui/retrofitting`

5.5 Results and model analysis

This section presents the results obtained by the word embeddings learned with the `dict2vec` model on two different tasks: a word semantic similarity task (Subsection 5.5.1) and a text classification task (Subsection 5.5.2). In addition to comparing `dict2vec` to other common methods used to learn word vectors, a comparison to a method which improves the captured semantic information with additional external knowledge is presented in Subsection 5.5.3. Moreover, a comparison between the use of dictionaries and the use of WordNet in the `dict2vec` architecture is presented in Subsection 5.5.4. Finally, Subsection 5.5.5 analyzes the influence of the different hyperparameters of `dict2vec`.

5.5.1 Word semantic similarity

Table 5.2 reports the Spearman's rank correlation scores obtained on several datasets and following the protocol described in Subsection 5.4.3 for the word semantic similarity task. Each model (`word2vec`, `fasttext` and `dict2vec`) have been trained on 3 different corpora sizes: 50M tokens, 200M tokens and the full Wikipedia dump (which contains around 4.1B tokens). Since all the models are trained on the same corpora, the percentage of out-of-vocabulary (OOV) pairs (a pair of words in a similarity dataset where one of the word is not in the training corpus) is the same for all the models. The ratio of OOV pairs for each dataset is reported in Table 5.1. All the pairs have words within the training corpora except for the RW dataset, which is expected because this dataset contains many Rare Words (RW) which are only found in large corpora (like the full Wikipedia dump).

	50M	200M	full
MC-30	0%	0%	0%
MEN-TR-3k	0%	0%	0%
MTurk-287	0%	0%	0%
MTurk-771	0%	0%	0%
RG-65	0%	0%	0%
RW	36%	16%	2%
SimVerb	3%	0%	0%
WS353-ALL	0%	0%	0%
WS353-REL	0%	0%	0%
WS353-SIM	0%	0%	0%
YP-130	3%	0%	0%

Table 5.1: Percentage of pairs of words not used for evaluation in the semantic similarity task for different sizes of corpus files and datasets. When a pair contains a word not in the vocabulary of the corpus, it is discarded for the evaluation.

`dict2vec` vs. others. The `dict2vec` model outperforms other approaches on most of the datasets for the 50M and the 200M tokens files, and on all datasets (excepted one) for the full Wikipedia dump (this is significant according to a two-sided Wilcoxon signed-rank test with $\alpha = 0.05$). On the weighted average score, the `dict2vec` model improves `fasttext` performances on the "Wikipedia" corpus by 20.8% for the 50M tokens file, by 13.1% for the 200M tokens file and by 12.8% for the full dump. Even when `fasttext` has access to additional knowledge during training (scores in the column "Wikipedia `+` definitions"), the `dict2vec` model still has better performances with an increase of 2.9% for the 50M tokens file, 5.2% for the 200M tokens file and 11.3% for the full dump.

	Wikipedia			Wikipedia + definitions		
	`word2vec`	`fasttext`	`dict2vec`	`word2vec`	`fasttext`	`dict2vec`
MC-30	0.697	0.722	0.840	0.847	0.823	**0.859**
MEN-TR-3k	0.692	0.697	0.733	0.753	**0.767**	0.762
MTurk-287	0.657	0.657	0.665	**0.688**	0.685	0.682
MTurk-771	0.596	0.597	0.685	0.677	0.692	**0.713**
RG-65	0.714	0.671	0.824	0.865	0.842	**0.875**
RW	0.375	0.442	0.475	0.420	**0.512**	0.489
SimVerb	0.165	0.179	0.363	0.371	0.374	**0.432**
WS353-ALL	0.660	0.657	0.738	0.739	0.739	**0.753**
WS353-REL	0.619	0.623	0.679	**0.700**	0.696	0.688
WS353-SIM	0.714	0.714	0.774	**0.797**	0.790	0.784
YP-130	0.458	0.415	0.666	0.679	0.674	**0.696**
W.Average	0.453	0.467	0.564	0.562	0.582	**0.599**

(a) Trained with a Wikipedia corpus file of 50M tokens.

	Wikipedia			Wikipedia + definitions		
	`word2vec`	`fasttext`	`dict2vec`	`word2vec`	`fasttext`	`dict2vec`
MC-30	0.742	0.795	**0.854**	0.830	0.814	0.827
MEN-TR-3k	0.734	0.754	0.752	0.758	**0.772**	0.768
MTurk-287	0.642	0.671	0.667	**0.671**	0.661	0.666
MTurk-771	0.628	0.632	0.682	0.669	0.675	**0.704**
RG-65	0.771	0.755	0.857	0.842	0.829	**0.877**
RW	0.377	0.475	0.467	0.408	**0.507**	0.478
SimVerb	0.183	0.206	0.377	0.306	0.329	**0.424**
WS353-ALL	0.694	0.701	**0.762**	0.734	0.735	0.758
WS353-REL	0.665	0.644	**0.710**	0.706	0.685	0.699
WS353-SIM	0.743	0.758	0.784	**0.792**	0.792	0.787
YP-130	0.449	0.509	0.616	0.592	0.639	**0.665**
W.Average	0.471	0.503	0.569	0.533	0.563	**0.592**

(b) Trained with a Wikipedia corpus file of 200M tokens.

	Wikipedia			Wikipedia + definitions		
	`word2vec`	`fasttext`	`dict2vec`	`word2vec`	`fasttext`	`dict2vec`
MC-30	0.809	0.831	**0.860**	0.826	0.815	0.847
MEN-TR-3k	0.733	0.752	**0.756**	0.728	0.751	0.755
MTurk-287	0.660	**0.672**	0.661	0.656	0.671	0.660
MTurk-771	0.623	0.631	**0.696**	0.620	0.638	0.694
RG-65	0.787	0.817	**0.875**	0.802	0.820	0.867
RW	0.407	0.464	**0.482**	0.427	0.468	0.476
SimVerb	0.186	0.222	**0.384**	0.214	0.233	0.379
WS353-ALL	0.705	0.729	0.756	0.721	0.723	**0.758**
WS353-REL	0.664	0.687	0.702	0.681	0.686	**0.703**
WS353-SIM	0.757	0.775	0.781	0.767	0.779	**0.781**
YP-130	0.502	0.533	**0.646**	0.475	0.553	0.607
W.Average	0.476	0.508	**0.573**	0.488	0.512	0.570

(c) Trained with the full Wikipedia corpus file.

Table 5.2: Semantic similarity scores on several datasets and different sizes of corpora for `word2vec`, `fasttext` and `dict2vec`. Each model is trained and evaluated 3 times; the mean score of the 3 runs is reported in the table. *W.average* is the weighted average for all word similarity datasets.

Models based on external knowledge sources presented in Section 3.3 of Chapter 3 do not provide word semantic similarity scores for all the datasets used in this evaluation. However, for the reported scores, `dict2vec` outperforms these models: Kiela et al. [Kiela et al., 2015] reach a score of 0.72 on the MEN dataset (0.756 for `dict2vec`); Xu et al. [Xu et al., 2014] obtain 0.683 on the WS353-ALL dataset (0.758 for `dict2vec`).

Adding definitions to the corpus. The column "Wikipedia + definitions" in Table 5.2, which represents the case where training corpora are composed of Wikipedia and word definitions concatenated to them, has better results than the column "Wikipedia" which represents training corpora only composed of Wikipedia: +18.3% on average for the 50M tokens file, +9.7% for the 200M tokens file. This demonstrates the strong semantic relations one can find in definitions and that simply incorporating definitions into small training corpora can boost the performances of the word embeddings trained on them. This is significant because the training time is largely decreased when the corpus file is smaller (training with the 50M tokens corpus file is 150 times faster than training on the full Wikipedia corpus) so adding definitions is a better option than extending the corpus with content from Wikipedia. However, when the training file is large (full dump), the supervision brought by strong and weak pairs is more efficient to improve the word semantic similarity scores than adding word definitions to the corpus as the boost brought by the concatenation of definitions is not significant for the full dump (+0.9% on average). It is also important to observe that, although the `dict2vec` model is superior for each corpus size, its word embeddings trained on the "Wikipedia" corpus of 50M tokens outperforms the word embeddings of other models trained on the "Wikipedia" full dump (an improvement of 11% compared to the results of `fasttext` vectors, its best competitor, trained on the full dump). This means that considering strong and weak pairs in a model is more efficient than increasing the training corpus size and that using dictionaries is a good way to improve the quality of the embeddings when the training file is small.

5.5.2 Text classification accuracy

Table 5.3 reports the classification accuracies (percentage of correctly classified documents) obtained for `word2vec`, `fasttext` and `dict2vec` on the datasets listed in Subsection 5.4.3. The `dict2vec` model achieves similar level of performances as `word2vec` and `fasttext` when trained on the 50M tokens file and slightly better results when trained on the 200M tokens file and on the full dump. Using strong and weak pairs as a supervision during training does not make the `dict2vec` model specialized for the word semantic similarity task, which demonstrates that the learned embeddings can also be used in downstream models for extrinsic tasks. In this classification evaluation, values of the embeddings are not updated during the training phase (only the parameters of the classifier are learned) since the objective is to evaluate the capacity of the embeddings to be used in another task rather than obtaining the best possible classification model. It is however possible to obtain better results than those presented in Table 5.3 by updating the embeddings values during training with the supervised information of labeled documents in order to adapt the embeddings for this classification task, as done by Joulin et al. [Joulin et al., 2016b].

5.5.3 `dict2vec` vs. `retrofitting`

`retrofitting` [Faruqui et al., 2015a] is a method to improve the semantic information encoded into word embeddings **after** they have been trained on a corpus (this method is described in more details in Section 3.3 of Chapter 3). Faruqui et al. method has been applied on each of the word embeddings trained on "Wikipedia" corpora with pairs of related words extracted from WordNet to produce retrofitted vectors of the `word2vec`,

	Wikipedia			Wikipedia + definitions		
	word2vec	fasttext	dict2vec	word2vec	fasttext	dict2vec
AG-News	**87.4**	86.8	87.1	87.1	87.1	86.6
DBPedia	93.5	94.2	**94.4**	94.2	94.4	94.4
Yelp Pol.	80.8	82.1	83.2	83.5	**84.2**	83.4
Yelp Full	45.1	46.0	47.1	46.9	**47.3**	47.2

(a) Trained with a Wikipedia corpus file of 50M tokens.

	Wikipedia			Wikipedia + definitions		
	word2vec	fasttext	dict2vec	word2vec	fasttext	dict2vec
AG-News	**88.6**	88.0	88.0	88.2	88.1	88.0
DBPedia	95.2	95.7	**96.0**	95.6	95.8	95.9
Yelp Pol.	83.7	85.2	85.6	85.5	85.9	**85.9**
Yelp Full	47.7	48.8	49.9	49.1	49.5	**50.1**

(b) Trained with a Wikipedia corpus file of 200M tokens.

	Wikipedia			Wikipedia + definitions		
	word2vec	fasttext	dict2vec	word2vec	fasttext	dict2vec
AG-News	88.5	**88.7**	88.1	88.5	88.7	88.4
DBPedia	96.6	96.7	96.8	96.6	96.7	**96.9**
Yelp Pol.	86.5	87.2	**87.6**	86.7	87.4	87.5
Yelp Full	50.6	51.2	51.6	50.6	51.4	**51.8**

(c) Trained with the full Wikipedia corpus file.

Table 5.3: Accuracies on a text classification task (percentage of correctly classified documents). Each model is trained and evaluated 3 times; the mean score of the 3 runs is reported.

`fasttext` and `dict2vec` models. Retrofitted vectors have been evaluated on the word semantic similarity task on all the datatsets and scores of retrofitted vectors are compared to either: (i) the score of non-retrofitted vectors for the same model and the same corpus size (vs. self (not **rf**)); (ii) the score of non-retrofitted vectors of `dict2vec` for the same corpus size (vs. `d2v` (not **rf**)). The percentages of score differences are reported in Table 5.4.

The `retrofitting` method improves the word semantic similarity scores for all models on almost all datasets because the percentages differences are almost always positive when retrofitted vectors are compared to their non-retrofitted version ("vs. self (not **rf**)" in Table 5.4). However, for the datasets RW, WS353-ALL and WS353-REL, the percentages are negative, which indicates that `retrofitting` deteriorates the semantic linguistic information encoded into the word embeddings which is needed to evaluate that pairs from those datasets are related. While `retrofitting` can improve the quality of information encoded into word embeddings and thus increase the scores on the semantic similarity task, it cannot reach the performance increase that `dict2vec` brings. Indeed, even when `word2vec` and `fasttext` vectors are retrofitted (columns $\texttt{w2v}_{\textbf{rf}}$ and $\texttt{ft}_{\textbf{rf}}$ in Table 5.4), their scores are still lower than the scores of non-retrofitted `dict2vec` vectors (every percentages in the "vs. `d2v` (not **rf**)" rows are negative). It can also be noticed that `dict2vec` model is compatible with a retrofitting improvement method as its scores are also increased with the Faruqui et al. method. `dict2vec` uses additional information from dictionaries while `retrofitting` uses information from WordNet. This demonstrates that the two external sources of kwowledge are not overlapping nor exclusive and that a model using both have better performances than a model which uses only one of the two.

		50M			200M			full		
		w2v$_{rf}$	ft$_{rf}$	d2v$_{rf}$	w2v$_{rf}$	ft$_{rf}$	d2v$_{rf}$	w2v$_{rf}$	ft$_{rf}$	d2v$_{rf}$
MC-30	vs. self (not rf)	+13.9%	+9.2%	+1.3%	+5.8%	+4.8%	+3.0%	+5.2%	+2.9%	+1.2%
	vs. d2v (not rf)	**-7.3%**	**-4.4%**	–	**-3.6%**	**-2.4%**	–	**-1.0%**	**-0.6%**	–
MEN-TR-3k	vs. self (not rf)	+0.9%	-0.7%	-0.1%	+0.7%	-1.9%	+0.4%	+1.4%	-2.8%	+1.6%
	vs. d2v (not rf)	**-4.2%**	**-7.4%**	–	**-1.3%**	**-1.6%**	–	**-1.7%**	**-3.3%**	–
MTurk-287	vs. self (not rf)	+1.4%	+0.2%	+3.5%	-2.9%	-3.3%	+3.0%	-0.9%	-5.7%	+1.1%
	vs. d2v (not rf)	**-1.2%**	**-4.0%**	–	**-4.3%**	**-2.7%**	–	**-1.1%**	**-4.1%**	–
MTurk-771	vs. self (not rf)	+8.2%	+4.9%	+1.6%	+6.3%	+4.3%	+1.5%	+4.5%	+1.6%	+0.6%
	vs. d2v (not rf)	**-7.3%**	**-8.8%**	–	**-3.8%**	**-3.4%**	–	**-6.5%**	**-7.9%**	–
RG-65	vs. self (not rf)	+10.9%	+17.1%	+4.0%	+6.6%	+8.5%	+3.0%	+7.0%	+5.0%	+2.4%
	vs. d2v (not rf)	**-2.1%**	**-4.9%**	–	**-2.2%**	**-4.4%**	–	**-3.8%**	**-1.9%**	–
RW	vs. self (not rf)	-20.3%	-24.4%	-14.3%	-24.4%	-25.9%	-20.3%	-18.7%	-25.4%	-19.1%
	vs. d2v (not rf)	**-37.4%**	**-26.9%**	–	**-37.7%**	**-24.6%**	–	**-31.3%**	**-28.2%**	–
Simverb	vs. self (not rf)	+46.0%	+34.0%	+19.8%	+49.7%	+39.8%	+19.9%	+44.6%	+38.7%	+16.7%
	vs. d2v (not rf)	**-34.4%**	**-28.4%**	–	**-30.5%**	**-23.6%**	–	**-29.9%**	**-19.8%**	–
WS353-ALL	vs. self (not rf)	-4.2%	-10.8%	-1.1%	-3.2%	-8.0%	-1.3%	-4.4%	-10.7%	-2.0%
	vs. d2v (not rf)	**-13.8%**	**-19.2%**	–	**-11.9%**	**-15.4%**	–	**-10.8%**	**-13.9%**	–
WS353-REL	vs. self (not rf)	-16.1%	-22.7%	-4.9%	-10.4%	-17.9%	-4.5%	-10.7%	-19.7%	-6.0%
	vs. d2v (not rf)	**-20.9%**	**-27.2%**	–	**-17.7%**	**-25.5%**	–	**-15.5%**	**-21.4%**	–
WS353-SIM	vs. self (not rf)	+4.3%	+0.8%	+3.2%	+2.6%	+0.0%	+3.2%	+0.0%	-3.6%	+2.4%
	vs. d2v (not rf)	**-3.9%**	**-6.7%**	–	**-2.9%**	**-3.3%**	–	**-3.0%**	**-4.4%**	–
YP-130	vs. self (not rf)	+17.8%	+3.2%	+3.3%	+13.0%	+6.9%	+8.0%	+11.1%	+8.3%	+5.0%
	vs. d2v (not rf)	**-23.6%**	**-28.2%**	–	**-16.6%**	**-11.7%**	–	**-13.6%**	**-10.7%**	–

Table 5.4: (**rf** = retrofitted). Percentage differences of word semantic similarity scores between retrofitted vectors (X$_{rf}$) and non-retrofitted vectors (not **rf**). Each model is compared to their own non-retrofitted version (vs. self) and `dict2vec` non-retrofitted version (vs. `d2v`). A positive (resp. negative) percentage indicates that `retrofitting` improves (resp. deteriorates) the semantic score compared to non-retrofitted vectors. As an illustration: the top left value +13.9% means that `word2vec` retrofitted vectors improves the initial `word2vec` vectors performance by 13.9%, while the value −7.3% right below it means that their performance is 7.3% lower than the non-retrofitted vectors of `dict2vec`.

5.5.4 Dictionaries vs. WordNet

Table 5.5 reports the weighted average of semantic similarity scores of vectors obtained after training (not retrofitted) and vectors retrofitted with either pairs from WordNet or pairs generated from dictionaries. Retrofitting with dictionaries pairs outperforms retrofitting with WordNet pairs, meaning that information from dictionaries is better suited to improve word embeddings towards semantic similarities when retrofitting.

		not retrofitted	retrofitted$_{\text{WordNet}}$	retrofitted$_{\text{dictionaries}}$
50M	`word2vec`	0.453	0.474	**0.479**
	`fasttext`	0.467	0.476	**0.489**
	`dict2vec`	0.569	0.582	**0.582**
200M	`word2vec`	0.471	0.488	**0.494**
	`fasttext`	0.503	0.504	**0.524**
	`dict2vec`	0.569	0.581	**0.587**
full	`word2vec`	0.488	0.507	**0.512**
	`fasttext`	0.508	0.503	**0.529**
	`dict2vec`	0.571	0.583	**0.592**

Table 5.5: Weighted average of word semantic similarity scores for non-retrofitted vectors and after retrofitting vectors with WordNet pairs or with dictionary pairs.

To compare the linguistic information one can found in WordNet or in dictionaries, the `dict2vec` model has also been trained without using any strong nor weaks pairs during training or by using WordNet pairs as strong pairs. When no pairs are used, the `dict2vec` model is equivalent to the `word2vec` Skip-gram model. Weighted average of word semantic scores are reported in Table 5.6. Training with WordNet pairs used as strong pairs increases the scores compared to a model that does not use any pairs at all, showing that the supervision brought by the positive sampling of `dict2vec` is beneficial to the model. But it is not as good as when the model uses dictionary pairs, demonstrating once again that dictionaries contain more semantic information than WordNet.

	50M	200M	full
No pairs	0.453	0.471	0.488
With WordNet pairs	0.564	0.566	0.559
With dictionary pairs	**0.569**	**0.569**	**0.571**

Table 5.6: Weighted average semantic similarity scores of `dict2vec` models when trained without pairs and with WordNet or dictionary pairs.

5.5.5 Influence of `dict2vec` hyperparameters

Number of strong and weak pairs. The different experiments carried out when training the `dict2vec` model has shown one thing: the number of strong and weak pairs (resp. n_s and n_w) drawn for each word must be set accordingly to the size of the training file. For the 50M and the 200M tokens files, the `dict2vec` model is set with the hyperparameters $n_s = 4$ and $n_w = 5$. When training on the full dump (which is 20 times larger than the 200M tokens file because it contains about 4.1B tokens), the number of context windows in the corpus is largely increased, so is the number of (*target*, *context*) pairs and thus the number of times the positive sampling is used for the most frequent words. Therefore, the influence of strong and weak pairs needs to be adjusted and decreased because frequent words would be learned mainly from their strong and weak pairs not from the context windows in which they occur. When trained on the full dump, the number of strong and weak pairs used to learn word embeddings are set to $n_s = 2$ and $n_w = 3$.

Weight of strong and weak pairs. For the hyperparameters of the positive sampling, an empirical grid search shows that a $\frac{1}{2}$ ratio between β_s and β_w is a good rule-of-thumb for tuning these hyperparameters. It has also been noticed that when these coefficients are too low ($\beta_s \leq 0.5$ and $\beta_w \leq 0.2$), results get worse because the model does not take into account the information from the strong and weak pairs. On the other side, when they are too high ($\beta_s \geq 1.2$ and $\beta_w \geq 0.6$), the model discards too much information from the context in favor of the information from the pairs. This behavior is similar when the number of strong and weak pairs is too low or too high ($n_s, n_w \leq 2$ or $n_s, n_w \geq 5$).

Number of negative samples. The controlled negative sampling of `dict2vec` (see Subsection 5.3.2), which prevents the vectors of words forming a strong or a weak pair to be moved apart when words are randomly chosen during the negative sampling, increases the weighted average semantic similarity scores of learned word embeddings by 0.7% compared to the non-controlled version. Increasing the number of negative samples used during the training phase does not significantly improve the results except on the RW dataset where using 25 negative samples can boost the scores by 10%, as shown in Figure 5.2. Indeed, this dataset is mostly composed of rare words so word embeddings must learn to differentiate unrelated words rather than learning which vectors should be moved closer.

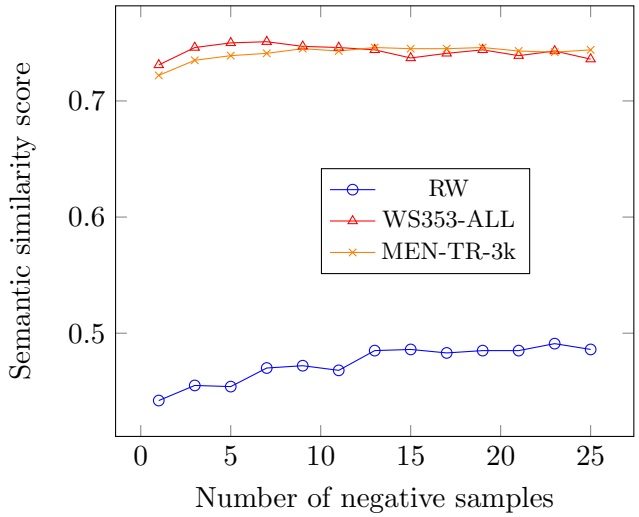


Figure 5.2: Semantic similarity scores on several datasets (RW, WS353-ALL and MEN-TR-3k) when the `dict2vec` model is trained with different numbers of negative samples.

Dimensions of word embeddings. Word embeddings with different numbers of dimensions have been trained with the `fasttext` and `dict2vec` models on the "Wikipedia" corpus of 50M tokens. Their semantic similarity scores on RW and WS353-ALL datasets are reported in Figure 5.3. It can be observed that `dict2vec` is still able to outperform the competitor approach when the number of dimensions of vectors is reduced to 20 or 40. Moreover, increasing the number of dimensions does increase the scores but only up to a certain point. When the number of dimensions is higher than 100, no significant improvements in semantic similarity scores are observed, which supports the fact that other common approaches choose vectors of dimension 100 when the training corpus is small.

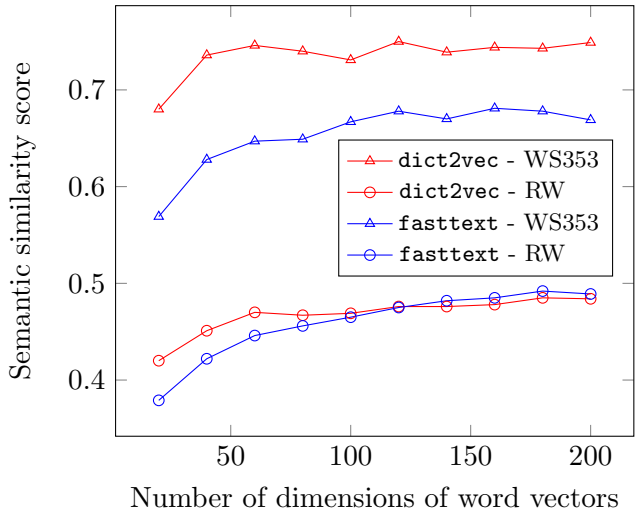


Figure 5.3: Semantic similarity scores on the RW and the WS353-ALL datasets for `fasttext` and `dict2vec` word embeddings trained with different numbers of dimensions on the "Wikipedia" corpus of 50M tokens.

Training time. The source code of the `dict2vec` model is based on the source code of the `word2vec` model, but the majority of the code has been rewritten and optimized for a faster training of word embeddings. The training times of different models on different

corpora sizes are reported in Table 5.7 (all experiments were run on a Intel E3-1246 v3 processor). Although `dict2vec` performs more operations during training than the `word2vec` Skip-gram model (because of the additional positive sampling and the controlled negative sampling), it is 4 times faster due to its code optimizations for word embeddings learned from the full Wikipedia dump, and almost 3 times as fast as `fasttext`.

	50M	200M	full
`word2vec`	15m30	86m	2600m
`fasttext`	8m44	66m	1870m
`dict2vec`	4m09	26m	642m

Table 5.7: Training time (in minutes) of the `word2vec`, `fasttext` and `dict2vec` models for several corpora sizes.

5.6 Conclusion

This chapter presented the first contribution of this thesis. As pointed out in Chapter 3, most of the existing methods to learn word embeddings only use generic text corpora like Wikipedia. The model presented here, named `dict2vec`, uses additional information from lexical dictionaries during training to encode more linguistic and semantic information into word embeddings. It generates pairs of related words (and in some cases, pairs of strongly related words) based on the cooccurrences of terms in dictionary definitions and uses this additional information in a novel positive sampling to extend the `word2vec` model.

Word embeddings learned with `dict2vec` have largely improved results on a word semantic similarity task and are slightly better on a text classification task compared to other common methods used to learn word embeddings like `word2vec` or `fasttext`. Moreover, `dict2vec` is better than other methods which use external information to learn word embeddings like `retrofitting` or than methods which use WordNet as an additional source of knowledge because dictionaries contain stronger semantic information about words than WordNet. Further experiments have also demonstrated that simply adding definitions of words to a small training corpus is better than extending this small corpus with Wikipedia content: small but specific corpora are superior than large and generic ones, as the information contained in dictionaries is more refined and less noisy.

One thing that has not been mentionned so far is the simplicity of the `dict2vec` model. Whether for creating strong and weak pairs or for learning word embeddings, all the computations and experiments of `dict2vec` have been able to run on the CPU of desktop computers. It was also possible to replicate the experiments from scratch on laptops, mainly due to the numerous code optimizations made from the original `word2vec` source code, leading to a faster and more efficient training. Source code to fetch dictionary definitions, generate related pairs of words, train the `dict2vec` model and evaluate the learned word embeddings on several datasets have been made publicly availabe[8].

Several works have extended the `dict2vec` model or the idea of using dictionaries to learn word embeddings. Bosc and Vincent [Bosc and Vincent, 2018] have proposed an architecture based on an autoencoder to learn a latent vector representation for each dictionary definition and reconstruct the definition from the latent vector. More recently, I participated in the development of an extension of the `dict2vec` model for the specific use case of anorexia detection. This system use pairs of words related to the semantic

[8] `https://github.com/tca19/dict2vec`

field of anorexia and update the word embeddings of these words to have more specific semantic information encoded into them. With the semantically-improved embeddings, a classifier system has better performances in a classification task which consists in detecting anorexia messages in social media posts. This work has been accepted and will appear in the proceedings of the IDA 2020 conference [Ramirez-Cifuentes et al., 2020]. Another possible way to extend the `dict2vec` model is to consider the polysemy information. Indeed, when definitions are extracted from online dictionaries, all the definitions of a word are concatenated together without considering that a word can have different senses and therefore different definitions. By separating the definitions of different senses, it would be possible to learn one embedding per sense like in [Iacobacci et al., 2015] to help downstream models to solve word sense disambiguation tasks [Iacobacci et al., 2016].

Finally, although the `dict2vec` model is simple, the learned word embedding matrix can have a size in memory in the order of gigabytes, which is a problem for using vectors on low-resource devices. This raises a new question: how can we learn word vectors with the benefits of the additional knowledge of dictionaries captured by `dict2vec` but with a smaller memory size? Chapter 4 presented methods to reduce the size of representations and explained how changing the representation space to use integer vector values also has the advantage of speeding up vector operations. But most of the presented methods were not targeted at word embeddings for use in downstream models. The next chapter presents the second contribution of this thesis, which addresses the problem of reducing the memory size of word embeddings and making vector operations faster, especially for the computation of semantic similarities between word vectors in downstream models.

Chapter 6

Binarization of Word Embeddings

This chapter is based on the following publication

Julien Tissier, Christophe Gravier, and Amaury Habrard. "Near-lossless binarization of word embeddings". In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2019)*, volume 33, pages 7104–7111, 2019.

6.1 Introduction

Word embeddings are usually computed for a large vocabulary size and with vectors of many dimensions. In the literature, it is common to see a vocabulary of millions of words and vectors of hundreds of dimensions. With such characteristics, storing word embeddings require several gigabytes of memory space. For example, with a vocabulary of 2 million words and 300-dimensional vectors, storing word embeddings require 2.4 gigabytes of memory space with real values encoded as `float`[1]. While learning and running models with word embeddings is not a problem for large computing servers, it becomes a concern to store word embeddings and run models which use them on embedded devices such as cell phones. Due to their limited memory and low computing power for floating-point arithmetic, it is hardly feasible in practice to do so. To overcome the problem of not being able to store word embeddings on memory-limited devices, NLP applications running on those devices usually send the data to process to large computing servers. The servers, which are able to store word embeddings and run the downstream model used for the task, handle the computations and return the results to the devices. This raises two concerns: (i) devices need an online connection to send data to computing servers; (ii) the privacy of device users is not preserved as their data are sent to computing servers and possibly stored server-side for other purposes. Reducing the size of word embeddings so downstream models could run on mobile devices would solve these two concerns as the computations would be done locally on the devices without needing to send data to a server, and therefore would allow one to run applications offline. Chapter 4 presented different methods to reduce the size of vector represensations like word embeddings.

Making word embeddings consuming less space in memory is not the only condition to be able to run models on low-resource devices. Indeed, such devices are not efficient for floating-point computations, which prevents fast operations between real-valued vectors. To overcome this problem, a solution is to use binary operations. One can either *binarize* the learning parameters of the model [Hubara et al., 2016] or *binarize* vector

[1] A `float` commonly requires 32 bits (= 4 bytes) in memory.

representations (see Section 4.3 in Chapter 4 for a review of such methods). The contribution described in this chapter stands in the second category. As explained in Section 4.3 of Chapter 4, vector operations are much faster with binary representations as they can be computed with bitwise operators instead of floating-point arithmetic. However, binary representations need to be aligned with CPU registers to fully take advantage of fast binary operations, which imposes the binary vectors to have a length of 64, 128 or 256 bits. Nevertheless, mapping words to binary codes is not enough as the vectors are then used in NLP applications. They also need to encode semantic and syntactic information; the objective then becomes to find binary vectors which preserve the linguistic properties and are small enough to fit in CPU registers. Note here that there is a difference between learning binary word embeddings from scratch (*i.e.* directly from text corpora or from external linguistic resources) and transforming pre-trained real-valued word embeddings. Most common models used to learn word embeddings propose to download pre-trained word vectors [Mikolov et al., 2013b, Pennington et al., 2014, Bojanowski et al., 2017, Tissier et al., 2017], each one with its own specific encoded linguistic information due to its learning scheme. Finding a method which can binarize pre-trained word embeddings is more useful than learning binary vectors from scratch because they would direcly encode the specific linguistic information of the original pre-trained word vectors instead of having to re-train the binary vectors with the appropriate learning scheme.

This chapter presents the second contribution of this thesis: a novel model to solve the problem of producing binary word vectors from pre-trained real-valued embeddings while preserving their linguistic and semantic information so they can be used in downstream models, requiring less memory space and speeding up vector operations. This model is based on an autoencoder architecture. The main results of this contribution are:

- The model architecture can transform any real-valued vectors to binary vectors of any size (*e.g.* 64, 128 or 256 bits to be in adequacy with CPU register sizes).
- The architecture has the ability to reconstruct original real-valued word vectors from the binary ones with high accuracy.
- Binary vectors **use 97% less memory space** than the original real-valued vectors with only a loss of $\sim$2% in accuracy on word semantic similarity, text classification and sentiment analysis tasks.
- A top-K query is **30 times faster** with binary vectors than with real-valued vectors.

This model is named `NLB` for "**N**ear-**L**ossless **B**inarization of word embeddings" in the rest of this chapter. In the following, Section 6.2 describes the autoencoder architecture used to *binarize* real-valued word embeddings into binary vectors while Section 6.3 presents the training settings and the evaluation protocol. Finally, Section 6.4 analyzes the performances of binary vectors on different downstream NLP task, as well as their semantic properties and their influence on the speed of vector operations. Entire source code to generate and evaluate binary vectors is publicly available online[2].

6.2 Binarizing word vectors with an autoencoder

As explained in the introduction of this chapter, transforming real-valued embeddings to binary vectors has the dual benefit of reducing the memory space of word embeddings and making vector operations faster. A naive approach to produce binary vectors is to map each value $\mathbf{x}_{[i]}$ of a pre-trained real-valued word vector $\mathbf{x}$ to 0 if $\mathbf{x}_{[i]} < 0$ and to 1 otherwise.

[2] `https://github.com/tca19/near-lossless-binarization`

This method does not require any training but suffers from an important drawback: binary vectors *have the same number of dimensions* as the original ones. Available pre-trained word embeddings typically have lengths of 100, 200 or 300 dimensions so to produce binary vectors of 64, 128 or 256 bits (to be aligned with CPU registers size), one has to train from scratch real-valued embeddings with those dimensions and then apply the naive binarization. Results in Subsection 6.4.4 show that it is slower and less efficient than the proposed method which directly transforms vectors with the appropriate binary size.

6.2.1 Autoencoder architecture

Let $\mathcal{V}$ be a vocabulary of words and $\mathbf{M} \in \mathbb{R}^{|\mathcal{V}| \times d}$ the embedding matrix whose rows are d-dimensional real-valued vectors representing the embedding of each word of $\mathcal{V}$. The main objective is to transform each word vector $\mathbf{x}_i$ (*i.e* each row of the matrix $\mathbf{M}$) into a binary vector $\mathbf{b}_i$ of n dimensions with $n \ll (d \times k)$ where k corresponds to the encoding size of real-valued numbers in memory (*e.g.* 32 bits for standard single-precision floating-point numbers) while preserving the semantic information contained in $\mathbf{x}_i$. The proposed method achieves this objective by using an autoencoder model composed of two parts: an encoder which binarizes a word vector $\mathbf{x}$ to $\Phi(\mathbf{x})$ and a decoder which reconstructs a real-valued vector $\hat{\mathbf{y}}$ from $\Phi(\mathbf{x})$. The binarized vector is therefore the latent representation $\Phi(\mathbf{x})$ of the autoencoder. Figure 6.1 summarizes this model architecture.

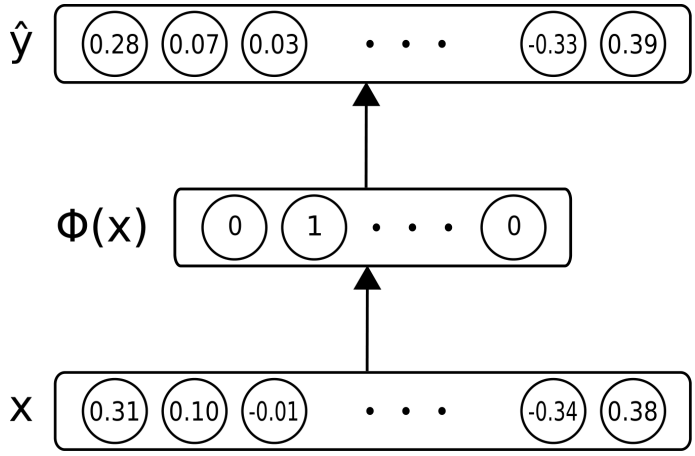


Figure 6.1: Autoencoder architecture used to binarize word vectors and reconstruct a real-valued representation. The model can learn $\Phi(\mathbf{x})$ vectors of any arbitrary size. To be aligned with CPU registers, the length of $\Phi(\mathbf{x})$ is suggested to be 64, 128 or 256 bits.

Encoding to binary vectors

Let $\mathbf{W} \in \mathbb{R}^{n \times d}$ be a $n \times d$ matrix and $\mathbf{x}_i \in \mathbb{R}^d$ a d-dimensional word embedding. The binarized vector $\mathbf{b}_i$ of $\mathbf{x}_i$ is defined as:

$$\mathbf{b}_i = \Phi(\mathbf{x}_i) = h(\mathbf{W} \cdot {\mathbf{x}_i}^\top) \tag{6.1}$$

where $h(.)$ is an element-wise function which outputs a bit given a real value, such as the Heaviside step function. The dimension of $\mathbf{W}$ is $(n \times d)$ *i.e.* the desired size n of binary vectors and the size d of the original real-valued vector $\mathbf{x}_i$. Therefore this model can be used to generate binary vectors of any size by choosing the appropriate value for n, independently of the size of the original real-valued vectors.

Reconstructing a real-valued vector

The latent representation $\Phi(\mathbf{x}_i)$ is then used to compute a reconstructed vector $\hat{\mathbf{y}}_i$ as:

$$\hat{\mathbf{y}}_i = f(\mathbf{W}^\top \cdot \Phi(\mathbf{x}_i) + \mathbf{c}) \tag{6.2}$$

where $\mathbf{c}$ is a d-dimensional real-valued bias vector and f is an element-wise non-linear function. The f function must be chosen such that its output domain is the same as the range of values in the input real-valued embedding $\mathbf{x}_i$ of the autoencoder. Indeed, if for example input embeddings all have negative values and f is a function whose output domain is $\mathbb{R}^+$, values of vectors $\hat{\mathbf{y}}_i$ would be positive. Since autoencoders are trained so their outputs are close to their inputs, it would not be possible to train it correctly in this case (inputs are negative, outputs would be positive).

The proposed model is trained to be able to reconstruct word embeddings. Most of the existing pre-trained word embeddings have values which are within the $[-1, 1]$ range. Following the imposed condition on f, the hyperbolic tangent function is used as f in the proposed model because its output domain is also the $[-1, 1]$ range. For the few word vectors having values outside this range, their values are clipped to be in $[-1, 1]$ before being used as inputs in the autoencoder model. Our experiments showed that the clipping step does not alter the quality of the pre-trained vectors (word semantic similarity scores stay the same for `GloVe` vectors) as most of the vector values are already within this range.

6.2.2 Objective function

The reconstruction loss ℓ_{rec} for a vector $\mathbf{x}_i$ is defined as the mean squared error between this original vector $\mathbf{x}_i$ and the reconstructed output vector $\hat{\mathbf{y}}_i$ of the autoencoder:

$$\ell_{rec}(\mathbf{x}_i) = \frac{1}{d} \left\| \mathbf{x}_i - \hat{\mathbf{y}}_i \right\|^2 = \frac{1}{d} \sum_{j=1}^{d} (\mathbf{x}_{i_{[j]}} - \hat{\mathbf{y}}_{i_{[j]}})^2 \tag{6.3}$$

where $\mathbf{x}_{i_{[j]}}$ (resp. $\hat{\mathbf{y}}_{i_{[j]}}$) is the j-th value of the vector $\mathbf{x}_i$ (resp. $\hat{\mathbf{y}}_i$). The autoencoder is trained to minimize this loss for all word vectors $\mathbf{x}_i$ in the embedding matrix $\mathbf{M}$. Learning the optimal parameters $\mathbf{W}$ and $\mathbf{c}$ of the autoencoder by minimizing this loss function produces good reconstructed vectors but the latent binary representations of words learned by the model are not preserving the semantic properties of original real-valued vectors. Indeed, since the model is trained to only minimize the reconstruction loss, the learned matrix $\mathbf{W}$ used in the encoding step discards too much similarity information between related word vectors from the original vector space in favor of the reconstruction (this behavior has been observed in practice in the experiments). To solve this problem, a regularization term has been added into the objective function of the model, defined as:

$$\ell_{reg} = \frac{1}{2} \left\| \mathbf{W}^\top \mathbf{W} - \mathbf{I} \right\|^2 \tag{6.4}$$

This term aims to minimize the correlation between the dimensions of latent binary vectors. Since the model aims to produce small-sized binary embeddings which preserve linguistic information, it needs to encode as much linguistic information as possible across all the dimensions of binary vectors. Therefore, minimizing the correlation between the dimensions of binary vectors is crucial to avoid duplicate information. This regularization has the effect of preserving the information contained in the real-valued embeddings, so vectors that are close in the original vector space are also close in the binary vector space. The balance between the influence of the reconstruction loss and the regularization loss is achieved with the λ_{reg} parameter. The global objective function $\mathcal{L}$ of the models is:

$$\mathcal{L} = \sum_{\mathbf{x}_i \in \mathbf{M}} \ell_{rec}(\mathbf{x}_i) + \lambda_{reg}\, \ell_{reg} \tag{6.5}$$

Note here that the same parameter matrix W is used both for the encoder and for the decoder part in the NLB autoencoder model. This is actually motivated by the fact that the function $h(\cdot)$ involved during the encoding step to tranform real-valued vectors with $\Phi(\cdot)$ is non-differentiable so it is not possible to compute a gradient to update the parameters of the encoder with the gradient descent technique. It is assumed that:

$$\frac{\partial \Phi(\mathbf{x}_i)}{\partial \mathbf{W}} = 0 \tag{6.6}$$

However, the values of the $\mathbf{W}$ matrix used during the encoding step can still be updated thanks to the information provided by the decoder and its gradient, which is:

$$\frac{\partial \left\| \mathbf{x}_i - \hat{\mathbf{y}}_i \right\|^2}{\partial \mathbf{W}^\top} \tag{6.7}$$

Since the matrices $\mathbf{W}$ and $\mathbf{W}^\top$ have the same values, the derivative of the global loss $\mathcal{L}$ with respect to $\mathbf{W}^\top$ is used to update the values of $\mathbf{W}$. The autoencoder is then trained with the stochastic gradient descent technique (SGD) to solve the optimization problem of minimizing the global loss. The objective function of the NLB autoencoder model is both non-convex and non-smooth due to the non-differentiable function $h(\cdot)$. Despite the fact that proving a general convergence is a hard problem, it has been observed in practice that the regularization term plays an important role and allows the model to converge to a local optimum (without the regularization term, the model oscillates and the bits in binary vectors keep flipping values at each iteration of the SGD optimization).

6.3 Experimental setup

This section presents the settings and the hyperparameters used to train the NLB autoencoder model (Subsection 6.3.1). It also describes the evaluation protocol followed to measure the performances of the binary and reconstructed vectors on several tasks, which are commonly used to evaluate word representations (see Section 1.4 in Chapter 1): intrinsic tasks with a word semantic similarity and a word analogy task (Subsection 6.3.2); and extrinsic tasks with a document classification, a question classification and a sentiment analysis task (Subsection 6.3.3). Another task (top-K queries) has been run to evaluate the computation efficiency of binary vectors (Subsection 6.3.4).

6.3.1 Training settings

Pre-trained word embeddings

The NLB model produces binary vectors from pre-trained word embeddings. The model has been trained to binarize several different pre-trained word embeddings:

- `dict2vec` [Tissier et al., 2017] which contains 2.3M word vectors and has been trained on the full English Wikipedia corpus.
- `fasttext` [Bojanowski et al., 2017] which contains 1M word vectors and has also been trained on the full English Wikipedia corpus.
- `GloVe` [Pennington et al., 2014] which contains 400K word vectors and has been trained on the English Wikipedia and the Gigaword 5 corpora.

All word vectors have 300 dimensions. The learning hyperparameters used are the ones from their respective paper. Since the three word embeddings are based on different learning methods (derivation of the Skip-gram model for `dict2vec` and `fasttext`, matrix factorization for `GloVe`), this demonstrates that the NLB model is general and works for all kinds of pre-trained real-valued vectors, independently of the method used to learn them.

Hyperparameters of the `NLB` autoencoder model

The size n of the latent representation of the autoencoder (which corresponds to the length of binary vectors) has been consecutively set to 64, 128, 256 and 512 to produce binary vectors with the same number of bits. The optimal other hyperparameters are found by using a grid search and are selected to minimize the reconstruction loss and the regularization loss described in Subsection 6.2.2. The model uses a batch size of 75 real-valued vectors for the stochastic gradient descent, 10 training epochs for `dict2vec` and `fasttext` (*i.e.* each vector of the embedding matrix $\mathbf{M}$ is seen 10 times) and 5 epochs for `GloVe` (the autoencoder converges faster due to the smaller number of word vectors in `GloVe`) and a learning rate of 0.001. The regularization hyperparameter λ_{reg} depends on the pre-trained vectors used and the binary vector size. It varies from 1 to 4 in the experiments but its influence on the performance is small ($\sim$2% variation).

6.3.2 Intrinsic evaluation tasks

Word semantic similarity

Both binary and reconstructed vectors are evaluated with the standard method of the word semantic similarity task, explained in details in Section 1.4.2 of Chapter 1. This task consists in computing the Spearman's rank correlation coefficient between the similarity scores attributed by humans to pairs of words and the similarity scores computed with the vectors of the words. The similarity score for real-valued vectors is computed with their cosine similarity while the similarity score of two binary vectors $(\mathbf{b}_1, \mathbf{b}_2)$ is computed with the Sokal & Michener similarity function [Sokal, 1958] defined as:

$$\text{sim}(\mathbf{b}_1, \mathbf{b}_2) = \frac{n_{11} + n_{00}}{n} \tag{6.8}$$

where n_{11} (resp. n_{00}) is the number of bits in $\mathbf{b}_1$ and $\mathbf{b}_2$ that are both set to 1 (resp. 0) simultaneously and n is the size of binary vectors. The datasets used for this task are:

- MEN [Bruni et al., 2014]
- RW [Luong et al., 2013]
- SimLex [Hill et al., 2015]
- SimVerb-3500 [Gerz et al., 2016]
- WordSim-353 [Finkelstein et al., 2001]

Word analogy

Binary and reconstructed vectors are also evaluated on the word analogy task, described in more details in Section 1.4.2 of Chapter 1. This evaluation follows the standard protocol used by Mikolov et al. [2013b]. The task consists in finding the word d in questions like "a is to b as c is to d" given the words a, b and c. First, the vector $\mathbf{v} = \mathbf{v}_b - \mathbf{v}_a + \mathbf{v}_c$ is computed with the respective word vectors of a, b and c. Then, if the closest vector of $\mathbf{v}$ in the embedding matrix is the vector associated to the word d, the analogy is correct. The score on the task reports the fraction of correctly guessed analogies among all analogies. For binary vectors, the vector addition is replaced by the `OR` bitwise operator and the vector subtraction by the `AND NOT` operator because adding or subtracting bits does not really make sense in the binary vecor space (*e.g.* subtracting the bit 1 to the bit 0). The closest vector of $\mathbf{v}$ is found by selecting the vector with the highest cosine (resp. Sokal & Michener) similarity among all the real-valued (resp. binary) vectors of the embedding

matrix **M**. The evaluation dataset of word analogies is separated into two parts: one consists of analogies about countries and currencies (semantic analogies), the other one about English grammar (syntactic analogies), as done in [Mikolov et al., 2013b].

6.3.3 Extrinsic evaluation tasks

The two intrinsic tasks measure how well the semantic and syntactic information from the original real-valued vectors has been preserved in the binary and reconstructed vectors, but they do not evaluate how well those vectors perform in downstream tasks. To evaluate the quality of binary and reconstructed vectors when they are used in downstream models, several additional evaluations have been performed on:

- document classification;
- question classification;
- sentiment analysis.

A detailed description of those tasks has been presented in Subsection 1.4.3 of Chapter 1. The evaluation follows the same protocol as described in the literature [Zhang et al., 2015, Joulin et al., 2016b] which consists in predicting the correct label given the Bag-of-Words representation of a text. A single hidden layer neural network where the input weights have been initialized with the binary or reconstructed vectors is used. The input weights are freezed during training so that the classification accuracy solely depends on the vectors used to initialize the neural network weights. The datasets used for the three tasks are:

- AG-News and DBpedia for the document classification task;
- Yahoo Answers for the question classification;
- Amazon and Yelp reviews (both polarity and full) for the sentiment analysis task.

Each dataset is split into a training and a test file and the same training and test files are used for all the models evaluating the binary and reconstructed vectors. Accuracy results are reported in Table 6.3 for binary vectors and in Table 6.5 for reconstructed vectors.

6.3.4 Computation speed evaluation

Evaluation tasks presented in Subsection 6.3.2 or in Subsection 6.3.3 only evaluate the quality of binary word embeddings from a linguistic point of view, based on their encoded information or on how useful the binary vectors are when they are used in downstream models. However, as explained in Section 6.1 of this chapter or in Chapter 4, binary vectors have the advantage of speeding up vector operations because they can be done with bitwise operators. To measure how much faster the computations are, a top-K queries evaluation is performed. It consists in finding the K closest vectors given a single word vector query. The closest vectors are the ones with the highest similarity to the query vector (Sokal & Michener similarity for binary vectors, cosine similarity for real-valued ones) and are found by performing a linear scan across all vectors. Two execution times are measured for both binary and real-valued vectors: the time it takes to get the results once the vectors are loaded in memory and the time it takes to load the vectors and perform the query.

6.4 Results and binary vectors analysis

This section presents the performances of binary and reconstructed vectors produced by the NLB autoencoder model on different evaluation tasks, respectively in Subsection 6.4.1 and in Subsection 6.4.2. The computational speed benefits of binary vectors are detailed in Subsection 6.4.3. Comparisons with other binarization methods are made in Subsection 6.4.4 and in Subsection 6.4.5. Lastly, Subsection 6.4.6 carries out an analysis of the semantic properties encoded into binary vectors.

6.4.1 Binary word embeddings performances

Word semantic similarity

Table 6.1 reports the Spearman's rank correlation scores obtained by the binarized vectors learned by the NLB autoencoder model (NLB row) and the scores of the original real-valued vectors ($\mathbb{R}$ column) whose size is 9,600 bits per vector (300 dimensions, 32 bits per `float` value) for different pre-trained word embeddings (`dict2vec`, `fasttext` and `GloVe`). The best scores of binarized vectors are reached with 256 or 512 bits. For `fasttext` and `GloVe`, the results of binary vectors are very close to the scores obtained by the original real-valued vectors (average absolute difference is smaller than 0.01). For `dict2vec`, the deviation is larger (between 0.03 and 0.05) but still in the same order of magnitude. Binarizing word vectors can lead to better scores compared to the original ones ($\mathbb{R}$ column), for example the 512-bit binary vectors of `fasttext` on WordSim (0.703 against 0.697) or the 256-bit binary vectors of `GloVe` on SimVerb (0.229 against 0.227). Moreover, the 64-bit binary vectors of `dict2vec` are better than the real-valued `GloVe` embeddings (which uses 9,600 bits of information per vector so binary codes are 150 times smaller) on SimVerb (0.253 against 0.227) and WordSim (0.637 against 0.609). This demonstrates that the method can produce rich semantically-related small binary codes but also more generally, that *binary vectors can encode the same semantic information as real-valued vectors.*

	dict2vec					fasttext					GloVe				
	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512
MEN	0.746					0.807					0.737				
NLB	-	0.661	**0.713**	0.703	0.713	-	0.579	0.720	0.759	**0.763**	-	0.461	0.633	0.694	**0.727**
LSH	-	0.477	0.562	0.626	0.678	-	0.477	0.607	0.700	0.750	-	0.508	0.620	0.649	0.710
RW	0.505					0.538					0.412				
NLB	-	0.365	0.420	0.456	0.456	-	0.368	0.447	**0.527**	0.527	-	0.251	0.343	**0.407**	0.402
LSH	-	0.264	0.372	0.417	**0.462**	-	0.345	0.403	0.475	0.465	-	0.263	0.330	0.358	0.383
SimLex	0.452					0.441					0.371				
NLB	-	0.320	0.381	**0.448**	0.429	-	0.251	0.380	**0.446**	0.430	-	0.205	0.314	**0.372**	0.368
LSH	-	0.296	0.359	0.402	0.395	-	0.287	0.320	0.386	0.411	-	0.247	0.305	0.331	0.346
SimVerb	0.417					0.356					0.227				
NLB	-	0.253	0.366	**0.384**	0.355	-	0.192	0.267	0.337	**0.351**	-	0.078	0.187	0.229	**0.230**
LSH	-	0.220	0.275	0.316	0.369	-	0.205	0.233	0.307	0.302	-	0.146	0.174	0.188	0.207
WordSim	0.725					0.697					0.609				
NLB	-	0.637	**0.716**	0.696	0.666	-	0.503	0.691	0.700	**0.703**	-	0.301	0.449	0.566	**0.603**
LSH	-	0.455	0.569	0.649	0.655	-	0.467	0.532	0.586	0.638	-	0.411	0.444	0.505	0.578

Table 6.1: Semantic similarity scores on several datasets for binary vectors produced with the proposed method (NLB row) or produced with the Local Sensitive Hashing method (LSH row) with different vector sizes: 64, 128, 256 and 512 bits. Several pre-trained word embeddings have been binarized (`dict2vec`, `fasttext` and `GloVe`). For each dataset, scores of original real-valued vectors are also reported ($\mathbb{R}$ column).

The average of word semantic similarity scores of all datasets (computed with the Fisher's transformation) for binary and original word embeddings (whose size is 9,600 bits per vector) are plotted in Figure 6.2. The 512-bit binary vectors of `GloVe` are on par with the real-valued vectors. However the performance loss is greater for `fasttext` and `dict2vec`. A possible explanation is that those word embeddings contain more vectors to binarize (1M vectors for `fasttext` and 2.3M for `dict2vec` whereas `GloVe` only has 400K vectors). Since the binary vector space has a finite size depending on the binary vector size, it becomes harder for the autoencoder to find a distribution of the binary vectors which preserves the semantic similarity between vectors when the number of vectors is larger.

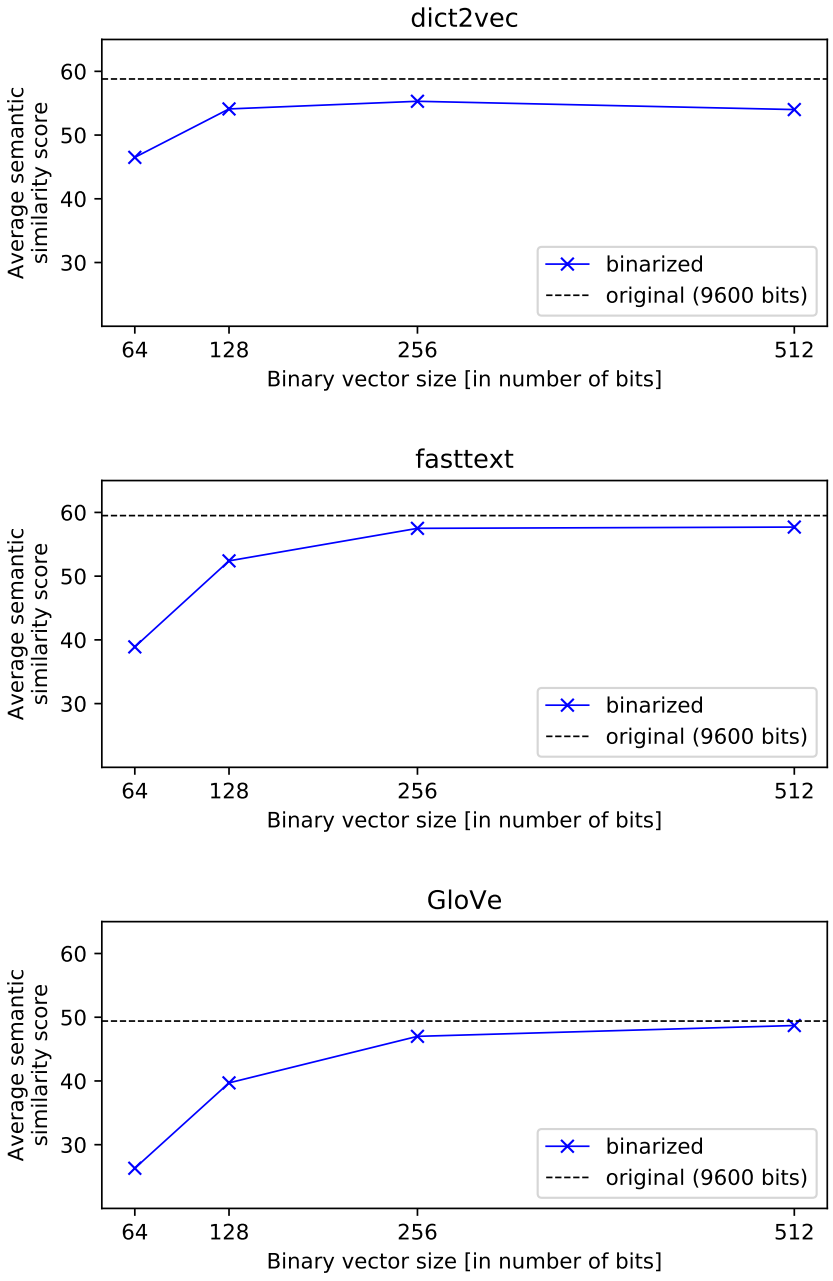


Figure 6.2: Average of the semantic similarity scores of all the datasets (computed with Fisher's transformation) for binary vectors of different sizes on a word semantic similarity task with different kinds of pre-trained vectors. The *original* baseline indicates the average scores obtained with the pre-trained real-valued embeddings.

Word analogy

Table 6.2 reports the scores of binary vectors on a word analogy task for semantic and syntactic analogies. Although the best scores are also obtained with larger binary codes (512 bits) like in the word semantic similarity task, the scores of binary vectors are lower than for real-valued vectors. This can be explained by the fact that the word analogy task is not suited to evaluate binary vectors. With real-valued vectors, analogies are found with vector additions and subtractions in the $\mathbb{R}^d$ space. In the binary space (where each value is either 0 or 1), adding or subtracting vectors does not make sense (like subtracting the bit 1 from the bit 0), resulting in lower scores of binary vectors on this analogy task.

	`dict2vec`					`fasttext`					`GloVe`				
	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512
Semantic	59.6					37.6					77.4				
`NLB`	-	2.6	12.0	26.7	**30.1**	-	2.3	7.5	18.0	**25.0**	-	8.5	26.7	53.4	**65.3**
`LSH`	-	0.8	4.6	14.9	29.9	-	0.8	6.4	13.0	20.4	-	6.1	25.9	47.9	59.3
Syntactic	54.0					87.4					67.0				
`NLB`	-	3.5	16.7	34.8	**36.2**	-	8.0	34.5	57.3	64.7	-	7.3	23.9	46.3	**52.4**
`LSH`	-	1.7	7.8	23.4	35.8	-	4.0	21.5	46.2	**65.7**	-	5.6	21.6	39.1	52.3

Table 6.2: Percentage of correctly predicted word analogies on several datasets (semantic and syntactic analogies) for binary vectors produced with the proposed method (`NLB` row) or produced with the Local Sensitive Hashing method (`LSH` row) with different vector sizes: 64, 128, 256 and 512 bits. Several pre-trained word embeddings have been binarized (`dict2vec`, `fasttext` and `GloVe`). For each dataset, scores of original real-valued vectors are also reported ($\mathbb{R}$ column).

Text classification

Table 6.3 reports the accuracy scores obtained on different datasets in the text classification tasks for binary vectors learned by the `NLB` autoencoder model (`NLB` row) and original real-valued vectors ($\mathbb{R}$ column). In all the text classification tasks, binary vectors achieve their best scores with 512 bits in general. Like for the word semantic similarity task, binary vectors are sometimes better than real-valued vectors. This is especially true for the 512-bit `fasttext` binarized vectors where the accuracy goes from 95.0 to 97.3 on DBpedia or from 49.0 to 49.8 on Amazon Full reviews. Using binarized vectors of less than 512 bits leads to better compression rates (because fewer bits are needed to encode each vector) but it causes a slight decrease in performances. The 256-bit binary vectors of `GloVe` are 37.5 times smaller than the original real-valued word embeddings (whose size is 9,600 bits per vector) but the accuracy drops between 0.4% and 2.5% depending on the dataset. The 64-bit binary vectors of `dict2vec` (compression rate of 150) have a loss of accuracy of about 4% on AG-News and DBpedia, about 11% on Yahoo Answers and about 16% on Amazon Full reviews compared to the scores obtained by real-valued vectors. The two latter datasets (Yahoo Answers and Amazon Full reviews) are the biggest ones, with respectively 1.4M and 3M training samples while AG-News and DBpedia respectively contain 120k and 560k training samples. As the dataset becomes larger, the amount of information required to correctly classify documents also increases, so is the accuracy loss on those datasets when small binary vectors are used. Overall, these experiments show once again that binary embeddings are competitive with real-valued ones.

	dict2vec					fasttext					GloVe				
	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512
AG-News	89.0					86.9					89.5				
NLB	-	85.3	85.9	87.7	87.8	-	84.5	85.9	87.3	87.7	-	84.0	87.2	88.5	88.5
LSH	-	78.8	82.6	86.1	**88.1**	-	77.5	83.3	86.1	**88.8**	-	83.5	86.6	88.4	**88.6**
DBpedia	97.6					95.0					97.2				
NLB	-	94.1	96.1	97.0	97.3	-	91.7	95.1	96.6	**97.3**	-	90.9	95.0	96.8	**97.2**
LSH	-	89.6	94.2	96.5	**97.4**	-	87.4	93.8	96.2	97.2	-	90.4	94.2	96.3	97.2
Yahoo Ans.	68.1					67.2					68.1				
NLB	-	60.7	63.8	66.0	66.8	-	60.4	63.9	66.4	**67.8**	-	57.5	62.5	66.4	66.1
LSH	-	52.3	59.9	64.5	**67.1**	-	52.2	59.5	64.9	66.9	-	56.8	62.0	65.3	**67.0**
Amazon Full	47.5					49.0					47.1				
NLB	-	39.9	43.9	46.8	47.7	-	39.0	43.9	47.9	**49.8**	-	37.4	42.6	46.7	47.8
LSH	-	38.3	42.5	45.6	**48.1**	-	38.6	42.7	47.3	49.5	-	37.9	43.0	45.9	**48.6**
Amazon Pol.	84.2					85.6					83.8				
NLB	-	76.3	80.7	83.2	83.8	-	75.1	80.2	84.5	**85.8**	-	73.1	78.9	83.2	84.4
LSH	-	74.3	79.0	81.9	**84.5**	-	73.8	78.5	83.4	85.7	-	74.7	79.1	82.1	**85.0**
Yelp Full	52.5					52.1					52.7				
NLB	-	45.1	48.7	51.6	52.0	-	44.2	49.7	53.0	**54.6**	-	42.7	48.4	51.8	53.2
LSH	-	43.0	47.7	51.0	**53.1**	-	44.3	47.6	52.4	54.3	-	43.6	48.2	51.5	**53.4**
Yelp Pol.	87.8					88.0					87.9				
NLB	-	80.8	84.5	86.6	87.6	-	80.1	84.5	88.1	89.5	-	77.8	84.2	86.9	**88.7**
LSH	-	77.9	82.8	86.1	**88.0**	-	80.3	82.2	87.2	**89.8**	-	79.0	83.1	86.6	88.6

Table 6.3: Accuracy of correctly predicted labels in document classification (top), question classification (middle) and sentiment analysis (bottom) on several datasets for binary vectors produced with the proposed method (NLB row) or produced with the Local Sensitive Hashing methods (LSH row) with different vector sizes: 64, 128, 256 and 512 bits. For each dataset, scores of original real-valued vectors are also reported ($\mathbb{R}$ column).

6.4.2 Reconstructed word embeddings performances

Word semantic similarity and word analogy

Table 6.4 reports the scores obtained on a word semantic similarity and on a word analogy tasks using the reconstructed real-valued vectors (*recons.* row). Reconstructed vectors are computed with the decoder part of the NLB autoencoder and the learned latent binary vectors of the model, which explains why there are as many different reconstructed vectors as there are different sizes of binary vectors.

For the word semantic similarity task (results in the top part in Table 6.4), fasttext and dict2vec work best in most cases using 256-bit binary vectors while GloVe requires larger binary vectors (512 bits) to reconstruct good real-valued vectors. On most datasets, vectors reconstructed from the 512-bit GloVe binary vectors outperforms the original real-valued vectors: 0.382 against 0.371 on SimLex, 0.247 against 0.227 on SimVerb and 0.620 against 0.609 on WordSim. For dict2vec and fasttext, binarizing and then reconstructing real-valued vectors from the binary vectors causes a loss of performances compared to the scores of original word vectors: between -4.8% (on WordSim) and -11.7% (on RW) for dict2vec; between -8.2% (on WordSim) and -28.1% (on SimVerb) for fasttext. The performance loss of reconstructed vectors is larger for fasttext than for dict2vec due to their different word embedding learning method. fasttext also encodes additional morphological information into word vectors by considering that a word is the sum of its subwords, which is harder to reconstruct after a dimension reduction (the binarization).

For the word analogy task (results in the bottom part in Table 6.4), scores of reconstructed vectors are far from the original ones: −39% for `dict2vec`, −49% for `fasttext` and −19% for `GloVe` (average loss of semantic and syntactic scores). `GloVe` reconstructed vectors have a smaller loss of performances compared to `dict2vec` and `fasttext`: since the number of vectors to binarize is smaller for `GloVe` embeddings, the model can preserve the arithmetic relations between vectors more easily. The low performances of reconstructed vectors is also a consequence of the low performances of binary vectors on this task: if analogy relations have not been preserved in the binary vector space during the binarization, then the model cannot recreate them from the binary word vectors.

	`dict2vec`					`fasttext`					`GloVe`				
	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512
MEN	0.746					0.807					0.737				
recons.	-	0.645	**0.696**	0.678	0.647	-	0.458	0.562	**0.623**	0.593	-	0.436	0.505	0.685	**0.722**
RW	0.505					0.538					0.412				
recons.	-	0.357	0.416	**0.446**	0.393	-	0.289	0.316	**0.457**	0.441	-	0.248	0.292	0.364	**0.405**
SimLex	0.452					0.441					0.371				
recons.	-	0.304	0.379	**0.424**	0.393	-	0.192	0.300	**0.405**	0.340	-	0.196	0.191	0.342	**0.382**
SimVerb	0.417					0.356					0.227				
recons.	-	0.237	0.354	**0.375**	0.294	-	0.128	0.182	**0.256**	0.253	-	0.080	0.124	0.221	**0.247**
WordSim	0.725					0.697					0.609				
recons.	-	0.614	**0.690**	0.674	0.588	-	0.365	0.536	**0.640**	0.536	-	0.265	0.422	0.565	**0.620**
Semantic	59.6					37.6					77.4				
recons.	-	2.6	10.2	22.8	**30.9**	-	1.8	5.0	14.6	**15.2**	-	7.7	23.0	49.1	**62.8**
Syntactic	54.0					87.4					67.0				
recons.	-	3.6	16.1	31.2	**37.5**	-	4.6	14.6	50.8	**53.1**	-	7.3	21.7	44.6	**54.0**

Table 6.4: Scores of real-valued vectors reconstructed by the `NLB` autoencoder model (*recons.* row) from learned binary vectors (of 64, 128, 256 and 512 bits) on a word semantic similarity task (top) and a word analogy task (bottom) for several datasets. For each dataset, scores of original real-valued vectors are also reported ($\mathbb{R}$ column).

Text classification

Table 6.5 reports the accuracy scores obtained on several extrinsic evaluation tasks (document classification, question classification and sentiment analysis) using the reconstructed real-valued vectors (*recons.* row). As explained at the beginning of Section 6.4.2, reconstructed vectors are computed with the decoder part of the `NLB` autoencoder and the learned latent binary vectors of the model. On downstream NLP tasks like document classification or sentiment analysis, the results of reconstructed vectors are very consistent: they exhibit close performances to binary word vectors, which in turn exhibit almost equal performances to the original vectors, independently of the initial pre-trained vectors used (*e.g.* 51.6 against 52.5 for the 256-bit binary vectors of `dict2vec` on Yelp Full; 96.4 against 95.0 for the 256-bit binary vectors of `fasttext` on DBpedia; 47.3 against 47.1 for the 512-bit binary vectors of `GloVe` on Amazon Full). Although the pairwise word semantic similarities are not perfectly preserved in the reconstructed vectors, text classification tasks only need vectors close enough for those of the same category but not necessarily very accurate within a category. This makes binary or reconstructed vectors good for real use case downstream tasks. The optimal number of bits required to reconstruct real-valued vectors with good performances in text classification tasks is the same as for the word semantic similarity task (256 bits for `dict2vec` and `fasttext`, 512 bits for `GloVe`).

	dict2vec					fasttext					GloVe				
	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512	$\mathbb{R}$	64	128	256	512
AG-News	89.0					86.9					89.5				
recons.	-	85.2	86.3	**87.9**	87.2	-	82.8	84.3	**87.7**	87.3	-	83.9	87.7	88.6	**89.2**
DBpedia	97.6					95.0					97.2				
recons.	-	94.0	95.9	**96.8**	96.6	-	89.5	92.6	**96.4**	96.0	-	91.2	95.2	96.8	**97.0**
Yahoo Ans.	68.1					67.2					68.1				
recons.	-	60.8	63.8	65.9	**66.0**	-	60.0	62.9	66.3	**66.8**	-	58.4	64.3	66.7	**67.0**
Amazon Full	47.5					49.0					47.1				
recons.	-	40.1	44.0	**46.8**	46.2	-	39.1	43.8	48.1	**48.5**	-	39.8	45.3	47.1	**47.3**
Amazon Pol.	84.2					85.6					83.8				
recons.	-	76.6	80.8	**83.2**	82.4	-	75.1	80.2	84.7	**84.8**	-	76.6	80.2	83.6	**83.7**
Yelp Full	52.5					52.1					52.7				
recons.	-	45.3	48.8	**51.6**	50.9	-	43.5	47.8	53.0	**53.1**	-	43.4	50.3	52.3	**52.8**
Yelp Pol.	87.8					88.0					87.9				
recons.	-	80.9	84.5	**86.6**	86.1	-	79.6	84.0	88.2	**88.5**	-	78.6	85.7	87.5	**87.7**

Table 6.5: Accuracy of correctly predicted labels in document classification (top), question classification (middle) and sentiment analysis (bottom) on several datasets for real-valued vectors reconstructed by the NLB autoencoder model (*recons.* row) from learned binary vectors of 64, 128, 256 and 512 bits. For each dataset, scores of original real-valued vectors are also reported ($\mathbb{R}$ column).

6.4.3 Speed improvements in top-K queries

The execution time (in milliseconds) of top-K queries benchmarks for binarized and original real-valued GloVe vectors are reported in Table 6.6. The first three rows (Top 1, Top 10 and Top 50) indicate the time used by the program to perform the query *after* the vectors are loaded into the memory *i.e.* the time needed to linearly scan all the vectors, compute the similarity with the query vector and select the K highest values. For binary vectors, the Sokal & Michener similarity is used while real-valued vectors use the cosine similarity. Finding the 10 closest vectors of a query vector is **30 times faster** with 256-bit binarized GloVe vectors than with the real-valued vectors and it can be up to 36 times faster with 64-bit binary vectors. Having faster computations of vector similarities is not the only interest of binary vectors. Since they take less space in memory, they are also much faster to load. The last row in Table 6.6 indicates the time needed to load the vectors from a file (stored on a SSD) into the memory (stored in RAM) and perform a top-10 query. It takes 23.5 seconds to load the real-valued vectors and run the query whereas it only takes 310 milliseconds to do it with 256-bit binary vectors, which is **75 times faster**.

Top-K query	64-bit	128-bit	256-bit	512-bit	Real-valued
Top 1	2.71	2.87	3.23	4.28	97.89
Top 10	2.72	2.89	3.25	4.29	98.08
Top 50	2.75	2.91	3.27	4.32	98.44
Loading + Top 10	160	213	310	500	23500

Table 6.6: Execution time (in milliseconds) to run a top-K query on binary and real-valued vectors. For the first three rows (Top 1, Top 10 and Top 50), the time needed to read and load the vectors into the memory is not taken into account. The loading time of vectors from the disk into the memory is only counted in the last row.

6.4.4 Comparison with the naive approach

A naive approach to produce binary vectors from pre-trained word embeddings is to map negative values of vectors to 0 and positive values to 1. Unfortunately, binary vectors produced with this method have the same number of bits as the the number of dimensions of original real-valued vectors. Since most pre-trained word vectors are not available in 64, 128 or 256 dimensions, binarized vectors are not aligned with CPU registers by using the naive binarization method, which is an important requirement to achieve fast vector operations (see Section 6.1 for a detailed explanation about the motivations to have binary vectors aligned with CPU registers). To produce aligned vectors with the naive method, one has first to train real-valued embeddings with 64, 128 or 256 dimensions and then apply the naive binarization approach to all vector values. First, this process is slower than using the `NLB` autoencoder model. It requires 8 hours while the `NLB` autoencoder only needs 13 minutes to produce 256-bit binary vectors from pre-trained `dict2vec` vectors. Second, binary vectors produced with the naive binarization have worse results than those produced with the `NLB` autoencoder model. For `dict2vec`, the average word semantic similarity scores are 0.447 and 0.496 for the naive 64-bit and 128-bit binary vectors while the 64-bit and 128-bit `NLB` binary vectors achieve respectively 0.465 and 0.541.

6.4.5 Comparison with other binarization methods

The *Locality Sensitive Hashing* method (`LSH`) is used to produce binary vectors of 64, 128, 256 and 512 bits from the same real-valued vectors as the `NLB` autoencoder model. They are evaluated on the same tasks as the `NLB` method and their scores are reported in Table 6.1 for the word semantic similarity task, in Table 6.2 for the word analogy task and in Table 6.3 for the text classification tasks. On the word semantic similarity and the word analogy tasks, `NLB` binary vectors are always better than the `LSH` ones (*e.g.* 0.703 against 0.638 for the 512-bit binary vectors of `fasttext` on WordSim; 34.8 against 23.4 for the 256-bit binary vectors of `dict2vec` on Syntactic analogies), except for the 64-bit binary vectors of `GloVe` and some 64-bit binary vectors of `fasttext`. On the classification tasks, 512-bit binary vectors of `NLB` are on par with the `LSH` ones, but the 128-bit and 256-bit binary vectors of `NLB` have better performances than the respective `LSH` vectors. The `NLB` model gives the best size to performance ratio for small binary vectors.

`NLB` binary vectors have also been compared to the binary vectors produced by [Faruqui et al., 2015b]. Faruqui et al. binary vectors are mostly sparse (90% of bits are set to the value 0) and therefore are too large to be computationally interesting: the number of bits is set to $3,000$ in their method so they do not fit in CPU registers. They also have lower performances than the `NLB` binary vectors: 0.560 for their average word semantic similarity score while the `NLB` model achieves 0.575 with 256-bit binary vectors.

6.4.6 Qualitative analysis of binary embeddings

Visualization of activated dimensions in binary vectors

In Figure 6.3, the 50 most similar (top) and the 50 least similar (bottom) vectors to the 512-bit vectors of 4 words ("queen", "automobile", "jupiter" and "man") are plotted with red (resp. white) pixels for bits set to 1 (resp. 0). Some vertical stripes are clearly visible in top areas: similar binary vectors have common bits on many dimensions and therefore have the same pixel color for those dimensions. In bottom areas, no vertical patterns are visible because non-similar binary vectors do not have a lot of bits in common and therefore do not have the same pixel color on many dimensions. This visualization shows that binarized vectors have similar semantic properties as real-valued embeddings, which are known to encode common semantic features on common sets of dimensions.

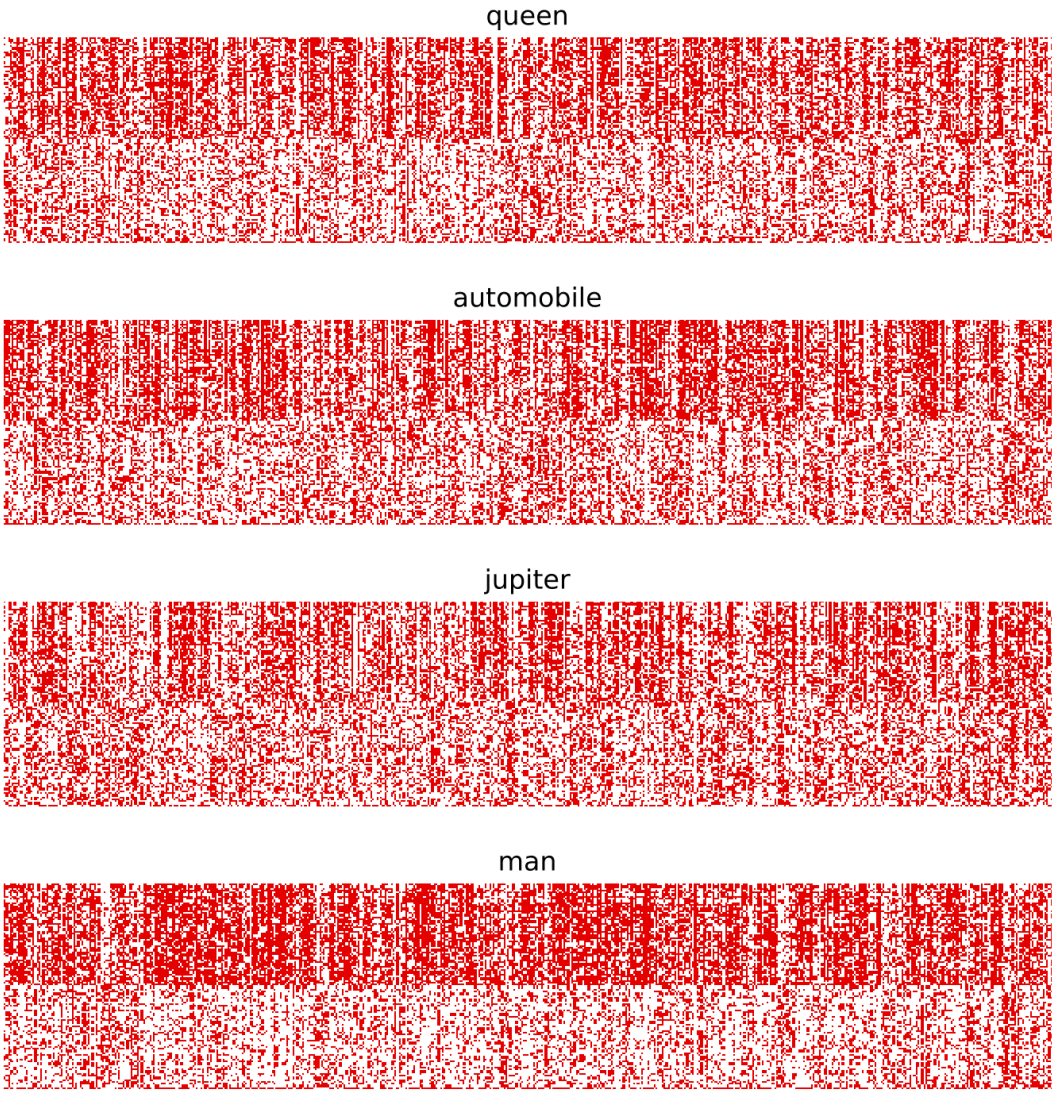


Figure 6.3: Visualization of the activated bits (red pixels for 1, white pixels for 0) in the binary vectors of the 50 closest (top) and the 50 furthest (bottom) neighbors of vectors.

Evolution of word vector similarity

The word semantic similarity task is based on datasets containing pairs of words associated to a value which represents the semantic similarity (evaluated by human annotators) between the words. For some pairs (reported in Table 6.7), the similarity of the pre-trained real-valued vectors of the corresponding words is far from the human assigned value, but the similarity of the binarized version of the vectors is closer to the human assigned value, making the binary embeddings closer to the human semantic representation of words.

Words	Human judgment	Binary similarity	Real-valued similarity
dollar – buck	0.92	0.56	0.13
seafood – sea	0.75	0.62	0.24
money – salary	0.79	0.58	0.41
car – automobile	0.98	0.69	0.60

Table 6.7: Semantic similarity for some pairs of words, evaluated by human or computed with binary and real-valued vectors.

6.5 Conclusion

Methods to learn word embeddings commonly associate a numerical vector to each word of a vocabulary. As the vocabulary gets bigger over the time because the size of training corpora is also largely increasing over the years, it is not uncommon to see word embeddings requiring several gigabytes in memory for storage. With new learning methods like contextual representations, the memory and computational power requirements are even bigger due to the huge number of parameters in the deep learning architectures involved.

Although most of the word embeddings learning methods are trained on large computing servers without any problems, their important requirements of computing resources prevents their use on low-resource devices like smartphones because of their limited memory capacity and their small computing power. This shortcoming of such methods motivated the emergence of new approaches (presented in Chapter 4) to reduce the size of vector representations and to find new solutions to minimize the amount of computing power needed to perform vector operations, with methods such as vector quantization which takes advantage of CPU-optimized functions on integer or binary vectors to speed up vector computations. However, most of the existing methods to reduce the size of vectors or to accelerate vector computations are for fast similarity searches (*i.e.* for vector retrieval tasks) not for using the size-reduced vectors in downstream NLP models.

This chapter presented the second contribution of this thesis, to address the problem of finding word embeddings with small size in memory and which allow fast vector operations while preserving the linguistic properties of words so they can be used in downstream NLP models. It details a new autoencoder architecture with a binary latent representation to transform pre-trained real-valued word vectors into binary vectors of any size. This is particularly suitable when the size of binary vectors is the same as the CPU registers size because it allows a significant increase in vector operations computing speed. This method, named `NLB`, has the advantage of being simple yet powerful and allows to preserve the semantic information of the original pre-trained vectors into the binary word embeddings.

Binary word embeddings produced with the `NLB` model exhibit almost the same performances as the original real-valued vectors on both word semantic similarity and text classification tasks. Furthermore, since the binary representations require less space in memory (a 256-bit binary vector is 97% smaller than a traditional 300-dimensional real-valued word vector), it allows one to run a top-K query 30 times faster than with real-valued vectors since the binary similarity is much faster to compute than a regular cosine similarity. Additionally, it is possible to reconstruct real-valued vectors from the binary representations using the decoder part of the `NLB` autoencoder. Vectors reconstructed from the binary vectors exhibit similar performances on downstream tasks like document classification or sentiment analysis compared to the original pre-trained word vectors.

Conclusion and Perspectives

In this thesis, we have considered to address an important problem to automatically solve linguistic tasks, like translating a whole document without the need of a human translator or processing a large number of online comments to find relevant information. Most of the algorithms used to solve those tasks are based on numerical representations of the language, and more specifically on representations of words. These word representations should encode the linguistic properties of words, as the algorithms need to have information about the language to perform well in aforementioned tasks. This thesis focuses on learning these representations and solves the problem of improving them along two complementary axes. The first axis is to improve the quality and amount of linguistic information encoded into word representations so that the algorithms using them have access to a greater knowledge of the language to solve NLP tasks. The second axis is to improve the word representations to be more computationally efficient, *i.e.* requiring less computing resources such as memory and computing power so that the word representations can be used on low-resource devices and not only on large computing servers as it is commonly the case.

Summary of contributions

To learn word representations (or word embeddings) which encode linguistic properties, most common methods rely on the distributional hypothesis that words appearing frequently together have the same meaning. Those methods use the cooccurrence information of words to assess the similarity of words and they learn word representations which reflect these similarities between words. However, they use generic text corpora to count the cooccurrences, which causes word representations to lack specific semantic information that are not present in those corpora. The first contribution of this thesis addresses the problem of incorporating more semantic information into word embeddings. The proposed model, named `dict2vec` [Tissier et al., 2017], leverages the linguistic information one can find in lexical dictionaries to improve the semantic knowledge encoded into word embeddings. It builds pairs of related words based on their cooccurrences in dictionary definitions, which are then used in addition to the cooccurrence information of a generic corpus to learn word embeddings. The vectors learned by `dict2vec` provide significant improvements in word semantic similarity tasks compared to other common methods used for learning word embeddings. This contribution also shows that the additional linguistic information found in dictionaries is richer than the one found in other lexical resources.

As training corpora get bigger over time, the size of the vocabulary needed to solve linguistic tasks also increases and so is the number of word embeddings to learn. Many methods, including `dict2vec`, require several gigabytes of memory to store word embeddings. Moreover, word embeddings are encoded as real-valued vectors, which are not suitable for low-resource devices because they have small computing power for floating-point operations, in addition to a limited memory capacity. Therefore, using word embeddings on such devices is not feasible in practice. The second contribution of this thesis addresses the problem of reducing the memory size of word embeddings as well as making vector operations on word embeddings faster. The proposed method *binarizes* word em-

beddings *i.e.* it transforms pre-trained real-valued vectors, like the `dict2vec` vectors, into binary vectors. The method, named `NLB` [Tissier et al., 2019], uses an autoencoder architecture where the latent representation is binary. The contribution proposes an original scheme to update the parameters of this autoencoder although its objective function is non-differentiable. The `NLB` model can transform real-valued vectors to binary vectors of any size but in practice, sizes of 64, 128 or 256 bits are the most suited ones because they allow binary word embeddings to be aligned with CPU registers and benefit from the fast optimized binary operations of processors. Binary word vectors of 256 bits learned by the `NLB` model are 97% smaller than their respective pre-trained real-valued vectors while their performances on tasks like word semantic similarity or document classification only decrease by 2% on average. Moreover, this contribution shows that performing a top-K query with the produced binary vectors is 30 times faster than with real-valed vectors.

Both contributions of this thesis deal with a specific (but complementary) problem of word embeddings. `dict2vec` is a novel method to extract semantic knowledge from lexical dictionaries in an unsupervised way and then incoporate this information into word embeddings to produce semantically-richer word vectors. `NLB` is a new architecture used to reduce the size of word embeddings by transforming them into binary vectors, which also has the benefit of making vector operations faster. With memory-reduced word embeddings, one can run models on low-resource devices which is of growing interest with the democratization of NLP applications on smartphones. Furthermore, all the source code to replicate the contributions as well as the pre-trained `dict2vec` vectors have been made publicly available[1], to foster their use by the community for other future models.

Perspectives and ideas for future works

Since both contributions are quite different in their approaches because they focus on different angles to improve word embeddings, their perspectives are also distinct. Several ideas to enhance proposed models have been presented in their respective chapters (Chapter 5 for `dict2vec`, Chapter 6 for `NLB`), but this section aims to present more general concepts and directions of improvement.

As detailed in Chapter 5 and reminded in the summary of contributions, `dict2vec` uses dictionaries to extract semantic knowledge. However, `dict2vec` does not consider the possible polysemy of words (when a word has multiple meanings which depend on the context in which it is used). Therefore, `dict2vec` considers the different sense definitions of polysemous words as a single entity, not separately. This results in learning a single embedding for each word, even for polysemous words whereas it would be more appropriate to learn one embedding for each sense in the case of polysemous words. New word embeddings learning methods released after `dict2vec` (2017) like `ELMO` [Peters et al., 2018] or `BERT` [Devlin et al., 2019] model the idea that different meanings should have different word representations. One direction to improve `dict2vec` is to consider separately the definitions of different meanings when learning word embeddings or when using them as additional source of information in downstream NLP tasks. During this thesis, I had the opportunity to be a visiting scientist at the *Sapienza University of Roma* in Italy for a stay of two months and start to work on this idea with Pr. Roberto Navigli, specialist in word sense disambiguation (two ERC grants on this topic of research). The first results of the polysemic improved version of `dict2vec` were not as good as for the methods which consider polysemy information from other sources of knowledge in their learning architecture. The main reason is that other methods learn one vector per sense according to a standard index of senses (word senses from WordNet) but the senses in dictionaries are not always aligned with this index (*i.e.* the first sense of a polysemous word in WordNet is not

[1] A complete description of the released source code is in the chapter *List of Released Softwares*.

always the same as its first sense in the dictionary). Since the polysemous evaluation tasks expect to have sense vectors labeled with the sense index of WordNet, an alignment step is required between the definitions of WordNet and those of dictionaries. This alignment process is difficult because the different senses of words have small definitions in WordNet. For this reason, this direction of improvement remains a work in progress.

Another way to improve `dict2vec` is to extend it to other languages. Indeed, `dict2vec` only learns embeddings for English words, and therefore only uses English dictionaries. Since dictionaries exist in all common languages, `dict2vec` can easily be ported to improve the semantic knowledge of word embeddings in different languages. Translation dictionaries can also be a possible source of semantic knowledge to learn aligned word embeddings of two languages, mainly to use them thereafter in downstream translation models.

Concerning the binarization of representations, an interesting perspective for future work is to see how to apply the `NLB` method to other kinds of representations like sentence embeddings or document embeddings in order to use them in different types of NLP applications. For example, below is a table indicating the number of articles among the respective version of Wikipedia of the ten languages with the most articles[2]:

		Language	Number of articles
1	EN	English	6,024,168
2	DE	German	2,403,129
3	FR	French	2,184,995
4	RU	Russian	1,601,114
5	IT	Italian	1,586,063
6	ES	Spanish	1,579,838
7	JA	Japanese	1,192,418
8	ZH	Chinese	1,099,305
9	PT	Portuguese	1,022,673
10	FA	Persian/Farsi	712,160

Number of different articles in ten language specific versions of Wikipedia (February 2020).

The number of articles in the different language versions of Wikipedia is in the order of millions (about the same number, if not more, than the number of vectors in common word embedding matrices). Learning and storing the real-valued vector representation of each article (*e.g.* for tasks like finding the closest article given a query word) is expensive. Learning a *binary representation* for each article would allow one to perform queries faster. The binarization process can also be applied to each individual sentence of each article, which would speed up the retrieval of a specific piece of information to answer a query like a question. During this thesis, I also had the opportunity to be a visiting scientist at the *RIKEN Center for Advanced Intelligence Project* in Japan for another stay of five months to work on the task of classifying each article among a hierarchical tree of 200 classes for the thirty language versions of Wikipedia with the highest number of articles. Several problems occurred during the development of a model to solve this task. First, the distribution of classes is unbalanced. In Wikipedia, 30% of all articles are about a person whereas some other classes represent less than 0.001% of all articles. The binary representations learned by the model were not able to differentiate the different classes as the representations of articles of less frequent classes were too similar to the representations of articles of persons, and therefore misclassified by the model. Second, some articles are

[2]These numbers come from `https://stats.wikimedia.org/`.

very long and contain many information (*e.g.* the Wikipedia article of "France" talks about its economy, its history, its landscapes, etc.). To learn a binary representation for this article which can be used to correctly classify it as a "Country", only the relevant information must be encoded into the binary vector, the rest of the information has to be discarded so the model will not classify it as a "Person" (because the page contain many names of former presidents and kings) or as a "Mountain" (because the page also countains many names of french mountains). For these reasons, this classification task is difficult and is still a work in progress.

`NLB` and other methods like hashing or quantization only focus on transforming data representations. However, internal representations of machine learning models like the different layers in a neural network are not very different. Another interesting perspective of binarization, which is also valid for other methods like quantization, is to transform the internal representations of models. The benefits of size reduction and faster computations would therefore also be applicable to entire models, which is of major interest for replicability on low-resource devices given the exponential growth of the number of parameters in the most recent NLP models like `BERT` [Devlin et al., 2019] or `RoBERTa` [Liu et al., 2019].

List of Publications

Publications in International Conferences

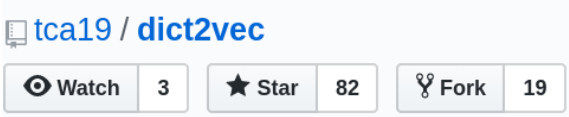
Julien Tissier, Christophe Gravier, and Amaury Habrard. "Dict2vec: Learning word embeddings using lexical dictionaries". In *Conference Methods in Natural Language Processing (EMNLP 2017)*, pages 254–263, 2017.

Julien Tissier, Christophe Gravier, and Amaury Habrard. "Near-lossless binarization of word embeddings". In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, volume 33, pages 7104–7111, 2019.

List of Released Softwares

Software released with the `dict2vec` publication:

`https://github.com/tca19/dict2vec`

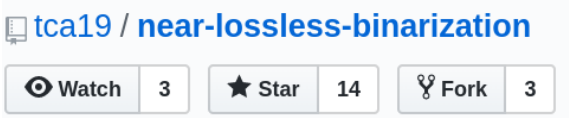


This repository contains:

- scripts to automatically download the webpages of 4 different English dictionaries (Cambridge, Oxford, Collins and dictionary.com), parse their content and extract word definitions;
- scripts to generate strong and weak pairs from the downloaded definitions;
- script to download the latest Wikipedia dump and generate the 50M tokens, the 200M tokens and the full Wikipedia training corpora;
- source code to train the `dict2vec` model and learn word embeddings using the pairs generated from the definitions of dictionaries;
- source code to evaluate word embeddings (learned with `dict2vec` or with any other common method) on a word semantic similarity task on thirteen evaluation datasets;
- links to download pre-trained vectors learned with the `dict2vec` model on a 50M tokens, a 200M tokens and a full Wikipedia corpora.

Software released with the `NLB` publication:

`https://github.com/tca19/near-lossless-binarization`



This repository contains:

- source code to train the `NLB` autoencoder to transform any pre-trained real-valued word embeddings into binary vectors;
- source code to evaluate binary word vectors on a word semantic similarity task on five evaluation datasets;
- source code to find the K closest words of a query word from the binary word vectors.

List of Figures

List of Tables

Bibliography

Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*, pages 459–468. IEEE, 2006.

Sanjeev Arora, Yuanzhi Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. Random walks on context spaces: Towards an explanation of the mysteries of semantic word embeddings. *arXiv preprint arXiv:1502.03520*, pages 1–23, 2015.

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. *The semantic web*, pages 722–735, 2007.

Taiwo Oladipupo Ayodele. Types of machine learning algorithms. *New advances in machine learning*, pages 19–48, 2010.

Lalit R Bahl, Peter F Brown, Peter V de Souza, and Robert L Mercer. A tree-based statistical language model for natural language speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(7):1001–1008, 1989.

Collin F Baker, Charles J Fillmore, and John B Lowe. The berkeley framenet project. In *Proceedings of the 17th international conference on Computational linguistics-Volume 1*, pages 86–90. Association for Computational Linguistics, 1998.

Ron Bekkerman, Ran El-Yaniv, Naftali Tishby, and Yoad Winter. Distributional word clusters vs. words for text categorization. *Journal of Machine Learning Research*, 3 (Mar):1183–1208, 2003.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

M Bierwisch. On classifying semantic features. *Bierwisch and Heidolph*, pages 27–50, 1970.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association of Computational Linguistics*, 5:135–146, 2017. URL `http://aclweb.org/anthology/Q17-1010`.

Antoine Bordes, Jason Weston, and Nicolas Usunier. Open question answering with weakly supervised embedding models. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 165–180. Springer, 2014.

Tom Bosc and Pascal Vincent. Auto-encoding dictionary definitions into consistent word embeddings. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1522–1532, 2018.

Peter F Brown, John Cocke, Stephen A Della Pietra, Vincent J Della Pietra, Frederik Jelinek, John D Lafferty, Robert L Mercer, and Paul S Roossin. A statistical approach to machine translation. *Computational linguistics*, 16(2):79–85, 1990.

Elia Bruni, Nam-Khanh Tran, and Marco Baroni. Multimodal distributional semantics. *J. Artif. Intell. Res.(JAIR)*, 49(1-47), 2014.

W Cavnar. Using an n-gram-based document representation with a vector processing retrieval model. *NIST SPECIAL PUBLICATION SP*, pages 269–269, 1995.

Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

Ting Chen, Martin Renqiang Min, and Yizhou Sun. Learning k-way d-dimensional discrete codes for compact embedding representations. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 854–863, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/chen18g.html`.

Billy Chiu, Anna Korhonen, and Sampo Pyysalo. Intrinsic evaluation of word vectors fails to predict extrinsic performance. In *Proceedings of the 1st workshop on evaluating vector-space representations for NLP*, pages 1–6, 2016.

Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(Aug):2493–2537, 2011.

Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. Very deep convolutional networks for text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 1107–1116, Valencia, Spain, April 2017. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/E17-1104`.

Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. L2 regularization for learning kernels. *arXiv preprint arXiv:1205.2653*, 2012.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41, 1990.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://www.aclweb.org/anthology/N19-1423`.

Manaal Faruqui, Jesse Dodge, Sujay Kumar Jauhar, Chris Dyer, Eduard Hovy, and Noah A. Smith. Retrofitting word vectors to semantic lexicons. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1606–1615. Association for Computational Linguistics, 2015a.

Manaal Faruqui, Yulia Tsvetkov, Dani Yogatama, Chris Dyer, and Noah Smith. Sparse overcomplete word vector representations. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1491–1500, Beijing, China, 2015b. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P15-1144`.

Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing search in context: The concept revisited. In *Proceedings of the 10th international conference on World Wide Web*, pages 406–414. ACM, 2001.

John Rupert Firth. *Papers in Linguistics 1934-1951: Repr.* Oxford University Press, 1957.

Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. Ppdb: The paraphrase database. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 758–764, 2013.

Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013.

Daniela Gerz, Ivan Vulić, Felix Hill, Roi Reichart, and Anna Korhonen. Simverb-3500: A large-scale evaluation set of verb similarity. *arXiv preprint arXiv:1608.00869*, 2016.

Michael U Gutmann and Aapo Hyvärinen. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of Machine Learning Research*, 13(Feb):307–361, 2012.

Guy Halawi, Gideon Dror, Evgeniy Gabrilovich, and Yehuda Koren. Large-scale learning of word relatedness with constraints. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1406–1414. ACM, 2012.

Felix Hill, Roi Reichart, and Anna Korhonen. Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695, 2015.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

Sabine Hunsicker, Chen Yu, and Christian Federmann. Machine learning for hybrid machine translation. In *Proceedings of the seventh workshop on statistical machine translation*, pages 312–316. Association for Computational Linguistics, 2012.

Ignacio Iacobacci, Mohammad Taher Pilehvar, and Roberto Navigli. SensEmbed: Learning sense embeddings for word and relational similarity. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 95–105, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1010. URL `https://www.aclweb.org/anthology/P15-1010`.

Ignacio Iacobacci, Mohammad Taher Pilehvar, and Roberto Navigli. Embeddings for word sense disambiguation: An evaluation study. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 897–907, 2016.

Laurent Jacob, Jean-philippe Vert, and Francis R Bach. Clustered multi-task learning: A convex formulation. In *Advances in neural information processing systems*, pages 745–752, 2009.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33 (1):117–128, 2010.

Hwiyeol Jo and Stanley Jungkyu Choi. Extrofitting: Enriching word representation and its vector space with semantic lexicons. *arXiv preprint arXiv:1804.07946*, 2018.

Thorsten Joachims. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 143–151, 1997.

William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hervé Jégou, and Tomas Mikolov. Fasttext. zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016a.

Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016b.

Douwe Kiela, Felix Hill, and Stephen Clark. Specializing word embeddings for similarity or relatedness. In *Proceedings of EMNLP*, 2015.

Joo-Kyung Kim, Gokhan Tur, Asli Celikyilmaz, Bin Cao, and Ye-Yi Wang. Intent detection using semantically enriched word embeddings. In *2016 IEEE Spoken Language Technology Workshop (SLT)*, pages 414–419. IEEE, 2016.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Gakuto Kurata, Bing Xiang, and Bowen Zhou. Improved neural network-based multi-label classification with better initialization leveraging label co-occurrence. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 521–526, 2016.

Guillaume Lample, Alexis Conneau, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=H196sainb`.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

Ken Lang. Newsweeder: Learning to filter netnews. In *Machine Learning Proceedings 1995*, pages 331–339. Elsevier, 1995.

Hang Le, Loïc Vial, Jibril Frej, Vincent Segonne, Maximin Coavoux, Benjamin Lecouteux, Alexandre Allauzen, Benoît Crabbé, Laurent Besacier, and Didier Schwab. Flaubert: Unsupervised language model pre-training for french. *arXiv preprint arXiv:1912.05372*, 2019.

Rémi Lebret and Ronan Collobert. Word embeddings through hellinger PCA. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 482–490, Gothenburg, Sweden, April 2014. Association for Computational Linguistics. doi: 10.3115/v1/E14-1051. URL `https://www.aclweb.org/anthology/E14-1051`.

Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *In Advances in neural information processing systems*, pages 2177–2185, 2014.

Omer Levy, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. In *Transactions of the Association for Computational Linguistics*, volume 3, pages 211–225, 2015.

Shaoshi Ling, Yangqiu Song, and Dan Roth. Word embeddings with limited memory. In *Association of Computational Linguistics*, 2016.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Huma Lodhi, Grigoris Karakoulas, and John Shawe-Taylor. Boosting strategy for classification. *Intelligent Data Analysis*, 6(2):149–174, 2002.

Kevin Lund and Curt Burgess. Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instrumentation, and Computers*, 28:203–208, 1996.

Minh-Thang Luong, Richard Socher, and Christopher D. Manning. Better word representations with recursive neural networks for morphology. In *CoNLL*, pages 104–113, 2013.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

Louis Martin, Benjamin Muller, Pedro Javier Ortiz Suárez, Yoann Dupont, Laurent Romary, Éric Villemonte de la Clergerie, Djamé Seddah, and Benoît Sagot. Camembert: a tasty french language model. *arXiv preprint arXiv:1911.03894*, 2019.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=r1gs9JgRZ`.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013a.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013b.

George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

George A Miller and Walter G Charles. Contextual correlates of semantic similarity. *Language and cognitive processes*, 6(1):1–28, 1991.

Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

Lili Mou, Ran Jia, Yan Xu, Ge Li, Lu Zhang, and Zhi Jin. Distilling word embeddings: An encoding approach. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1977–1980, 2016.

Jiaqi Mu and Pramod Viswanath. All-but-the-top: Simple and effective post-processing for word representations. In *6th International Conference on Learning Representations, ICLR 2018*, 2018.

Jinseok Nam, Jungi Kim, Eneldo Loza Mencía, Iryna Gurevych, and Johannes Fürnkranz. Large-scale multi-label text classification—revisiting neural networks. In *Joint european conference on machine learning and knowledge discovery in databases*, pages 437–452. Springer, 2014.

Arvind Neelakantan, Jeevan Shankar, Alexandre Passos, and Andrew McCallum. Efficient non-parametric estimation of multiple embeddings per word in vector space. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1059–1069, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1113. URL `https://www.aclweb.org/anthology/D14-1113`.

Mohammad Norouzi, Ali Punjani, and David J Fleet. Fast search in hamming space with multi-index hashing. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3108–3115. IEEE, 2012.

Charles E Osgood. Semantic differential technique in the comparative study of cultures. *American Anthropologist*, 66(3):171–200, 1964.

Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. 2015.

Karl Pearson. On lines and planes of closest fit to systems of points in space. In *Philosophical Magazine*, volume 2, pages 559–572, 1901.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–43, 2014.

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1202. URL `https://www.aclweb.org/anthology/N18-1202`.

Kira Radinsky, Eugene Agichtein, Evgeniy Gabrilovich, and Shaul Markovitch. A word at a time: computing word relatedness using temporal semantic analysis. In *Proceedings of the 20th international conference on World wide web*, pages 337–346. ACM, 2011.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*, 2019.

Diana Ramirez-Cifuentes, Christine Largeron, Julien Tissier, Ana Freire, and Ricardo Baeza-Yates. Enhanced word embeddings for anorexia nervosa detection on social media. *Intelligent Data Analysis*, 2020.

Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. Piscataway, NJ, 2003.

Vikas Raunak, Vivek Gupta, and Florian Metze. Effective dimensionality reduction for word embeddings. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 235–243, Florence, Italy, August 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-4328. URL `https://www.aclweb.org/anthology/W19-4328`.

Douglas L Reilly, Leon N Cooper, and Charles Elbaum. A neural model for category learning. *Biological cybernetics*, 45(1):35–41, 1982.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Herbert Rubenstein and John B Goodenough. Contextual correlates of synonymy. *Communications of the ACM*, 8(10):627–633, 1965.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *RBM*, 500(3):500, 2007.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Mark Schmidt. Least squares optimization with l1-norm regularization. *CS542B Project Report*, 504:195–221, 2005.

Mickaël Seznec, Nicolas Gac, André Ferrari, and François Orieux. A study on convolution using half-precision floating-point numbers on gpu for radio astronomy deconvolution. In *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 170–175. IEEE, 2018.

Dinghan Shen, Qinliang Su, Paidamoyo Chapfuwa, Wenlin Wang, Guoyin Wang, Ricardo Henao, and Lawrence Carin. Nash: Toward end-to-end neural architecture for generative semantic hashing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2041–2050. Association for Computational Linguistics, 2018. URL `http://aclweb.org/anthology/P18-1190`.

Raphael Shu and Hideki Nakayama. Compressing word embeddings via deep compositional code learning. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=BJRZzFlRb`.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

Robert R Sokal. A statistical method for evaluating systematic relationship. *University of Kansas science bulletin*, 28:1409–1438, 1958.

Charles Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904.

Julien Subercaze, Christophe Gravier, and Frederique Laforest. On metric embedding for boosting semantic similarity computations. In *Association of Computational Linguistics*, 2015.

Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014.

Julien Tissier, Christophe Gravier, and Amaury Habrard. Dict2vec: Learning word embeddings using lexical dictionaries. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2017)*, pages 254–263, 2017.

Julien Tissier, Christophe Gravier, and Amaury Habrard. Near-lossless binarization of word embeddings. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, volume 33, pages 7104–7111, 2019. doi: 10.1609/aaai.v33i01.33017104. URL `https://aaai.org/ojs/index.php/AAAI/article/view/4692`.

Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

Peter D. Turney. Domain and function: A dual-space model of semantic relations and compositions. *Journal of Artificial Intelligence Research*, 44:533—-585, 2012.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf`.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language

understanding. In *International Conference on Learning Representations*, 2019a. URL https://openreview.net/forum?id=rJ4km2R5t7.

Alex Wang, Amanpreet Singh, Yada Pruksachatkun, Nikita Nangia, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 3266–3280. Curran Associates, Inc., 2019b.

Wei Wang, Bin Bi, Ming Yan, Chen Wu, Jiangnan Xia, Zuyi Bao, Liwei Peng, and Luo Si. Structbert: Incorporating language structures into pre-training for deep language understanding. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJgQ4lSFPH.

Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1753–1760. Curran Associates, Inc., 2009. URL http://papers.nips.cc/paper/3383-spectral-hashing.pdf.

Chang Xu, Yalong Bai, Jiang Bian, Bin Gao, Gang Wang, Xiaoguang Liu, and Tie-Yan Liu. Rc-net: A general framework for incorporating knowledge into word representations. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1219–1228. ACM, 2014.

Jiaming Xu, Peng Wang, Guanhua Tian, Bo Xu, Jun Zhao, Fangyuan Wang, and Hongwei Hao. Convolutional neural networks for text hashing. In *IJCAI*, pages 1369–1375, 2015.

Dongqiang Yang and David MW Powers. *Verb similarity on the taxonomy of WordNet*. Masaryk University, 2006.

Jun Yin, Xin Jiang, Zhengdong Lu, Lifeng Shang, Hang Li, and Xiaoming Li. Neural generative question answering. *arXiv preprint arXiv:1512.01337*, 2015.

Zi Yin and Yuanyuan Shen. On the dimensionality of word embedding. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 887–898. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7368-on-the-dimensionality-of-word-embedding.pdf.

Mo Yu and Mark Dredze. Improving lexical embeddings with semantic knowledge. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 545–550, Baltimore, Maryland, June 2014. Association for Computational Linguistics. doi: 10.3115/v1/P14-2089. URL https://www.aclweb.org/anthology/P14-2089.

Wen Zhang, Taketoshi Yoshida, and Xijin Tang. A comparative study of tf* idf, lsi and multi-words for text classification. *Expert Systems with Applications*, 38(3):2758–2765, 2011.

Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

Will Y Zou, Richard Socher, Daniel Cer, and Christopher D Manning. Bilingual word embeddings for phrase-based machine translation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1393–1398, 2013.