SCIENCES SORBONNE UNIVERSITÉ

Doctoral Thesis

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité **Informatique**
École Doctorale Informatique, Télécommunications et Électronique (Paris)

Neural Architecture Search under Budget Constraints

Recherche d'Architectures de Réseaux de Neurones sous Contraintes de Budget

---

by

Tom Veniat

A thesis submitted in fulfilment of
the requirements for the degree of
PhD in Computer Science

Defended on 01/07/2021

Jury composed of :

Mr. Francois Fleuret	*University of Geneva*	Reporter
Mr. Jakob Verbeek	*Facebook*	Reporter
Mr. Patrick Gallinari	*Sorbonne University*	Examiner
Mrs. Raia HadsXell	*DeepMind*	Examiner
Mr. Vincenzo Lomonaco	*University of Pisa*	Examiner
Mr. Marc'Aurelio Ranzato	*Facebook*	Co-supervisor
Mr. Ludovic Denoyer	*Facebook*	Supervisor

ABSTRACT

The recent increase in computation power and the ever-growing amount of data available ignited the rise in popularity of deep learning algorithms. In addition to being strong candidates to solve new problems, these algorithms are often able to obtain improved performances over existing approaches, reaching or even beating human-level abilities in numerous domains. However, the expertise, the amount of data, and the computing power necessary to build such algorithms as well as the memory footprint and the inference latency of the resulting system are all obstacles preventing the use of these methods by a larger user base and on certain applications. In this thesis, we propose several methods allowing to make a step towards a more efficient and automated procedure to build deep learning models.

First, we focus on learning an efficient architecture for the standard image classification and segmentation problems. We propose a new model in which users can guide the architecture learning procedure by specifying a fixed budget and cost function. The training procedure then automatically learns a model and its architecture by jointly optimizing the predictive performance and the user-specified black-box cost function to fit the constraint. Then, we move away from the static setting to consider the problem of sequence classification, where a model can be even more computationally efficient by dynamically adapting the size of the model to the complexity of the signal to come. We show that both approaches result in significant budget savings on a range of cost functions and classes of models.

Finally, we tackle the efficiency problem through the lens of transfer learning. Arguing that a learning procedure can be made even more efficient if, instead of starting *tabula rasa*, it builds on knowledge acquired during previous experiences. We explore modular architectures in the continual learning scenario, where a model faces a sequence of tasks having different degrees of relatedness. In this context, solving a new problem reduces to finding the correct combination of pre-trained modules, representing already acquired skills, and new modules, representing the ability to adapt these skills to the task at hand. We present a new benchmark allowing a fine-grained evaluation of different kinds of transfer and show that the proposed modular approach is able to consistently beat existing approaches on most of these dimensions.

RÉSUMÉ

La récente augmentation de la puissance de calcul et la croissance de la quantité de données disponibles sont à l'origine de la montée en popularité des algorithmes d'apprentissage profond. En plus d'être de bons candidats pour résoudre de nouveaux problèmes, ces algorithmes sont souvent capables d'obtenir des performances améliorées par rapport aux approches existantes, atteignant ou même surpassant les capacités humaines dans de nombreux domaines. Cependant, l'expertise, la quantité de données et la puissance de calcul nécessaires pour construire de tels algorithmes ainsi que l'empreinte mémoire et la latence en déploiement sont autant d'obstacles empêchant l'utilisation de ces méthodes par plus d'utilisateurs et sur certaines applications. Dans cette thèse, nous proposons plusieurs méthodes permettant d'approcher une procédure plus efficace et automatisée pour construire des modèles de deep learning.

Tout d'abord, nous nous concentrons sur l'apprentissage d'architectures efficaces pour les problèmes de classification et de segmentation d'images. Nous proposons un nouveau modèle dans lequel les utilisateurs peuvent guider la procédure d'apprentissage de l'architecture en spécifiant un budget et une fonction de coût. La procédure d'apprentissage apprend ensuite automatiquement un modèle et son architecture en optimisant conjointement les performances prédictives et la fonction de coût spécifiée par l'utilisateur. Ensuite, on s'éloigne du cadre statique pour considérer le problème de la classification de séquences, dans lequel un modèle peut être plus efficace en adaptant dynamiquement la taille du modèle à la complexité du signal à venir. Nous montrons que les deux approches se traduisent par des économies de budget significatives sur une gamme de fonctions de coût et de classes de modèles.

Enfin, nous abordons le problème de l'efficacité à travers le prisme de l'apprentissage par transfert. Une procédure d'apprentissage peut être rendue encore plus efficace si, au lieu de démarrer *tabula rasa*, elle s'appuie sur les connaissances acquises lors d'expériences précédentes. Nous explorons les architectures modulaires dans le scénario d'apprentissage continu, où un modèle fait face à une séquence de tâches ayant différents degrés de relation. Dans ce contexte, résoudre un nouveau problème se réduit à trouver la bonne combinaison de modules préentraînés, représentant des compétences déjà acquises, et de nouveaux modules, représentant la capacité d'adapter ces compétences à la tâche à accomplir. Nous présentons un nouveau protocole d'évaluation permettant une analyse fine des différents types de transfert et montrons que l'approche modulaire proposée est capable de battre les approches existantes sur la plupart de ces dimensions.

REMERCIEMENTS

J'aimerais remercier toutes les personnes qui m'ont soutenu pendant la réalisation de cette thèse et qui ont rendu cette expérience aussi enrichissante.

Tout d'abord, merci à Ludovic et Marc'Aurelio pour leur accompagnement tant scientifique que personnel, merci de m'avoir fait confiance et permis de travailler sur les sujets qui m'intéressent.

Je souhaite également remercier tous les membres du jury, François Fleuret, Jakob Verbeek, Patrick Gallinari, Raia Hadsell et Vincenzo Lomonaco pour le temps qu'ils ont accordé à mon travail. Merci pour vos retours sur mon manuscrit ainsi que votre attention et vos questions lors de la soutenance.

Merci à l'ensemble de l'équipe MLIA, permanents, anciens doctorants et doctorants actuels pour m'avoir permis de passer ces 3 années dans un environnement aussi stimulant. Merci aussi à Christophe et Nadine, pour leur soutien technique et administratif, mais surtout personnel et sportif.

Un merci spécial à Sébastien Mosser et Frédéric Precioso, qui ont été présents lorsque j'en avais besoin, qui m'ont aidé à prendre la décision de faire cette thèse et m'ont guidé pour me permettre de la faire dans de si bonnes conditions. Merci aussi à tous les amis de Savoie, de Nice, de Rouen et de Paris pour vos encouragements tant pendant la thèse qu'avant.

Enfin, un immense merci à mes parents, Joëlle et Alain, mon frère, Bertrand et ma copine, Apolline pour votre soutien sans faille non seulement pendant la réalisation de cette thèse, mais surtout tout au long du chemin qui m'a permis d'y arriver. Tout ceci aurait été impossible sans votre confiance et votre soutien inconditionnel durant toutes ces années.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AI	Artificial Intelligence
ConvNet	Convolutional Neural Network
DL	Deep Learning
DNN	Deep Neural Network
ML	Machine Learning
RL	Reinforcement Learning
MLP	Multi-Layer Perceptron
NN	Neural Network
ANN	Artificial Neural Network
RNN	Recurrent Neural Network
GRU	Gated Recurrent Unit
NAS	Neural Architecture Search
CL	Continual Learning
BSN	Budgeted Super Networks
SANAS	Stochastic Adaptive Architecture Search
MNTDP	Modular Networks with Task Driven Prior
MNTDP-S	MNTDP-Stochastic
MNTDP-D	MNTDP-Deteministic
EWC	Elastic Weight Consolidation
ER	Experience Replay
HAT	Hard Attention to Tasks
DEN	Dynamically Expandable Networks
RCL	Reinforced Continual Learning
PNN	Progressive Neural Networks
CTrL	Continual Transfer Learning
F-MNIST	Fashion MNIST
R-MNIST	Rainbow MNIST
DTD	Describable Textures Dataset
SVHN	Street View House Number

KNN	K-Nearest Neighbors
MFCC	Mel Frequency Cepstral Coefficient
DCT	Discrete Cosine Transform
FLOP	Floating Point Operation

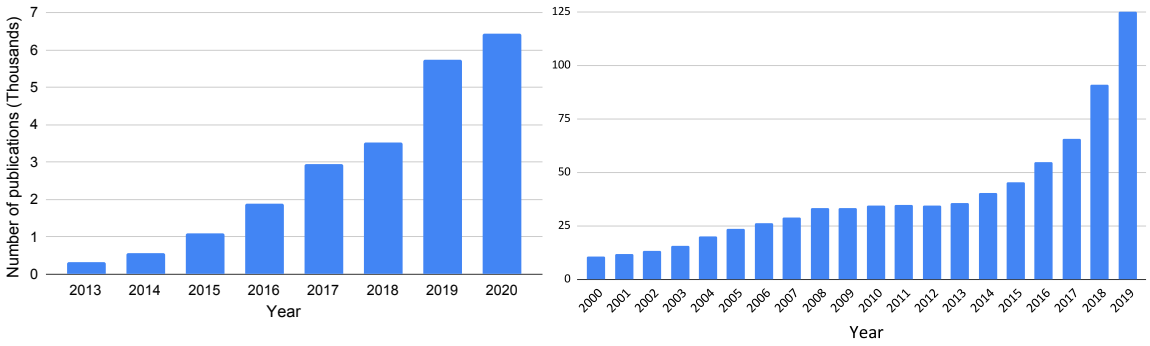
CHAPTER 1

INTRODUCTION

Contents

1.1 Context

Artificial Intelligence (AI) is commonly defined as the ability of machines to exhibit human or animal-like intelligence. While the idea of artificial beings showing signs of intelligence can be traced back to centuries in myths and legends, the first concrete steps in this direction were made in the 1950s. First, Alan Turing proposed the Turing test, allowing the evaluation of an “Intelligent“ system by asking the question, "Is it able to show intelligent behavior?" Then, John McCarthy coined the term Artificial Intelligence during the organization of the Dartmouth workshop on AI in 1956. This workshop is considered one of the founding events of the field since many of its attendees would continue making significant contributions to the field over the next decades. Since then, the term AI is also used to characterize all the techniques that humans have created through decades of research to build and train machines capable of performing tasks then impossible to automate.
Machine Learning (ML) is the branch of AI focused on this training. The general objective is to automatically extract knowledge or learn how to perform specific tasks from data. One way of doing that is through the use of Artificial Neural Networks (ANNs), a model loosely related to how our brain works and able to learn to map features given as inputs to the desired output. Recently, a subfield of ML called Deep Learning (DL) propose to learn this input/output mapping not directly using the given features but by going through intermediate representations, with the particularity that these successive representations are themselves automatically learned from data. The resulting system composed of this succession of Neural Network layers each feeding its output to another layer deeper in



(a) Number of Deep Learning-related publications on Arxiv, 2013-2020.

(b) Number of peer-reviewed Artificial Intelligence-related publications, 2000-2019.

Figure 1.1. – Trends in AI research, see Zhang et al. (2021) for more details.

the chain is called a Deep Neural Network (DNN).
One of the major strengths of DL, explaining in a great part the renewed interest in the field, is its ability to scale. While most of the basic concepts used in DL have been discovered decades ago, its ability to benefit from modern advances in hardware and to leverage the ever-increasing amount of data available allows this number of successive representations to increase, resulting in deeper and more powerful DNNs able to solve harder problems. This technical progress ignited a new wave of AI research projects, resulting in a dramatic increase in the number of new contributions, all proposing new architectures, new kinds of layers, or new training procedures – see Figure 1.1. These contributions allowed the community to make huge leaps first in computer vision (Krizhevsky et al., 2012; He et al., 2016; Huang et al., 2017), quickly followed by speech processing (Amodei et al., 2016; Baevski et al., 2020), natural language processing, reasoning, and numerous other domains. This progress on research problems also had a positive impact on a large number of concrete applications able to benefit from the recent advances in AI such as medicine, physics, biology, robotics or art.

1.2 Challenges

Yet, with the growing popularity of DL, it is becoming increasingly difficult to find the optimal components to tackle a new problem. Both because of the growing number of publications, making the space of possible solutions ever larger, and the pace at which the field is advancing, as the current state-of-the-art could be replaced by better performing approaches within a few months. The principal consequence is that each new project based on Deep Learning (DL) requires a lot

of resources. The two main being the expert knowledge necessary to build, tune and update high-performing architectures, the second being computing power, as the process of building and deploying these architectures is very energy-hungry. Indeed, a DNN being a complex combination of different types of layers, finding the correct arrangement of layers and the corresponding parameters requires expertise and time. Since the standard procedure is to try a solution, evaluate it and use the results to see what can be improved, the first part of the problem we tackle in this thesis is to explore to what degree this iterative procedure can be automatized using DL algorithms themselves. Such automation would not only free up time for experts to work on more advanced problems but may also improve the final performance of the system, as an automatically learned search procedure could be more efficient than a hand-made one. The final major benefit of such algorithms is that they would also allow experts from other domains to build systems suiting their own needs and, more generally, greatly reduce the entry cost of modern AI. The second issue we tackle has to do with the energy consumption of the modern DL pipeline, arising at two different steps of the life cycle of such systems:

Training In addition to the hours of human time spent on tuning the system, the training procedure of a single DNN can by itself can be very expensive. We can take the extreme example of GPT-3 (Brown et al., 2020), the latest language model [1] developed by Open AI. While being a technical feat and reaching impressive performance, being closer to pass the Turing test than any other machine, a single training run can be estimated to millions of dollars and 10s of millions if taking into account the iterative trial and errors necessary to its development. The resulting system requires hundreds of GB just in storage and is nowhere close to being usable on standard computers, let alone connected devices like our smartphone. The same observation can be made for other widely publicized contributions such as DeepMind's AlphaGo (Silver et al., 2016), the first algorithm able to beat a world champion in the game of go, and its more recent version AlphaZero (Silver et al., 2017) able to outperform all other algorithms on the games of go, chess and shogi.

Deployment Once trained, the purpose of a DNN is to be used on real-world problems, where the raw predictive performance is not the only measure that matters. Taking the example of a conversational agent for mobile devices, care should be taken to ensure that the model can work in low-resource environments. Large models should therefore be avoided for this application as they tend to be slower to evaluate and more energy-hungry. In order to find a model able to run

1. A DNN trained to predict the next word in a sentence.

close to real-time and that would not drain the battery of the device, the standard route is to highly restrict the DNN architectures that will be tried and/or to post-process the trained model to compress it. Both solutions are very time-consuming and harm the final performance of the deployed model.

1.3 Contributions and Thesis Outline

We propose in this thesis to tackle the problem of using Neural Networks to learn Neural Network Architectures, also referred to as Neural Architecture Search (NAS). We present in the next chapters different NAS algorithms allowing us to incorporate a budget constraint to the search procedure, automatically discovering architectures reaching higher performances than handmade ones and better suited for deployment. We explore three different settings of increasing complexity in which NAS can be leveraged:

- In Chapter 2, we present general background relevant to the entire thesis. A short background section is added to each chapter containing specific information only relevant to this part. In the general background, we cover algorithms commonly used to help the tuning of neural network architectures. We first present simple search algorithms based on heuristics and then more recent approaches based on ML. In this second category, we start with the standard approaches such as Bayesian Optimization and Evolutionary Algorithms before presenting in more depth the recent approaches based on Deep Learning, using combinations of Reinforcement Learning (RL) and specific architecture components. Since one of the objectives of this thesis is to learn more efficient Neural Networks, we present some background approaches on model compression in the second part of this chapter. The three main categories we present are quantization, pruning, and distillation.

- In Chapter 3, we focus on the NAS problem in a setting where the model used at inference must respect some user-defined budget We formulate this issue as a problem of automatically learning a neural network architecture under budget constraints. To tackle this problem, we propose a *budgeted learning* approach that integrates a maximum cost directly in the learning objective function. The main originality of our approach with respect to state-of-the-art is the fact that it can be used with any type of costs, existing methods being usually specific to particular constraints like inference speed or memory consumption – see Chapter 2 for a review of the state-of-the-art. In our case, we investigate the ability of our method to deal with three different costs: (i) the *computation cost* reflecting the inference speed of the resulting

model, (ii) *the memory consumption cost* that measures the final size of the model, and the (iii) *distributed computation cost* that measures the inference speed when computations are distributed over multiple machines or processors. This work led to the following publication: Tom Veniat and Ludovic Denoyer (2018). "Learning Time/Memory-Efficient Deep Architectures With Budgeted Super Networks". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, pp. 3492–3500. DOI: 10.1109/CVPR.2018.00368. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Veniat%5C_Learning%5C_Time Memory-Efficient%5C_Deep%5C_CVPR%5C_2018%5C_paper.html.

- In Chapter 4, we propose to apply NAS to temporal problems. While the data samples experienced by the model in Chapter 3 are independent and identically distributed (i.i.d), the problem on which we focus here doesn't have this property. In this setting, some patterns giving information on what will happen next can appear in the data. If a model can identify these patterns in the input distribution, it can reduce its computation budget further than any model making the i.i.d assumption. Indeed, by predicting when examples will be simpler, it can save time on those and spend it later, when it anticipates more challenging samples. To solve this problem, we propose a NAS approach able to adapt the architecture on-the-fly during a sequence, where saving (resp. spending) time reduces to selecting small (resp. large) architectures. This work led to the following publication: Tom Véniat, Olivier Schwander, and Ludovic Denoyer (2019). "Stochastic Adaptive Neural Architecture Search for Keyword Spotting". In: *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2019, Brighton, United Kingdom, May 12-17, 2019*. IEEE, pp. 2842–2846. DOI: 10.1109/ICASSP.2019.8683305. URL: https://doi.org/10.1109/ICASSP.2019.8683305.

- In Chapter 5, we propose to use NAS on dynamic tasks, where the problem to solve changes over time. This requires an ability of the search procedure to update a model to new tasks while, if possible, making use of the knowledge acquired on previous problems to greatly speed up the training procedure. This also implies that the prior knowledge to build upon should be retained, at least in part, from one experience to the next. To this end, we place ourselves in the Continual Learning (CL) framework which bears some similarities with this setting. We first pinpoint general properties that a good CL learner should have, confirming that NAS is a good candidate approach. Then we propose both a Lifelong Neural Architecture Search algorithm and a new benchmark, specifically tailored to assess the amount of knowledge retained and shared across tasks having different degrees of relatedness.

This work led to the following publication: Tom Veniat, Ludovic Denoyer, and Marc'Aurelio Ranzato (2021). "Efficient Continual Learning with Modular Networks and Task-Driven Priors". In: *9th International Conference on Learning Representations, ICLR 2021* abs/2012.12631. arXiv: 2012.12631. URL: https://arxiv.org/abs/2012.12631.

- In Chapter 6, we conclude with a summary of the contributions and propose some research perspectives.

Chapter 2

NEURAL ARCHITECTURE SEARCH AND BUDGETED DEEP LEARNING

Contents

In this chapter, we present the general problem of model definition in Machine Learning (ML), with a focus on the case of Neural Networks (NNs). We first introduce a broad set of methods often used to tune NN architectures in Section 2.1. Then we present several propositions to use Deep Learning (DL) methods to tackle this problem in Section 2.2, replacing the hand-tuned architectures and simple algorithms with a training procedure able to learn powerful architectures. This line of work is referenced to as Neural Architecture Search (NAS). Finally, we present existing work focused on obtained efficient models in Section 2.3.

In the general ML pipeline, depicted in Figure 2.1, the model definition step corresponds to the selection of a class of algorithms (e.g. Random Forests, Support Vector Machines, K-Nearest Neighbors, ...) and the selection of the hyper-parameters of a specific model that will be trained. For instance, if the selected class is NNs, the number of layers, the connection between them, and the number of neurons per layer are examples of hyper-parameters to select before training. The procedure to select these hyper-parameters is done following a predetermined procedure represented by the green arrow in Figure 2.1, and usually guided by the performance of the constructed model once trained.

While some of the methods we present in this section can be used to tune the hyper-parameters of both the model architecture and the training procedure, we focus here on the former as it is the main subject of the thesis. Nevertheless, it is important to note that some meta-learning approaches also tackle the problem of

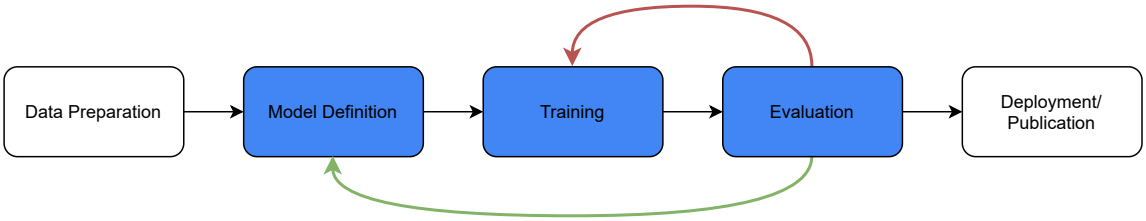


Figure 2.1. – Schematic representation of the standard ML pipeline. The blue blocs correspond to steps usually involved in the hyper-parameter selection process. Model definition corresponds to the selection of a class of algorithms as well as the corresponding parameters (e.g., if the class is NNs, depth and width are examples of such parameters). The tuning process can be split into two parts: model tuning (Green arrow) and training procedure tuning (red arrow), both processes being guided by the evaluation metrics.

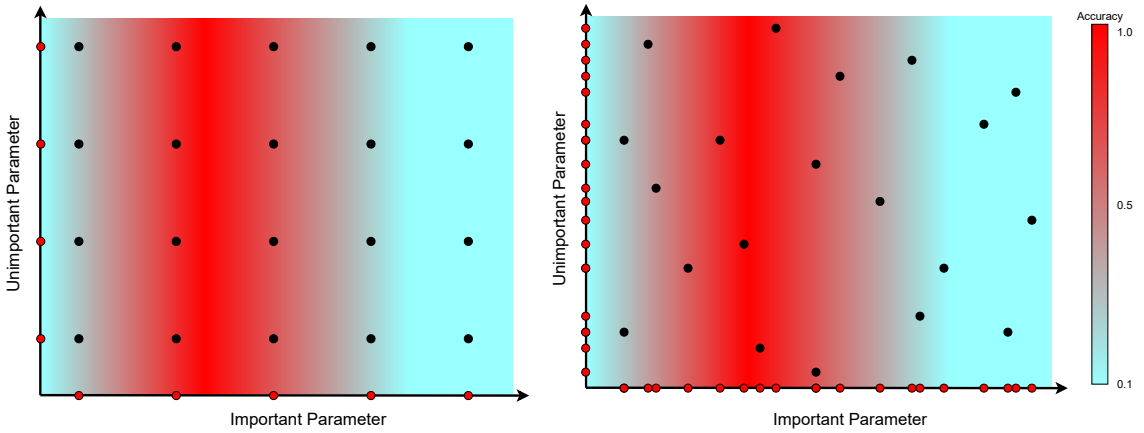
incorporating the tuning of the training procedure into the training itself (Thrun and Pratt, 1998; Hochreiter et al., 2001; Andrychowicz et al., 2016; Finn et al., 2017a; Li et al., 2017b; Nichol et al., 2018a).

2.1 Automatic Hyper-Parameter Tuning

The art of hand-tuning the hyper-parameters of a NN is a skillful practice involving rules of thumb, experience, intuition, and a lot of patience. Over the years, several approaches have been proposed to streamline the process. We present the most common of them in this section.

2.1.1 Simple search algorithms

Grid Search The simplest algorithm to tune hyper-parameters is the grid search. In this approach, the user defines a set of values to try for each parameter and train a model to convergence for each possible combination. The validation set performance is then used to select the best combination. The advantages of this algorithm are that it is simple, easy to implement, and highly parallelizable since the trials are independent. The main downside is its inefficiency, as the number of combinations to try grows exponentially w.r.t. the number of hyperparameters to tune. While it can be a good option for small networks with simple training procedures, other approaches are generally used for recent models.



(a) Grid Search (b) Random Search

Figure 2.2. – Toy comparison of grid search and random search with 20 trials on a 2D space. We observe that random search (b) results in a better coverage of the space than the grid search (a) for the same number of trials, especially when some parameters are more important than others.

Random Search In the random search, the parameters are drawn from user-defined distributions instead of selected from a fixed set of possible values. It has the same advantages as the grid search and is in general favored over it because it offers better coverage of the hyper-parameters space as illustrated in Figure 2.2, resulting in higher performances.

To better exploit the structure of the problem at hand, one can use these search algorithms in an iterative way, each step allowing to identify regions of the space that will be searched over using a finer scale in the next iteration. Other algorithms can also be used to schedule the search more efficiently. For example, some algorithms early-stops less promising trials to free up resources for other combinations (Jamieson and Talwalkar, 2016; Li et al., 2017a; Li et al., 2020) or combine the hyper-parameters of different trials during training to allow the search of schedules where some parameters changes over time (Jaderberg et al., 2017).

2.1.2 Bayesian Optimization

As a global optimization method of black-box functions, Bayesian Optimization is a good candidate approach for optimizing the hyperparameters of a NN. For example, Snoek et al. (2012) proposes to see the generalization performance of a model as a sample from a Gaussian Process. The search algorithm alternates the usage of the acquisition function to select where to evaluate the function

(an evaluation corresponding to the construction and training a model) and the update of the prior distribution using the new data (validation performance of the trained model). This method achieved state-of-the-art performance on Cifar-10 Krizhevsky (2009a) by tuning the hyper-parameters of AlexNet Krizhevsky et al. (2012). While showing a better sample efficiency than the random search, this method is less commonly used due to its iterative nature, preventing a massive parallelization of the search procedure.

2.2 Neural Architecture Search

The main idea of NAS is to move the conception of a specific architecture from the Model Definition component of Figure 2.1 to the training itself. Different authors have proposed to provide networks with the ability to learn to select the computations that will be applied i.e choosing the right architecture for a particular task/sample. This is the case for example in Denoyer and Gallinari (2014) where the path that a sample will follow is dynamically sampled during inference or in Srivastava et al. (2015) based on gating mechanisms.

Different algorithms have been proposed to achieve this goal. In general, a NAS algorithm has two main components. Those components are the search space, which defines at the same which architectures can be learned by an approach and how the architectures are represented during the search procedure. The second component is the search strategy itself, corresponding to how the search-space will be explored.

2.2.1 Search Space

Sequential representation One of the first kind of approaches (Zoph and Le, 2017; Zoph et al., 2018) use a sequence of tokens to represent an architecture. In Zoph and Le (2017) each token represents one hyper-parameter of a layer. It allows for example to represent any feed-forward N-layers Convolutional Neural Networks (ConvNets) as a sequence of size 5N, each group of 5 items representing respectively the kernel height, kernel width, stride height, stride width, and number of filters. They also propose different structures of the generated sequence to allow skip-connections and to represent the recurrent cell of a Recurrent Neural Network (RNN). They propose to see each token as an action and train an agent to generate the best possible sequence of actions to obtain a high-performing architecture.

Similarly, Baker et al. (2017) propose a search space in which each action corresponds to a full layer, resulting in a larger set of actions available at each step but a more compact architecture representation. They also add restrictions to the search space by removing some actions to reduce the search space size and make learning tractable.

Cell representation Other representations emerged based on the insights provided by high-performing hand-crafted architectures such as the ResNet (He et al., 2016) or the DenseNet (Huang et al., 2017). For example, Zoph et al. (2018), Real et al. (2019), Liu et al. (2018a), Liu et al. (2018b), and Zhong et al. (2018) propose to represent a network as a sequence of two kinds of cells: *normal cells*, that keeps the same resolution between the inputs and the outputs and *reduction cells* that reduce the scale of the feature maps (and generally increase the number of maps). The learned cell can then be stacked to obtain the desired depth.

Sub-Graph/Super Networks Another popular representation of the search space is in the form of a large computational graph encompassing all possible architectures (Fernando et al., 2017; Liu et al., 2019; Pham et al., 2018; Wu et al., 2019). The objective of the search procedure is then to find the best possible sub-graph by pruning large parts of the search space. For example Pham et al. (2018) uses this approach to learn high-performing architectures on the Cifar10 image classification dataset and obtain state-of-the-art performance on the Penn Treebank benchmark for language models using a learned RNN cell. Liu et al. (2019) propose to use several NN modules in parallel at each layer and to learn which one to keep to obtain the best possible performance, this approach can also be represented as a large graph in which we want to remove connections. This Super Network method is more thoroughly presented in Chapter 3 as it is the search space we will use in this work.

2.2.2 Search Strategy

From the NN point of view, this problem is often viewed as a network architecture discovery problem and solved with Neural Architecture Search (NAS) methods in which the search is guided by a trade-off between prediction quality and prediction cost (Huang et al., 2018; Gordon et al., 2018). While these models often rely on expensive training procedures where multiple architectures are trained, some recent works have proposed to simultaneously discover the architecture of the network while learning its parameters resulting in models that are fast both at training and at inference time.

Reinforcement Learning Zoph and Le (2017) and Zoph et al. (2018) use a RNN called controller to generate the sequence of tokens representing the architecture. Each token is sampled from the distribution given by the controller and can be interpreted as an action. Once the controller generated a full sequence, the corresponding architecture is trained and can be evaluated. The validation set accuracy after training is used as the reward signal that the controller will receive for this sequence of actions. REINFORCE (Williams, 1992), which is presented in more details in Section 3.2, is used to update the controller before generating new architectures. Pham et al. (2018) also use Policy Gradient with the validation accuracy as the reward to find the optimal sub-graph of the large Super Network.

Baker et al. (2017) also use the validation accuracy of a trained architecture as the reward signal. Instead of using REINFORCE, they use Q-learning with an agent following an ϵ-greedy strategy to learn the architectures. They also make use of experience replay and cache the accuracy of trained models to make the training procedure more efficient. Zhong et al. (2018) use Q-learning and an ϵ-greedy exploration strategy to generate the structure of a cell that will be stacked to generate the architecture of the final model.

Proxy tasks Since Reinforcement Learning (RL)-approaches are expensive to train, Zoph et al. (2018) propose to learn the architecture on an easier proxy task. Once the training identified high-performing cells on the Cifar10 (Krizhevsky, 2009a) proxy task, a larger number of them can be stacked to scale up the architecture before training it on the task of interest, Imagenet (Deng et al., 2009).

Hyper-networks Another line of work takes inspiration from the hyper networks introduced in Ha et al. (2017): Instead of training the "child" architecture using standard gradient descent techniques, these methods use a second component (the first one being the architecture selector) to directly generate the weights of the selected architecture (Brock et al., 2018). The resulting model is then directly evaluated on a batch of training data and back-propagated through to update the architecture and weights generators.

Evolutionary Algorithms After observing the amount of computation required to obtain good performances when using the RL approach proposed in Zoph and Le (2017) and Zoph et al. (2018), Real et al. (2017) propose to explore how well simple evolutionary technique would perform when also given a large computational budget. Their approach is able to reach similar performances on Cifar10 and outperform other NAS-based approaches on Cifar100. More recently, Real et al. (2019) introduce a new tournament selection procedure which evolves architectures able to beat the state of the art on ImageNet (Deng et al., 2009) and

show that evolutionary algorithms are more efficient than RL-based ones using the same hardware.

Liu et al. (2018b) also reaches high accuracy in both Cifar and ImageNet tasks by hierarchically evolving motifs that can be reused later as building blocks in higher-level motifs.

Joint Training/Weight Sharing To obtain more efficient search procedures, some approaches propose to jointly learn the model architecture and its corresponding parameters. This one-shot approach allows to greatly reduce the cost of the search procedure, as it removes the training of several child networks from the procedure.

For example, Pham et al. (2018) uses the SuperNet approach and proposes to share the parameters of a layer across all the architecture that uses it, removing the need to train it from scratch each time it is sampled. Wu et al. (2019) propose a similar approach but uses the Gumbel-Softmax trick instead of REINFORCE to back-propagate the gradient w.r.t. the architecture selector parameters.

2.3 Budgeted Learning

Most of the early NAS algorithms are only guided by the final predictive performance of the network such as its accuracy on a held-out set. While showing great performances, we focus in this thesis on some problems where this performance is not the only thing that matters. Some problems come with specific constraints that must be taken into account in the design of the architecture. In this section, we present some work that tackles this problem. As in architecture creation, most of the existing approaches are done by hand either *a priori*, by training an efficient architecture specifically designed to respect the desired restriction from the beginning, or *a posteriori*, where the model is first trained and then compressed to fit the constraints. We will first present existing approaches in both categories before moving on to *end-to-end* approaches where the model is shaped to adapt to the constraint during training, all methods proposed in this thesis belonging to the latter category.

Efficient architectures Architecture improvements have been widely used in CNN to improve the cost efficiency of network components, some examples are the bottleneck units in the ResNet model (He et al., 2016), the use of depth-wise separable convolution in Xception (Chollet, 2017) and the lightweight MobileNets (Howard et al., 2017) or the combination of point-wise group convolution and channel shuffle in ShuffleNet (Zhang et al., 2018).

This approach is powerful but requires strong expertise to find the best optimization and is also highly specific to each architecture and/or hardware. Furthermore, most NAS algorithms making no assumption on the architecture of the base network or the components of its search space, these highly efficient computational blocks can be combined with NAS to further improve the computational efficiency of the learned architectures.

Model Compression One of the first approaches to obtain an efficient model is by pruning some connections in an already trained NN. The Optimal Brain Damage (LeCun et al., 1989) and Optimal Brain Surgeon (Hassibi and Stork, 1992) use the Hessian of the loss function on a fully trained NN to approximate the contribution of each weight and identify the less important ones that can be removed without hurting the performance.

The problem of network compression can also be seen as a way to speed up a particular architecture, for example by using quantization of the weights of the network (Vanhoucke et al., 2011), or by combining pruning and quantization (Han et al., 2015). Other algorithms include the use of hardware efficient operations that allow a high speedup (Devlin, 2017).

End-to-end approaches The first example of end-to-end approaches is the usage of quantization at training time: different authors trained models using binary weight quantization coupled with full precision arithmetic operations (Courbariaux et al., 2015; Lu, 2017). More recently, Micikevicius et al. (2017) proposed a method using half-precision floating numbers during training. Another line of work (Zhu et al., 2017; Nan and Saligrama, 2017) proposes to leverage knowledge distillation, a concept initially introduced by Hinton et al. (2015) and Romero et al. (2014). In knowledge distillation, a *student network* is trained to imitate the outputs of a *teacher network* already able to solve the task at hand. This method can be used to compress knowledge initially acquired by a large teacher network into a much smaller student. Another benefit of this approach is that the distillation procedure can be more efficient than the initial training of the teacher network: the probability distribution over labels provided by the teacher implicitly defines a similarity metric over the classes (e.g. confusing a bus with a truck is a less serious mistake than confusing a bus with a flower), an information absent from the initial supervision and often referred to as *dark knowledge*.

Other approaches propose to learn a dynamic network that is trained end-to-end to conditionally select the modules to make the inference more efficient (Bolukbasi et al., 2017; Bengio et al., 2015; McGill and Perona, 2017; Huang et al., 2018; Gordon et al., 2018).

Finally, some methods build on top of the techniques described in Section 2.2, using NAS to directly learn the efficient architecture (Huang and Wang, 2018; Wu et al., 2019; Cai et al., 2019) .

CHAPTER 3

NEURAL ARCHITECTURE SEARCH FOR BUDGETED COMPUTER VISION

Contents

Chapter abstract

In this chapter, we propose to focus on the problem of discovering neural network architectures efficient in terms of both prediction quality and cost. For instance, our approach can solve the following tasks: learn a neural network able to predict well in less than 100 milliseconds or learn an efficient model that fits in a 50 Mb memory. Our contribution is a novel family of models called Budgeted Super Networks (BSN). They are learned using gradient descent techniques applied on a budgeted learning objective function which integrates a maximum authorized cost while making no assumption on the nature of this

cost. We present a set of experiments on computer vision problems and analyze the ability of our technique to deal with three different costs: the computation cost, the memory consumption cost, and a distributed computation cost. We particularly show that our model can discover neural network architectures that have better accuracy than the ResNet and Convolutional Neural Fabrics architectures on CIFAR-10 and CIFAR-100, at a lower cost.

The work in this chapter has led to the publication of a conference paper:

- Tom Veniat and Ludovic Denoyer (2018). "Learning Time/Memory-Efficient Deep Architectures With Budgeted Super Networks". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, pp. 3492–3500. DOI: 10.1109/CVPR.2018.00368. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Veniat%5C_Learning%5C_TimeMemory-Efficient%5C_Deep%5C_CVPR%5C_2018%5C_paper.html.

Resources to reproduce the work in this chapter are publicly available:

- Source code of the experiments: https://github.com/TomVeniat/BSN.

3.1 Introduction

In this section, we focus on the problem of discovering neural network architectures specific to a given problem. To better fit real-world constraints and guide the search procedure, the discovered architectures should not only be able to reach high prediction quality but should also be efficient. Both prediction quality and efficiency can be measured in different ways. As commonly done with classification problems, we chose in this chapter to evaluate the prediction quality using the test set accuracy. The efficiency is measured using different cost functions, each of them reflecting some realistic constraints one could want to apply to its model. For example, one could want to find the best predictor possible while still being able to embed it on a device having only 100Mb of available memory or to use it for a real-time application where it is mandatory to obtain the prediction in less than 100 milliseconds.

Since using this kind of non-differentiable metrics can't directly be optimized using standard stochastic gradient descent algorithms, we start this chapter with a quick overview of *Reinforcement Learning* in §3.2 and more specifically on the *Policy Gradient* algorithm and its variants that will be required later in this chapter and throughout the thesis.

The proposed approach, named *Budgeted Super Networks* (BSN), is the first to introduce *Super Networks* for architecture search. It takes as input a search space in the form of a large Neural Network, a cost function, and the maximum allowed budget. It returns the best path and the corresponding parameters found in the search space able to solve the problem while respecting the specified budget. The Super Networks are presented in more details in §3.3. Since finding the best path with the optimal parameters is an intractable combinatorial problem when dealing with large search spaces, we relax the optimization problem (section 3.4) and propose a stochastic model able to solve it using a combination of standard gradient-based optimization methods and reinforcement learning. This model, called *Stochastic Super Networks*, is presented in details in §3.4.2.

To assess the performance and the flexibility of the proposed approach, we thoroughly evaluate it on various computer vision tasks, using different Super Networks taking inspiration from the *Residual Networks* (He et al., 2016) and *Convolutional Neural Fabrics* (Saxena and Verbeek, 2016) models as search spaces, and with several budget constraints. The results presented in 3.5 show that the different versions of the Stochastic Super Networks model are able to consistently outperform strong baselines while running on a lower budget at inference time.

3.2 Reinforcement Learning background

In the standard Reinforcement Learning (RL) setting, an agent interacts with an environment using a sequence of *actions*. Each action a_t has an impact on the environment *state* and generates a *reward* r_t. Depending on the problem, the environment can be fully or partially observable, giving at timestep t either the full state s_t or a partial observation o_t to the agent. A sequence of states and actions is called a trajectory $\tau = (s_0, a_0, s_1, a_1, ...)$. The general problem of RL is to find a strategy maximizing the reward over trajectories. This strategy is also called *policy*.

Policy Gradient Several methods exist to tackle this problem. We limit ourselves to deep reinforcement learning (i.e., solutions involving Deep Neural Networks) in this section as it is the technique we will use later in this chapter, please refer to Sutton and Barto (1998) for a thorough introduction to RL.

More specifically, we present *policy gradient* (Sutton et al., 1999) methods with the REINFORCE algorithms (Williams, 1992). In this approach, the agent policy

π is directly represented by a Neural Network (NN) taking as input the current state and producing a distribution $\pi(a_t|s_t)$ over the possible actions:

$$a_t \sim \pi_\theta(s_t).$$

In which θ corresponds to the parameters of the policy. The reward is given by the function $r_t = R(s_t, a_t, s_{t+1})$. Note that the reward for a given action depends on both the current and the next states, as the state transition mechanism can be stochastic. We define the reward of a trajectory as $R(\tau) = \sum_{t=0}^{T} r_t$.

The objective is to find the best possible agent, i.e., the agent that will maximize the reward obtained while interacting with the environment:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau\sim\pi_\theta} R(\tau) = \arg\max_\theta J(\pi_\theta). \tag{3.1}$$

Even though the actions and the reward function are not necessarily differentiable w.r.t. the policy parameters θ, we can still compute the gradient of the expectation of Equation 3.1:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \sum_\tau P(\tau|\theta) R(\tau) && (3.2)\\
&= \sum_\tau \nabla_\theta P(\tau|\theta) R(\tau) && (3.3)\\
&= \sum_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) && (3.4)\\
&= \mathbb{E}_{\tau\sim\pi_\theta} \left[\nabla_\theta \log P(\tau|\theta) R(\tau)\right] && (3.5)\\
&= \mathbb{E}_{\tau\sim\pi_\theta} \left[\sum_{t=0}^{T} \left[\nabla_\theta \log \pi_\theta(a_t|s_t)\right] R(\tau)\right]. && (3.6)
\end{aligned}$$

Where step in Equation 3.4 uses the log-derivative trick and step in Equation 3.6 relies on the identity $\nabla_\theta \log P(\tau|\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)$.

In the REINFORCE algorithms, $\nabla_\theta J(\pi_\theta)$ is approximated using Monte Carlo methods, letting the agent interact with the environment to collect several trajectories and then use Equation 3.6 to compute the approximation of the gradient. This approximation is used to update θ and the process is repeated until the policy converges.

3.3 Super Networks

We consider the classical supervised learning problem defined by an input space $\mathcal{X}$ and an output space $\mathcal{Y}$. In the following, input and output spaces correspond to multi-dimensional real-valued spaces. The training set is denoted as $\mathcal{D} = \{(x^1, y^1), ..., (x^\ell, y^\ell)\}$ where $x^i \in \mathcal{X}$, $y^i \in \mathcal{Y}$ and ℓ is the number of supervised examples. $\Delta : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is a differentiable loss function taking as input predictions from a model and the ground truth. At last, we consider a model $f : \mathcal{X} \to \mathcal{Y}$ that predicts an output given a particular input.

We first describe a family of models called *Super Networks* (S-networks), the main contribution of this chapter presented in §3.4.2 being a stochastic extension of this model. Note that the principle of Super Networks is not new and similar ideas have been already proposed in the literature under different names such as Deep Sequential Neural Networks (Denoyer and Gallinari, 2014), Neural Fabrics (Saxena and Verbeek, 2016), or even PathNet (Fernando et al., 2017) who were the first to use the name *Super Network* with an architecture close to ours but for a completely different purpose.

A Super Network is composed of a set of layers connected together in a direct acyclic graph (DAG) structure. Each edge is a (small) neural network, the S-Network corresponds to a particular combination of these neural networks and defines a computation graph. Examples of the ResNet Fabric and Convolutional Neural Fabric super networks are given in Figures 3.1 and 3.2, illustrating the two supports on top of which cost-constrained architectures will be discovered. The ResNet Fabric is a generalization of ResNets (He et al., 2016), while CNF has been proposed in Saxena and Verbeek (2016) (both super networks are presented in more details in §3.5.1). In both cases, our objective is to discover architectures that are efficient in both prediction quality and cost, by sampling edges over these S-networks. More formally, let us denote $l_1, ..., l_N$ a set of layers, N being the number of layers, such that each layer l_i is associated with a particular representation space $\mathcal{X}_i$ which is a multi-dimensional real-valued space. l_1 will be the *input layer* while l_N will be the *output layer*. We also consider a set of (differentiable) functions $f_{i,j}$ associated to each possible pair of layers such that $f_{i,j} : \mathcal{X}_i \to \mathcal{X}_j$. Each function $f_{i,j}$ is referred to as a *module* in the following. Note that each $f_{i,j}$ can make disk/memory/network operations having consequences on the inference cost of the S-network. Each $f_{i,j}$ module is associated with parameters in $\theta_{i,j} \in \theta$, where θ corresponds to the ensemble of parameters of the Super-Network. In the rest of this chapter, we omit θ in the notation when not necessary for sake of clarity.

On top of this structure, any binary adjacency matrix $E = \{e_{i,j}\}, (i, j) \in [1; N]^2$ over the N layers represent a particular architecture. In this chapter, we decide to focus on the subset of such architectures having the following properties:

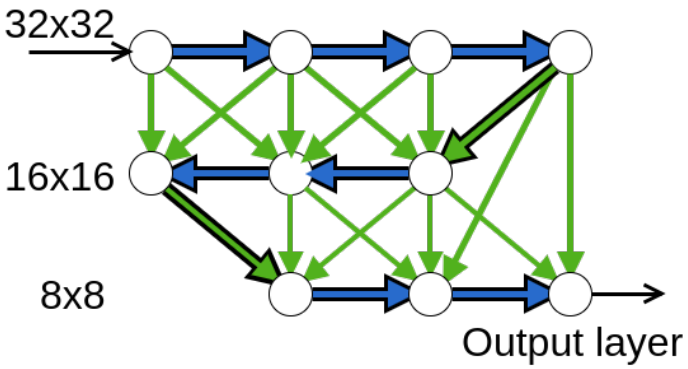


Figure 3.1. – **ResNet Fabric**: The ResNet Fabric is a super network that includes the ResNet model as a particular sub-graph. Each row corresponds to a particular size and number of feature maps. Each edge represents a simple *building block* as described in He et al. (2016) (i.e., two stacked convolution layers and a shortcut connection). We use projection shortcuts (with 1x1 convolutions) for all connections going across different feature map sizes (green edges). Note that here, the subgraph corresponding to the bold edges is a ResNet-20. By increasing the width of the ResNet Fabric, we can include different variants of ResNets, starting from ResNet-20 with width 3 up to Resnet-110 with a width of 18 .

- Non-recurrent architectures, corresponding to the cases where E represents a DAG.

- Single input and single output models, corresponding to the E with a single source node l_1 and a single sink node l_N .

Different matrices E will thus correspond to different super network architectures. A super network will be denoted (E, θ) in the following, θ being the parameters of the different *modules* and E being the architecture of the super network.

Predicting with S-networks: The computation of the output $f(x, E, \theta)$ given an input x and a S-network (E, θ) is made through a classic forward algorithm, the main idea being that the output of modules $f_{i,j}$ and $f_{k,j}$ leading to the same layer l_j will be added in order to compute the value of l_j. Let us denote $l_i(x, E, \theta)$ the value of layer l_i for input x, the computation is recursively defined as:

$$\begin{aligned} \text{Input: } & l_1(x, E, \theta) = x \\ \text{Layer Computation: } & l_i(x, E, \theta) = \sum_k e_{k,i} f_{k,i}(l_k(x, E, \theta)). \end{aligned} \tag{3.7}$$

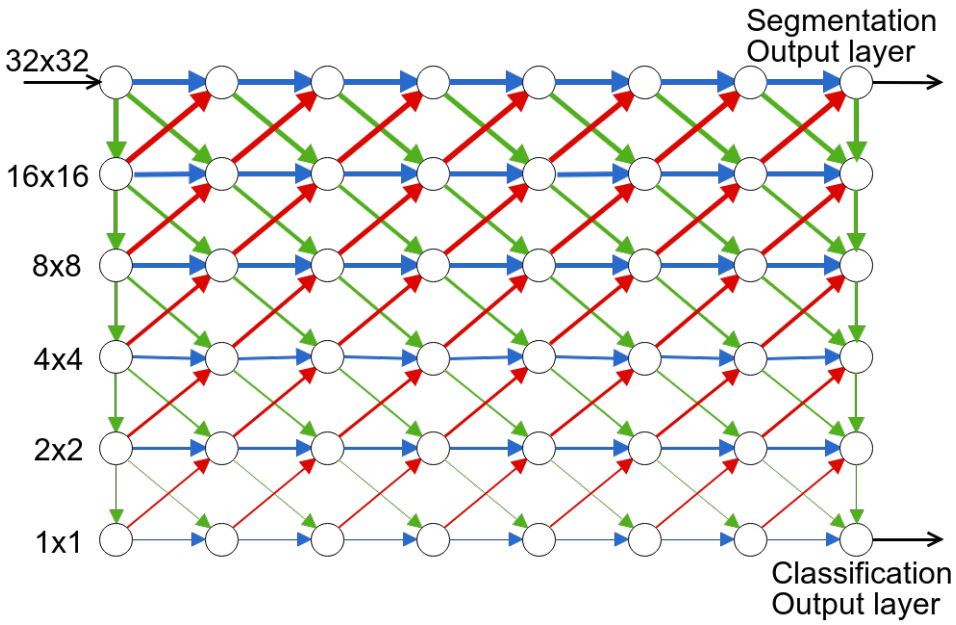


Figure 3.2. – **Convolutional Neural Fabrics** (Saxena and Verbeek, 2016): Each row corresponds to a particular resolution of feature maps. The number of features map is constant across the whole network. Each edge represents a convolution layer. The color of an edge represents the difference between input and output feature maps resolutions. Blue edges keep the same resolution, green edges decrease the resolution (stride > 1) and red edges increase the resolution (upsampling). An addition is used to aggregate the feature maps at each node before forwarding the result to the next layers.

The simple definitions of eq. 5 allow us to implement the forward algorithm as presented in alg. 1, on which we will build in §3.4.2. In this configuration, provided that each module $f_{i,j}$ used in E is differentiable, the training of the corresponding θ can be made using classical back-propagation and gradient-descent techniques.

3.4 Learning Cost-constrained architectures

Given the super-network we defined in the previous section, the idea for obtaining cost-efficient models is to consider that the structure E of the S-network (E, θ) describes not a single neural network architecture but a large set of possible architectures. Indeed, each sub-graph of E (i.e., each subset of edges) corresponds itself to a super network. Let us introduce the notation $H \odot E$, where H corre-

Algorithm 1: Super Network forward algorithm

Data: x, E, θ

$l_1 \leftarrow x$; // Init the first layer

for $i \in [2..N]$ **do**

  $l_i \leftarrow \sum_{k<i} e_{k,i} f_{k,i}(l_k; \theta_{k,i})$; // Propagate through the super network graph

end

return l_N

sponds to a binary matrix used as a mask to select the edges in E and $\odot$ is the Hadamard product. Our objective is now to identify the best matrix H such that the corresponding super network $(H \odot E, \theta)$ will be a network efficient in terms of both predictive quality and computation/memory/... cost.

In the next sections, we introduce the tools we need to be able to discover such structures, the overall structure is as follows:

1. First, we formalize this problem as a combinatorial problem where one wants to discover the best matrix H in the set of all possible binary matrices of size $N \times N$.
2. Since this optimization problem is intractable, we propose a new family of models called *Stochastic Super Networks* where E is sampled following a parametrized distribution Γ before each prediction.
3. We then show that the resulting budgeted learning problem is continuous and that its solution corresponds to the optimal solution of the initial budgeted problem (Proposition 1).
4. Finally, we propose a practical learning algorithm to learn Γ and θ simultaneously using gradient descent techniques.

3.4.1 Budgeted Architectures Learning

Given a specific sub-network represented by the (0,1) matrix H of size $N \times N$, let us define the function $C : \{0,1\}^{N \times N} \rightarrow \mathbb{R}^+$ which evaluates the cost of an architecture. We can now measure $C(H \odot E)$, corresponding to the computation cost of the super network $(H \odot E, \theta)$. Note that even if we consider that the cost only depends on the network architecture, the model could easily be extended to costs that depend on the input x to process or to stochastic costs, the only required property of $C(H \odot E)$ being that this cost must be measurable during training.

Let us also define C the maximum cost the user would allow. For instance, when solving the problem of *learning a model with a computation time lower than*

200 ms then $\mathbf{C}$ is equal to $200ms$. In this case, the function C would associate to each architecture its time to evaluate in ms on the target infrastructure.

Given these definitions, we aim at solving the following constrained optimization problem:

$$\begin{aligned} \min_{H,\theta} \quad & \frac{1}{\ell} \sum_i \Delta(f(x^i, H \odot E, \theta), y^i) \\ \text{subject to} \quad & C(H \odot E) \leq \mathbf{C}. \end{aligned} \tag{3.8}$$

We can reformulate the problem of equation 3.8 as an unconstrained optimization problem by using the penalty method:

$$\min_{H,\theta} \frac{1}{\ell} \sum_i \Delta(f(x^i, H \odot E, \theta), y^i) + \lambda \max(0, C(H \odot E) - \mathbf{C}). \tag{3.9}$$

where λ corresponds to the importance of the cost penalty. Note that the evaluated cost is specific to the particular infrastructure on which the model is ran. For instance, if $\mathbf{C}$ is the cost in milliseconds, the value of $C(H \odot E)$ will not be the same depending on the device on which the model is used. In other words, solving this problem on a mobile device will produce a different structure that the one obtained when solving this problem on a cluster of GPUs which is an important property of our model.

Finding a solution to this learning problem is far from trivial since it involves the computation of all possible architectures which is prohibitive ($\mathcal{O}(2^N)$ in the worst case) and the handling of the non-differentiable cost function C necessitate special care. We explain in the next section how both these problems can be solved at the same time using *Stochastic Super Networks*.

3.4.2 Stochastic Super Networks

To overcome this combinatorial problem during the search procedure, we introduce a probability distribution over sub-graphs of E. This distribution, called Γ, has the same dimensionality as E and each of its component $\gamma_{i,j}$ is the parameter of an independent Bernoulli distribution. Given a particular architecture E, we consider the following stochastic model – called **Stochastic Super Network** (SSN) – that computes a prediction in two steps:

1. A binary matrix H is sampled following the independent components of Γ. This operation is denoted $H \sim \Gamma$.
2. The final prediction is made using the associated sub-graph i.e. by computing $f(x, H \odot E, \theta)$.

Algorithm 2: Stochastic Super Network forward algorithm

Data: x, E, Γ, θ
$H \sim \Gamma$; // Sample an architecture
$l_1 \leftarrow x$;
for $i \in [2..N]$ **do**
  $l_i \leftarrow \sum_{k<i} e_{k,i} h_{k,i} f_{k,i}(l_k; \theta_{k,i})$; // Propagate through the sampled network
end
return l_N

A SSN is thus defined by a triplet (E, Γ, θ), where both Γ and θ contain learnable parameters. The updated forward procedure is presented in alg.2.

We can now rewrite the budgeted learning objective of Equation 3.8 and introduce the optimal parameters Γ^* and θ^* as:

$$\Gamma^*, \theta^* = \arg\min_{\Gamma,\theta} \frac{1}{\ell} \sum_i \mathbb{E}_{H\sim\Gamma} \left[\Delta(f(x^i, H \odot E, \theta), y^i) + \lambda \max(0, C(H \odot E) - \mathbf{C}) \right]. \quad (3.10)$$

Proposition 1. When the solution of Equation 3.10 is reached, then the models sampled following (Γ^*) and using parameters θ^* are optimal solution of the problem of Equation 3.8.

Demonstration of Proposition 1 Let us consider the stochastic optimization problem defined in Equation 3.10. The schema of the proof is the following:

- First, we lower bound the value of Equation 3.10 by the optimal value of Equation 3.8.

- Then we show that this lower bound can be reached by some particular values of Γ and θ in Equation 3.10. Said otherwise, the solution of Equation 3.10 is equivalent to the solution of 3.8.

Let us denote:

$$B(H \odot E, \theta, \lambda) = \frac{1}{\ell} \sum_i \Delta(f(x^i, H \odot E, \theta), y^i) + \lambda \max(0, C(H \odot E) - C) \quad (3.11)$$

Given a value of Γ, let us denote $supp(\Gamma)$ all the H matrices that can be sampled following Γ. The objective function of Equation 3.10 can be written as:

$$\begin{aligned} E_{H\sim\Gamma}[B(H\odot E,\theta,\lambda)] &= \sum_{H\in supp(\Gamma)} B(H\odot E,\theta,\lambda)P(H|\Gamma) \\ &\geq \sum_{H\in supp(\Gamma)} B((H\odot E)^*,\theta^*,\lambda)P(H|\Gamma) \\ &= B((H\odot E)^*,\theta^*,\lambda). \end{aligned} \tag{3.12}$$

where $(H\odot E)^*$ and θ^* correspond to the solution of:

$$(H\odot E)^*,\theta^* = \arg\min_{H,\theta} B(H\odot E,\theta,\lambda) \tag{3.13}$$

Now, it is easy to show that this lower bound can be reached by considering a value of Γ^* such that $\forall H\in supp(\Gamma), H\odot E = (H\odot E)^*$. This corresponds to a value of Γ where all the probabilities associated with edges in E are equal to 0 or 1.

Said otherwise, solving the stochastic problem will provide a deterministic model that has a good predictive performance under the given cost constraint.

Edge Sampling: In order to avoid inconsistent architectures where the input and the output layers are not connected, we sample H using the following procedure: For each layer l_i visited in the topological order of E (from the first layer l_1 to the last one l_N) and for all $k<i$: If l_k is connected to the input layer l_1 based on the previously sampled edges, then $h_{k,i}$ is sampled following the corresponding Bernoulli distribution with parameter $\gamma_{k,i}$. In the other cases, $h_{k,i}=0$. In practice we optimize Γ using the real-valued parameter $\hat{\Gamma}=\sigma^{-1}(\Gamma)$ with σ a logistic function (i.e., $\gamma_{k,i}$ is obtained by applying the *sigmoid* function over a real-valued parameter, this is made implicit in our notations).

3.4.3 Learning Algorithm

We consider the generic situation where the cost-function $C(.)$ is unknown but can be observed at the end of the computation of the model over an input x. Note that this case also includes stochastic costs where C is a random variable, caused by some network latency during distributed computation for example. We now describe the case where C is deterministic.

Let us denote $\mathcal{D}(x, y, \theta, E, H)$ the quality of the super network $(H \odot E, \theta)$ on a given training pair (x, y):

$$\mathcal{D}(x, y, \theta, E, H) \quad = \quad \Delta(f(x, H \odot E, \theta), y) + \lambda \max(0, C(H \odot E) - \mathbf{C}) \quad (3.14)$$

We propose to use a policy gradient inspired algorithm to learn θ and Γ. Let us denote $\mathcal{L}(x, y, E, \Gamma, \theta)$ the expectation of $\mathcal{D}$ over the possible sampled matrices H:

$$\mathcal{L}(x, y, E, \Gamma, \theta) = \mathbb{E}_{H \sim \Gamma} \left[\mathcal{D}(x, y, \theta, E, H) \right] \quad (3.15)$$

The gradient of $\mathcal{L}$ can be written as:

$$\begin{aligned} \nabla_{\theta,\Gamma} \mathcal{L}(x, y, E, \Gamma, \theta) \\ = \sum_H P(H|\Gamma) \left[(\nabla_{\theta,\Gamma} \log P(H|\Gamma)) \mathcal{D}(x, y, \theta, E, H) \right] \\ + \sum_H P(H|\Gamma) \left[\nabla_{\theta,\Gamma} \Delta(f(x, H \odot E, \theta), y) \right]. \end{aligned} \quad (3.16)$$

Derivation of $\nabla_{\theta,\Gamma}\mathcal{L}(x, y, E, \Gamma, \theta)$:

$$\begin{aligned}
\nabla_{\theta,\Gamma}\mathcal{L}(x, y, E, \Gamma, \theta) &= \nabla_{\theta,\Gamma}\mathbb{E}_{H\sim\Gamma}\mathcal{D}(x, y, \theta, E, H) && (3.17)\\
&= \nabla_{\theta,\Gamma}\sum_H P(H|\Gamma)\mathcal{D}(x, y, \theta, E, H) && (3.18)\\
&= \sum_H \nabla_{\theta,\Gamma}(P(H|\Gamma)\mathcal{D}(x, y, \theta, E, H)) && (3.19)\\
&= \sum_H \nabla_{\theta,\Gamma}(P(H|\Gamma))\mathcal{D}(x, y, \theta, E, H)\\
&\quad + P(H|\Gamma)\nabla_{\theta,\Gamma}\mathcal{D}(x, y, \theta, E, H) && (3.20)
\end{aligned}$$

Using the log-derivative trick:

$$\begin{aligned}
&= \sum_H P(H|\Gamma)\nabla_{\theta,\Gamma}\log P(H|\Gamma)\mathcal{D}(x, y, \theta, E, H)\\
&\quad + P(H|\Gamma)\nabla_{\theta,\Gamma}\mathcal{D}(x, y, \theta, E, H) && (3.21)\\
&= \sum_H P(H|\Gamma)((\nabla_{\theta,\Gamma}\log P(H|\Gamma))\mathcal{D}(x, y, \theta, E, H)\\
&\quad + \nabla_{\theta,\Gamma}\Delta(f(x, H\odot E, \theta), y)) && (3.22)\\
&= \sum_H P(H|\Gamma)\left[(\nabla_{\theta,\Gamma}\log P(H|\Gamma))\mathcal{D}(x, y, \theta, E, H)\right]\\
&\quad + \sum_H P(H|\Gamma)\left[\nabla_{\theta,\Gamma}\Delta(f(x, H\odot E, \theta), y)\right] && (3.23)
\end{aligned}$$

Using Equation 3.14:

$$\begin{aligned}
&= \sum_H P(H|\Gamma)\left[(\nabla_{\theta,\Gamma}\log P(H|\Gamma))\Delta(f(x, H\odot E, \theta), y)\right]\\
&\quad + \lambda\sum_H P(H|\Gamma)\left[(\nabla_{\theta,\Gamma}\log P(H|\Gamma))\max(0, C(H\odot E) - \mathbf{C})\right]\\
&\quad + \sum_H P(H|\Gamma)\left[\nabla_{\theta,\Gamma}\Delta(f(x, H\odot E, \theta), y)\right]. && (3.24)
\end{aligned}$$

The first term of equation 3.16 (corresponding to the first two terms or equation 3.24) is the gradient over the log-probability of the sampled structure H. This term is very similar to the policy gradient equations presented in §3.2. This similarity allows us to interpret this result intuitively: $\mathcal{D}(x, y, \theta, E, H)$ can be seen as the reward that an agent selecting the architecture would receive after each choice. The last term is the standard gradient of the prediction loss given the sampled architecture $(H \odot E)$ and allows us to update θ.

Learning can now be made using back-propagation and stochastic-gradient descent algorithms as it is made in Deep Reinforcement Learning models. Note

that in practice, in order to reduce the variance of the estimator, the update is made following:

$$\nabla_{\theta,\Gamma}\mathcal{L}(x,y,E,\Gamma,\theta) \approx (\nabla_{\theta,\Gamma}\log P(H|\Gamma))(\mathcal{D}(x,y,\theta,E,H)-\tilde{\mathcal{D}}) + \nabla_{\theta,\Gamma}\Delta(f(x,H\odot E,\theta),y). \quad (3.25)$$

where H is sampled following Γ, and where $\tilde{\mathcal{D}}$ is the average value of $\mathcal{D}(x,y,\theta,E,H)$ computed on a batch of learning examples.

3.5 Experiments

3.5.1 Implementation

We study two particular super network architectures:

The Convolutional Neural Fabric The first network we use in our experiments is based on the dense Convolutional Neural Fabrics. It has first been proposed in Saxena and Verbeek (2016) and can be seen as a multi-layer and multi-scale convolutional neural network used for both **image classification** and **image segmentation**. As shown in Figure 3.2, this architecture has 2 axes: The first axis represents the different columns (or width) W of the network while the second axis corresponds to different scales (or height) H of output feature maps, the first scale being the size of the input images, each subsequent scale being of a size reduced by a factor of 2 up to the last scale corresponding to a single scalar.

Each layer (l,s) in this fabric takes its input from three different layers of the preceding column: (i) One with a finer scale $(l-1,s-1)$ on which a convolution with stride 2 is applied to obtain feature maps having half the size of the input, (ii) one with the same scale $(l-1,s)$ on which a convolution with stride 1 is applied to obtain feature map of the same resolution as the input and (iii) one with a coarser scale $(l-1,s+1)$ on which convolution with stride 1 is applied after a factor 2 up-sampling to obtain feature maps having twice the size of the input. The three feature blocks are then added before passing through the ReLU activation function to obtain the final output of this layer (l,s). Each convolution layer uses kernels of size 3 and is followed by a batch normalization layer (Ioffe and Szegedy, 2015).

The first and last columns are the only two which have vertical connections within scales of the same layer. This is made to allow the propagation of the information to all nodes in the first column and to aggregate the activations of

the last column to compute the final prediction. A more detailed description of this architecture can be found in the CNF original article.

We used two different Convolutional Neural Fabrics in our experiments: One for the classification task (CIFAR-10 and CIFAR-100) with $W = 8$ columns, $H = 6$ scales and 128 filters per convolution, and one for the segmentation task (Part Label) with $W = 8$ layers, $H = 9$ scales (from 256x256 to 1x1 feature map sizes) and 64 filters per convolution. Different values of W ($W = 4$ and $W = 2$) are used as smaller-footprint baselines.

The ResNet Fabric Based on the ResNet architecture (He et al., 2016), the structure of a ResNet Fabric can be seen as a standard ResNet on which extra modules (edges) have been added. It is exclusively used for **image classification** in our experiments.

Illustrated in Figure 3.1, it is constructed as a stack of k groups of layers, each group being composed of $2n$ layers where n represents the width of the Fabric. The feature maps size and number of filters stay constant across the layers of each group and are modified between groups.

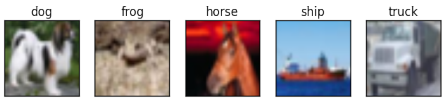
Due to its linear structure, the standard ResNet architecture spans a limited number of possible (sub-)architectures. To increase the size of the search space, we add several connections between groups as shown in the figure: each block in the second to last groups receives two (for the first and last block of each group) or three (for every other block) inputs from preceding groups. To stay consistent with the rest of the network, each connection is a *basic block* (He et al., 2016) composed of 2 convolutional layers and a shortcut connection.

In our experiments, we use stacks of $k = 3$ blocks and $n = \{3, 5, 7, 9, 18\}$ to respectively include the ResNet-$\{20, 32, 44, 56, 110\}$ in the Fabric. Between each block, the feature maps size is reduced by a factor of 2, and the number of maps is doubled.

Such super network not only allows our algorithm to search an architecture on the output map size dimension (as is the case with the CNF-based super network) but also on the depth dimension, benefiting from the residual connections and the added layer to find an appropriate depth while respecting the given budget. Note that a particular sub-graph, in bold on Figure 3.1, of the ResNet Fabric exactly corresponds to the ResNet model. We thus aim at testing the ability of our approach to discover ResNet-inspired efficient architectures, or at least to converge to a ResNet model that is known to be efficient.

Cost Functions. For these two architectures denoted *B-CNF* and *B-ResNet*, we consider three different costs functions:

(a) MNIST



(b) CIFAR-10

(c) Part Label

(d) CIFAR-100

Figure 3.3. – Datasets used to evaluate the Budgeted Super Networks.

- The *computation cost* measured as the number of operations (i.e., the number of Mult-Add operations required to fully evaluate a network) made by the sampled architecture as used in Dong et al. (2017) or Huang et al. (2018). Note that this cost is highly correlated with the execution time. Expressing constraint directly in *milliseconds* has been also investigated, with results similar to the ones obtained using the computation cost.
- The *memory consumption cost*, measured as the number of parameters of the resulting models
- The *distributed computation cost* which is detailed in §3.5.6 and corresponds to the ability of a particular model to be efficiently computed over a distributed environment.

3.5.2 Experimental Protocol and Baselines

3.5.2.1 Datasets

Both structures are evaluated on the image classification task using the CIFAR-10 and CIFAR-100 (Krizhevsky, 2009b) dataset. To assess if the BSN can be used in different settings, we also evaluate the CNF-based super network on the image segmentation task using the Part Label dataset (Kae et al., 2013). Finally, we use the MNIST (LeCun et al., 1998) dataset to test the behavior of the model on hand-crafted toy scenarios. Samples from each dataset are presented in figure 3.3. A short summary of each dataset and how we use it is presented below.

CIFAR-10. The CIFAR-10 dataset consists of 60k 32x32 images with 10 classes and 6000 images per class. The dataset is decomposed into 50k training and 10k testing images. We split the training set following the standard, i.e 45k training samples, and 5k validation samples. We use two data augmentation techniques:

padding the image to 36x36 pixels before extracting a random crop of size 32x32 and horizontally flipping. Images are then normalized in the range [-1,1].

CIFAR100. The CIFAR-100 dataset is similar to CIFAR-10, with 100 classes and 600 images per class. We use the same train/validation split and data augmentation technique as with CIFAR-10.

Part Labels. The Part Labels dataset is a subset of the LFW dataset composed of 2927 250x250 face images in which each pixel is labeled as one of the Hair/Skin/Background classes. The standard split contains 1500 training samples, 500 validation samples, and 927 test samples. Images are zero-padded from 250x250 to 256x256. We use horizontal flipping as data augmentation. Images are then normalized in the range [-1,1].

MNIST The MNIST dataset contains 60k 28x28 hand-written digit images. Similar to what is done we Cifar-10, we split this dataset into 50k training and 10k testing images. We split the train set, keeping 80% of the data for training and 20% for validation. Only random crops are used for data augmentation and images are normalized.

3.5.2.2 Target costs

Each model is trained with various values for the objective cost C. For the image classification problem, since we directly compare to ResNet, we select values of C that corresponds to the cost of the ResNet-20/32/44/56/110 architectures. This allows us to compare the performance of our method at the same cost level as the ResNet variants. When dealing with the B-CNF model, we select C to be the cost of different versions of the CNF model, having different width W. The height H being fixed by the resolution of the input image.

3.5.2.3 Model Selection Protocol

For each experiment, multiple versions of the different models are evaluated over the validation set during learning. Since our evaluation now involves both a cost and an accuracy, we select the best models using the Pareto front on the cost/accuracy curve on the validation set. The reported performances are then obtained by evaluating these selected models over the test set. The detailed procedure of the hyper-parameters and model selection is illustrated in figure 3.4 where many different models are reported on the validation set (blue circles) with the corresponding performance on the test set (red crosses). The selection of

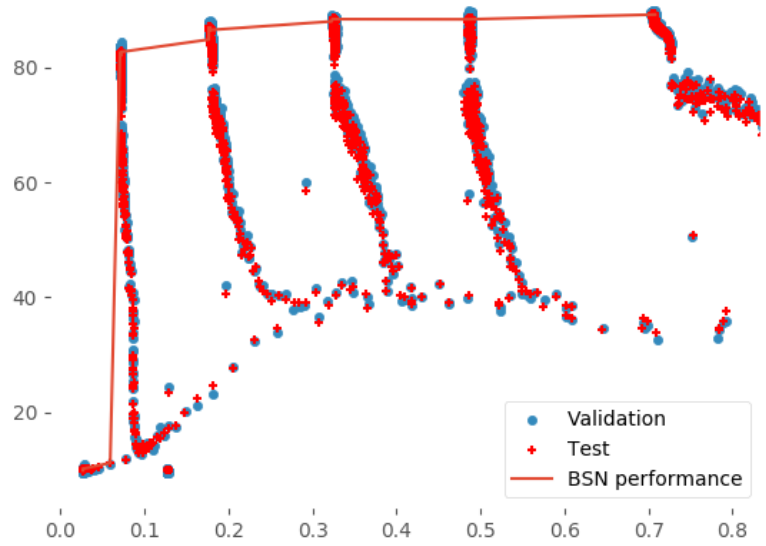


Figure 3.4. – Illustration of our model selection procedure. The horizontal axis represents the normalized cost while the vertical axis reports the accuracy performance.

reported models is obtained by computing the Pareto front of the accuracy/cost curve on the validation set and reporting the performance obtained on the test set by the model on the front.

When reporting results, we provide the performance of both reference models (ResNet and CNF) and the related existing models Low-Cost Collaborative Layer (LCCL) (Dong et al., 2017) and MSDNet (Huang et al., 2018) (under the anytime classification settings). Note that the baseline methods have been designed to reduce exclusively the *computation cost*, while the BSN can deal with any type of cost. We provide the performance of the budgeted version of ResNet (B-ResNet) and the budgeted version of CNF (B-CNF). For a fair comparison, the published results of ResNet and CNF are reported, but also the ones that we have obtained by training these models by ourselves, **our results being of better quality** than the previously published performance.

3.5.2.4 Training details

The learning is done using a classical stochastic gradient-descent algorithm for all parameters with learning rate decay, momentum of 0.9, and weight decay of 10^{-4} for θ.

When training our budgeted models, we first train the network for 50 "warm-up" epochs during which no sampling is done (The whole super network is trained). After this warm-up phase, the probability of each edge is initialized and we start sampling architectures.

The real-valued parameter associated with each layer $\hat{\gamma_{i,j}}$ (and used to generate the parameter $\gamma_{i,j}$ of the Bernoulli distribution of the corresponding edge) is initialized to the constant 3, resulting in a $\sigma(3) \approx 0.95$ initial probability of being sampled for each edge.

On CIFAR-10 and CIFAR-100 datasets we train all models for 300 epochs. We start with a learning rate of 10^{-1} and divide it by 10 after 150 and 225 epochs. On Part Label dataset all models are trained for 200 epochs with a learning rate initialized to 10^{-1} and divided by 10 after 130 epochs.

For all models and all cost functions, we select the λ hyper-parameter based on the order of magnitude m of the maximum authorized cost $\mathbf{C}$. λ is determined using cross-validation on values logarithmically spaced between 10^{m-1} and 10^{m+1}.

The source code of our implementation is openly available [1].

1. https://github.com/TomVeniat/bsn

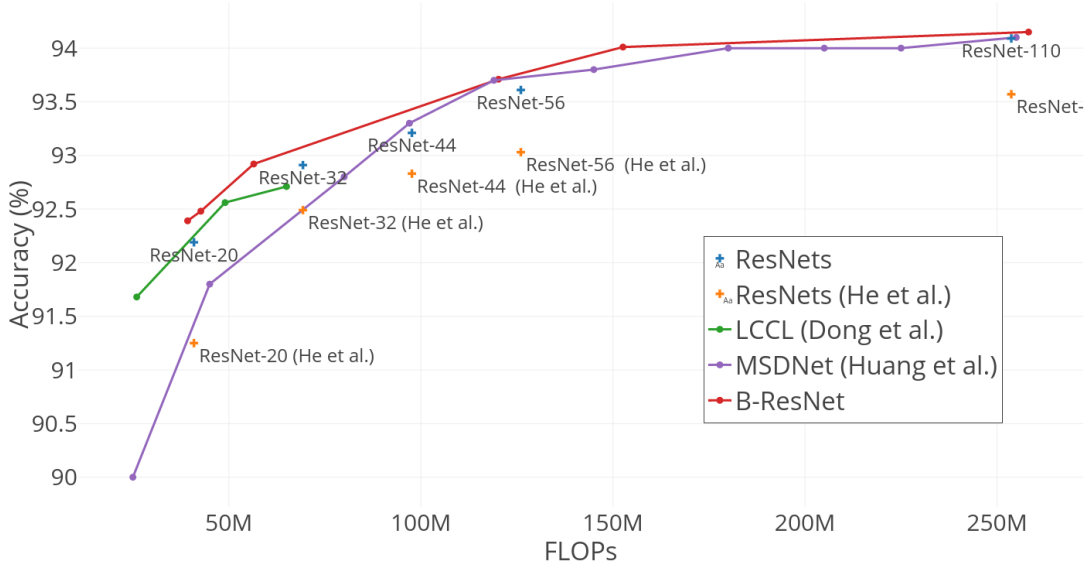
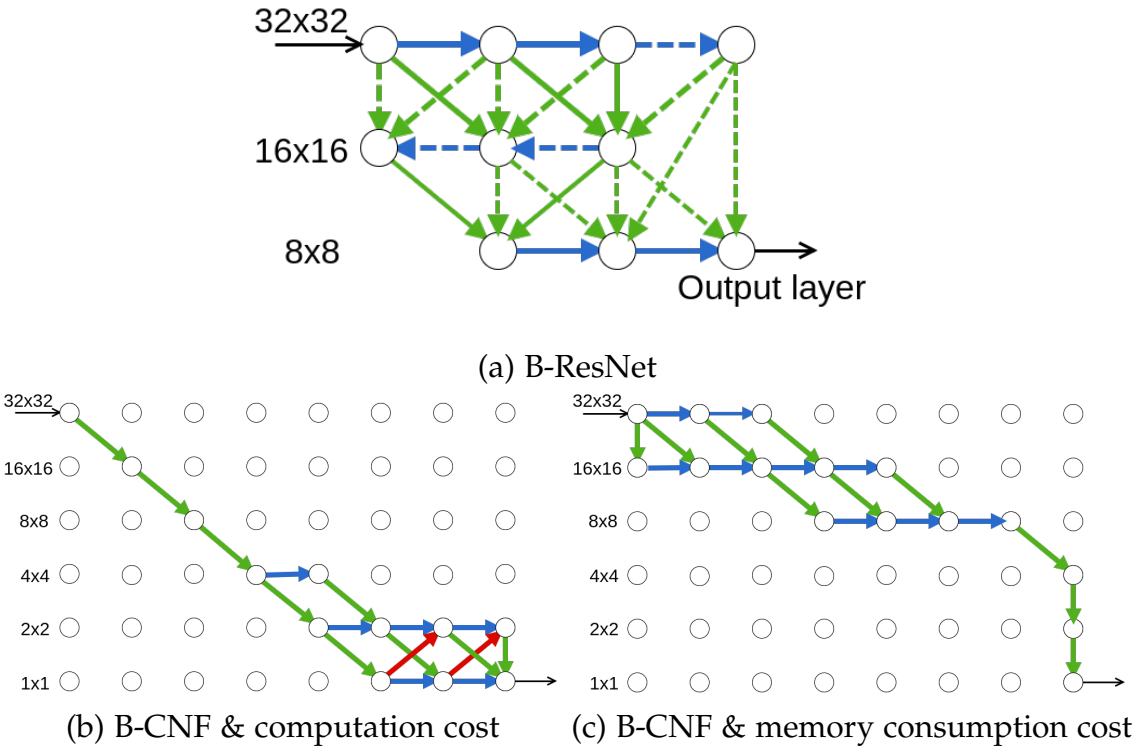


Figure 3.5. – Accuracy/Time trade-off using B-ResNet on CIFAR-10.

3.5.3 Reducing the computation cost of classification models

Figure 3.5 and Table 3.1 show the performance of different models over CIFAR-10. Each point corresponds to a model evaluated both in terms of accuracy and *computation cost*. When considering the B-ResNet model, and by fixing the value of C to the computation cost of the different ResNet architectures, we obtain budgeted models that have approximatively the same costs as the ResNets, but with higher accuracy. For example, ResNet-20 obtains an accuracy of 92.19% for 40.9×10^6 flop, while B-ResNet is able to discover an architecture with 92.39% accuracy at a slightly lower cost (39.25×10^6 flop). Moreover, the B-ResNet model also outperforms existing approaches like MSDNet or LCCL, particularly when the *computation cost* is low (i.e., for architectures that can be computed at a high speed). When comparing CNF to B-CNF, one can see that our approach can considerably reduce the computation cost while keeping high accuracy. For example, one of our learned models obtained an accuracy of 93.14% with a cost of 103×10^6 flop while CNF has an accuracy of 92.54% for a cost of 406×10^6 flop. Note that the same observations can be drawn for CIFAR-100 (Table 3.2).

Figure 3.6a and Figure 3.6b illustrate two architectures discovered by B-ResNet and B-CNF with a low *computation cost*. One can see that B-ResNet converges to an architecture that is a little bit different than the standard ResNet architecture, explaining why its accuracy is better. On the CNF side, the BSN model is able to extract a model that has a minimum of high-resolution convolution operations, resulting in a high speedup.



(a) B-ResNet

(b) B-CNF & computation cost

(c) B-CNF & memory consumption cost

Figure 3.6. – Discovered architectures: **(Top)** is a low *computation cost* B-ResNet where dashed edges correspond to connections in which the two convolution layers have been removed (only shortcut or projection connections are kept). **(Left)** is a low *computation cost* B-CNF where high-resolution operations have been removed. **(Right)** is a low *memory consumption cost* B-CNF: the algorithm has mostly kept all high resolution convolutions since they allow fine-grained feature maps and have the same number of parameters than lower-resolution convolutions. It is interesting to note that our algorithm, constrained with two different costs, automatically learned two different efficient architectures.

Model	FLOPs (millions)	Accuracy
ResNet (He et al., 2016)		our/*original*
ResNet-110	253.70	94.09/*93.57*
ResNet-56	126.01	93.61/*93.03*
ResNet-44	97.64	93.21/*92.83*
ResNet-32	69.27	92.91/*92.49*
ResNet-20	40.90	92.19/*91.25*
Low Cost Collaborative Layer (Dong et al., 2017)		
LCCL (ResNet-110)	166	93.44
LCCL (ResNet-44)	65	92.71
LCCL (ResNet-32)	49	92.56
LCCL (ResNet-20)	26	91.68
Multi Scale DenseNet (Huang et al., 2018)		(values read on plot)
MSDNet	≈ 255	94.1
	≈ 180	94.0
	≈ 145	93.8
	≈ 97	93.3
	≈ 45	91.8
	≈ 25	90.0
Budgeted ResNet		
B-ResNet	407.51	94.29
	258.20	94.15
	152.60	94.01
	120.20	93.71
	56.47	92.92
	42.69	92.48
	39.25	92.39
CNF (Saxena and Verbeek, 2016)		our/*original*
CNF W=8	2,219.00	94.83/*90.58*
CNF W=4	1,010.00	93.75/*87.91*
CNF W=2	406.00	92.54/*86.21*
CNF W=1	54.00	89.91
Budgeted CNF		
B-CNF	2,150.00	94.92
	1,407.00	94.85
	1,144.00	94.69
	103.00	93.14
	85.00	92.17

Table 3.1. – Accuracy/speed trade-off on CIFAR-10 using B-ResNet and B-CNF. Values reported as *our* corresponds to results we obtained when training a reproduction of the models, *original* corresponds to values from the original article.

Model	FLOPs (millions)	Accuracy (%)
ResNet-110	253.7	71.85
ResNet-56	126	70.57
ResNet-44	97.64	70.28
ResNet-32	69.27	69.28
ResNet-20	40.9	67.14
MSDNet (Huang et al., 2018)	215	76
	180	75
	150	74
	109	72.5
	80	71
	45	67.5
	15	62.5
B-ResNet	349.5	73.28
	115.09	71.46
	69.84	70.27
	64.96	70.12
	46.29	69.02
	39.22	68.45

Table 3.2. – Accuracy/speed trade-off on Cifar-100 using ResNet Fabrics.

Model	# params (millions)	Accuracy (%)
ResNet (He et al., 2016)		our/*original*
ResNet-110	1.73	94.09/*93.57*
ResNet-56	0.86	93.61/*93.03*
ResNet-44	0.66	93.21/*92.83*
ResNet-32	0.47	92.91/*92.49*
ResNet-20	0.27	92.19/*91.25*
Budgeted ResNet		
B-ResNet	4.38	94.35
	2.27	94.2
	1.29	93.85
	0.48	93.42
	0.34	92.72
	0.3	92.52
	0.29	92.17
Convolutional Neural Fabrics (Saxena and Verbeek, 2016)		our/*original*
CNF W=8	18.04	94.83/*90.58*
CNF W=4	8.58	93.75/*87.91*
CNF W=2	3.85	92.54/*86.21*
CNF W=1	0.74	89.91
Budgeted CNF		
B-CNF	7.56	94.88
	4.98	94.58
	4.28	94.55
	3.67	94.42
	2.65	94.00
	1.19	93.53

Table 3.3. – Accuracy/memory trade-off on Cifar-10 using B-ResNet and B-CNF.

3.5.4 Reducing memory consumption on classification

We now present similar experiments considering the *memory consumption cost* that measures the number of parameters of the learned architectures. We want to demonstrate here the ability of the BSN to be used with a large variety of costs, and not only to reduce the computation time. Table 3.3 presents the results obtained on CIFAR-10. As with the *computation cost*, one can see that our approach can discover architectures that, given a particular memory cost, obtain better accuracy. For example, for a model having ≈ 0.47 millions parameters, ResNet-32 has a classification error of 7.81% while B-ResNet only makes 6.58% error with ≈ 0.48 million parameters.

Model	FLOPs(billions)	Accuracy
CNF	35.614	95.06
CNF W=8 (Saxena and Verbeek, 2016)	35.614	95.39
B-CNF	28.49	95.21
B-CNF	21.37	95.43

Table 3.4. – Accuracy/Speed trade-off on Part Label using CNF.

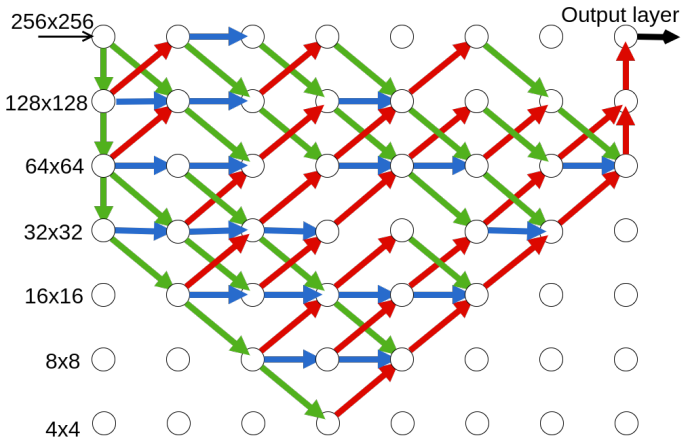


Figure 3.7. – Example of architecture learned for the semantic segmentation task.

3.5.5 Reducing the computation cost of segmentation models

We also perform experiments on the image segmentation task using the Part Label dataset with CNF and B-CNF (Table 3.4). In this task, the model computes a map of pixel probabilities, the output layer is now located at the top-right position of the CNF matrix. It is thus more difficult to reduce the overall computation cost. On the Part Label dataset, we are able to learn a BSN model with a computation gain of 40%. Forcing the model to reduce further the computation cost by decreasing the value of C results in inconsistent models. At a computation gain of 40%, BSN obtains an error rate of 4.57%, which can be compared with the error of 4.94% for the full model. The best B-CNF learned architecture is given in 3.7.

Figure 3.7 is an example of segmentation architecture discovered on the Part Label dataset using the *flop cost*. It is interesting to note that only one layer with 256x256 input and output is kept and that most of the computations are done at lower and therefore less-expensive layers.

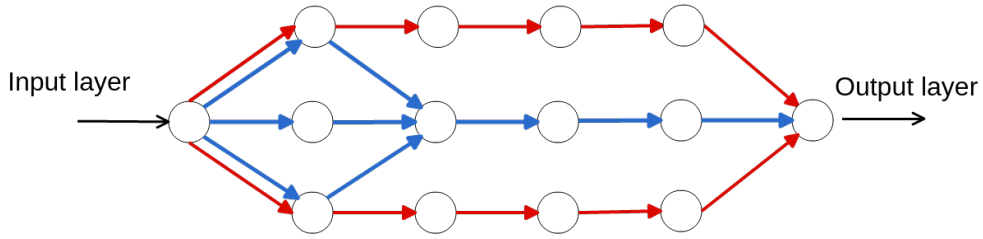


Figure 3.8. – Two networks illustrating the need to have a cost function evaluating the global architecture of a network. Considering an environment with $n = 2$ machines performing computations in parallel, the blue network composed of 9 computational modules has a *distributed computation cost* of 6 while the red network, composed of 10 modules, has a smaller cost of 5.

3.5.6 Learning Distributed Architectures

The third set of experiments is focused on distributed computing. For example, we can take the real-life example of a network that will once optimized have to run on a computing infrastructure where different modules can be computed simultaneously (e.g., on the different GPUs/computers of a large cluster).

In this case, we can evaluate the quality of an architecture by its ability to be efficiently parallelized. The *distributed computation cost* corresponds to the number of steps needed to compute the output of the network (e.g., on an architecture with $n = 2$ computers, two edges could be computed simultaneously if their inputs are ready to be processed). If an architecture is flat a sequence of layers, then this parallelization becomes impossible. An illustration of the interest of this cost function is provided in Figure 3.8.

The *distributed computation cost* function takes the following three elements as inputs:

1. A network architecture (represented as a graph for instance).
2. An allocation algorithm.
3. A maximum number of concurrent possible operations n.

The cost function then returns the number of computation cycles required to run the architecture given the allocation strategy on n parallel processing units.

It is very interesting to evaluate budgeted approaches on such a cost function because it allows measuring the ability of the model to handle complex costs that cannot be decomposed as a sum of individual modules costs as it is usually done in related works.

Results using the B-ResNet model are illustrated in Figure 3.9 and architectures learned using the B-CNF model are presented in Figure 3.10 for the CIFAR-10

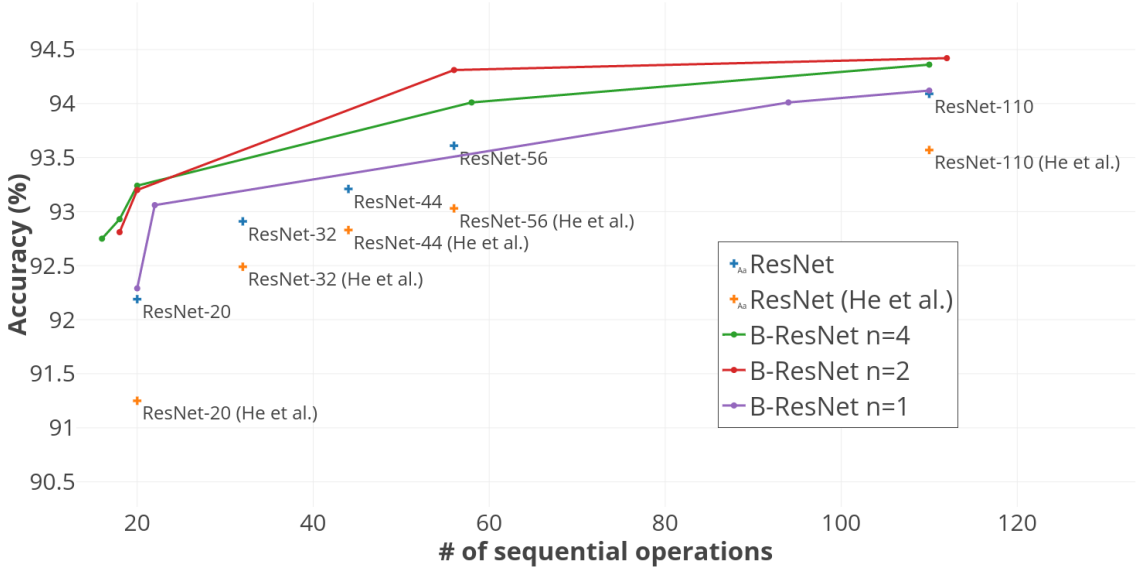
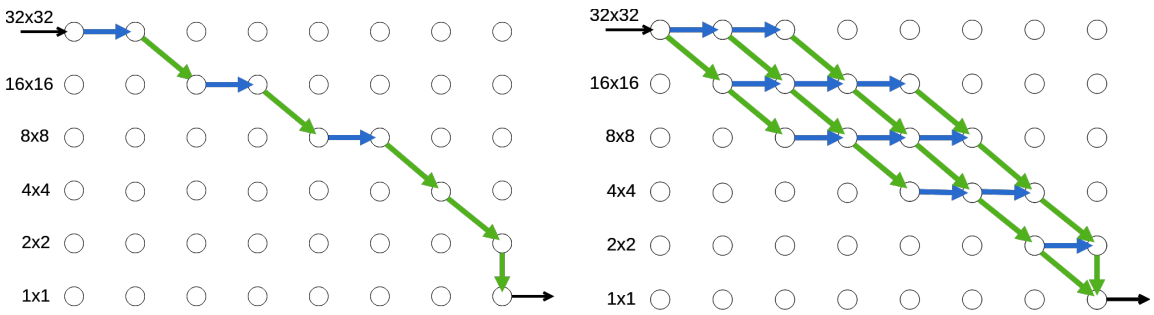


Figure 3.9. – Accuracy/number of operation for different number of cores n on CIFAR-10 using B-ResNet.

dataset. Note that ResNet is typically an architecture that cannot be efficiently distributed since it is a sequences of modules. One can see that our approach is able to find efficient architectures for $n = 2$ and $n = 4$. Surprisingly, when $n = 4$ the discovered architectures are less efficient, which is mainly due to an over fitting of the training set, the cost constraint becomes too large and stop acting as a regularizer on the network architecture. On Figure 3.10, one can see two examples of architectures discovered when $n = 1$ and $n = 4$. The shape of the architecture when $n = 4$ clearly confirm that BSN is able to discover parallelized architectures, and to 'understand' the structure of this complex cost. Detailed numbers are presented in tables 3.5, 3.5 and 3.5 for both B-CNF and B-ResNet on Cifar-10 and in tables 3.8 and 3.9 for B-ResNet on the Cifar-100 task.



(a) $n = 1$ (b) $n = 4$

Figure 3.10. – Architectures discovered on CIFAR-10 using the distributed computation cost on the B-CNF model for different number of cores n. In both scenarios, the training procedure is the same and the resulting architecture takes 8 steps to be evaluated. It demonstrates that the BSN is able to adapt the structure to the underlying cost function without any prior knowledge about it.

Model	# of sequential operations	Accuracy %
ResNet (He et al., 2016)		our/*original*
ResNet-110	112.00	94.09/*93.57*
ResNet-56	58.00	93.61/*93.03*
ResNet-44	46.00	93.21/*92.83*
ResNet-32	34.00	92.91/*92.49*
ResNet-20	22.00	92.19/*91.25*
Budgeted ResNet		
B-ResNet	184.00	94.42
	110.00	94.12
	94.00	94.01
	22.00	93.06
	20.00	92.29
Convolutional Neural Fabrics (Saxena and Verbeek, 2016)		our/*original*
CNF W=8	171.00	94.83/*90.58*
CNF W=4	83.00	93.75/*87.91*
CNF W=2	39.00	92.54/*86.21*
CNF W=1	12.00	89.91
Budgeted CNF		
B-CNF	98.00	95.02
	50.00	94.62
	45.00	94.55
	39.00	94.35
	33.00	93.00
	26.00	92.91
	18.00	92.87

Table 3.5. – Results for *Distributed computation cost* on CIFAR-10 with $n = 1$

Model	# of sequential operations	Accuracy %
ResNet (He et al., 2016)		our/*original*
ResNet-110	110.00	94.09/*93.57*
ResNet-56	56.00	93.61/*93.03*
ResNet-44	44.00	93.21/*92.83*
ResNet-32	32.00	92.91/*92.49*
ResNet-20	20.00	92.19/*91.25*
Budgeted ResNet		
B-ResNet	179.00	94.36
	112.00	94.42
	56.00	94.31
	20.00	93.20
	18.00	92.81
Convolutional Neural Fabric (Saxena and Verbeek, 2016)		our/*original*
CNF W=8	90.00	94.83/*90.58*
CNF W=4	47.00	93.75/*87.91*
CNF W=2	26.00	92.54/*86.21*
CNF W=1	12.00	89.91
Budgeted CNF		
B-CNF	47.00	94.67
	30.00	94.68
	28.00	94.58
	24.00	94.41
	20.00	94.35
	18.00	92.86

Table 3.6. – Results for *Distributed computation cost* on CIFAR-10 with $n = 2$

Model	# of sequential operations	Accuracy %
ResNet (He et al., 2016)		our/*original*
ResNet-110	110.00	94.09/*93.57*
ResNet-56	56.00	93.61/*93.03*
ResNet-44	44.00	93.21/*92.83*
ResNet-32	32.00	92.91/*92.49*
ResNet-20	20.00	92.19/*91.25*
Busgeted ResNet		
B-ResNet	110.00	94.36
	58.00	94.01
	20.00	93.24
	18.00	92.93
	16.00	92.75
Convolutional Neural Fabric (Saxena and Verbeek, 2016)		our/*original*
CNF W=8	53.00	94.83/*90.58*
CNF W=4	31.00	93.75/*87.91*
CNF W=2	19.00	92.54/*86.21*
CNF W=1	12.00	89.91
Budgeted CNF		
B-ResNet	31.00	94.96
	25.00	94.72
	21.00	94.36
	18.00	93.86

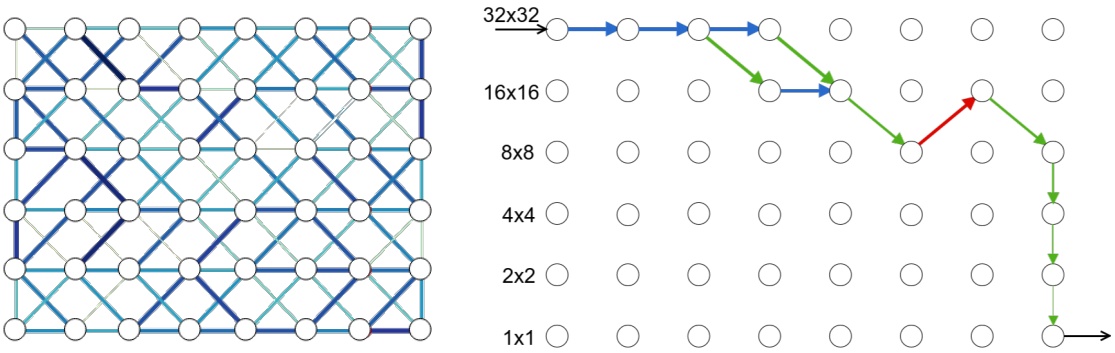
Table 3.7. – Results for *Distributed computation cost* on CIFAR-10 with $n = 4$

Model	# of sequential operations	Accuracy (%)
ResNet(He et al., 2016)		
ResNet-110	112.00	71.85
ResNet-56	58.00	70.57
ResNet-44	46.00	70.28
ResNet-32	34.00	69.28
ResNet-20	22.00	67.14
Budgeted ResNet		
B-ResNet	320.00	74.35
	184.00	73.85
	110.00	72.88
	67.00	72.02
	32.00	69.60
	20.00	68.48

Table 3.8. – Results for *Distributed computation cost* on CIFAR-100 with $n = 1$

Model	# of sequential operations	Accuracy (%)
ResNe (He et al., 2016)		
ResNet110	110.00	71.85
ResNet56	56.00	70.57
ResNet44	44.00	70.28
ResNet32	34.00	69.28
ResNet20	20.00	67.14
Budgeted ResNet		
B-ResNet	179.00	74.35
	112.00	73.85
	49.00	71.84
	29.00	69.94
	22.00	69.09

Table 3.9. – Results for *Distributed computation cost* on CIFAR-100 with $n = 2$



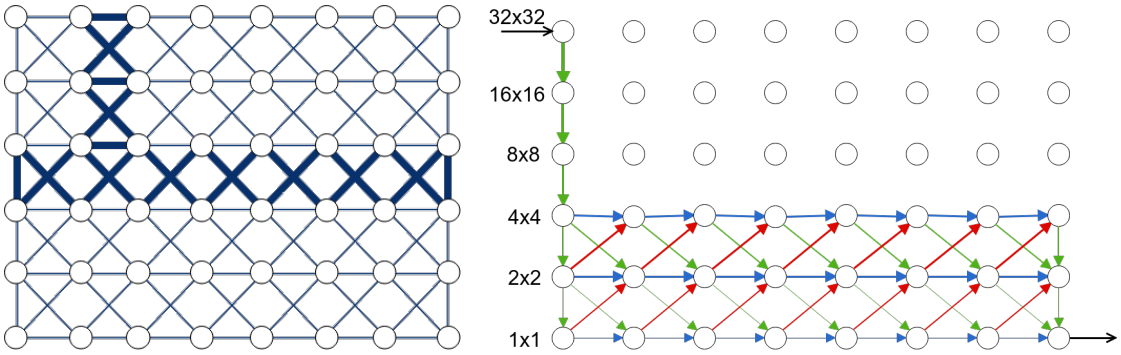
(a) (b)

Figure 3.11. – **Randomly sampling edges cost**. a shows the costs attributed randomly to each edge of the graph where darker blue correspond to larger costs. b corresponds to the architecture learned using these costs. The discovered path successfully avoids the most expensive layers.

3.5.7 Learning Infrastructure-Specific Architectures

Finally, we qualitatively evaluate B-CNF on handcrafted problems to observe whether the model behaves as expected when we use well-controlled cost functions:

1. The first setup, presented in Figure 3.11, consists of randomly attributing costs to the edges of the graph. As expected, we observe that the B-CNF is able to find a path minimizing the overall architecture cost while preserving the expressiveness of the model.
2. The second setup, presented in Figure 3.12, is a simulation of an application in the wild having to communicate to a cloud infrastructure through a set of expensive network communication edges. Here again, the path discovered lines up with the intuition: it is better to directly send the information to the cluster using the minimum number of edges than pre-computing too many features on-device and then having to send them all through expensive network calls.



(a) (b)

Figure 3.12. – **Simulating a cloud environment.** a shows the costs attributed to simulate a small amount of computation (e.g. a smart device) connected to more powerful servers through a very expensive communication channel (e.g. network calls). b shows the architecture selected for this cloud simulation. BSN learns a path that takes only one expensive edge while making use of the maximum amount of computation available to process the image afterward.

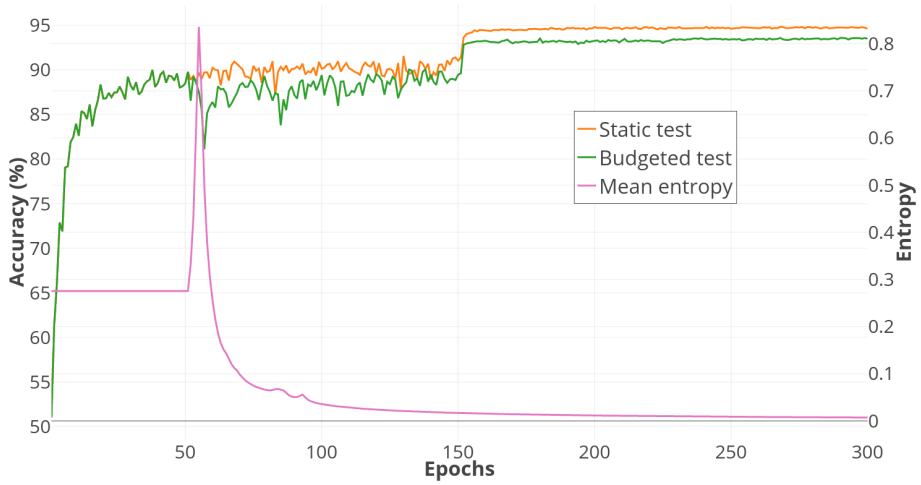


Figure 3.13. – Evolution of the loss function and the entropy of Γ during training. The period between epoch 0 and 50 is the burn-in phase. The learning rate is divided by 10 after epoch 150 to increase the convergence speed.

3.5.8 Learning Dynamics

Figure 3.13 illustrates the learning dynamics of B-CNF and CNF. First, one can see (entropy curve) that the model becomes deterministic at the end of the learning procedure, and thus converges to a unique architecture. Moreover, the training speed of B-CNF and CNF are comparable showing that our method does not result in a slower training procedure. Note that the figure illustrates the fact that during a burn-in period, we don't update the probabilities of the edges, which allows us to obtain a faster convergence speed.

3.6 Conclusion

We proposed a new model called *Budgeted Super Network* able to automatically discover cost-constrained neural network architectures by specifying a maximum authorized cost. The experiments in the computer vision domain show the effectiveness of our approach. Its main advantage is that BSN can be used for any costs (computation cost, memory cost, etc.) without any assumption on the shape of this cost. A promising research direction is now to study whether this model could be adapted to reduce the training time (instead of the test computation time). This could for example be done using meta-learning approaches.

CHAPTER 4

ADAPTIVE NAS FOR AUDIO PROCESSING ON A BUDGET

Contents

Chapter abstract

The problem of keyword spotting i.e. identifying keywords in a real-time audio stream is mainly solved by applying a neural network over successive sliding windows. Due to the difficulty of the task, baseline models are usually large, resulting in a high computational cost and energy consumption level.

We propose a new method called SANAS (Stochastic Adaptive Neural Architecture Search) which can adapt the architecture of the neural network on-the-fly at inference time such that small architectures will be used when the stream is easy to process (silence, low noise, ...) and bigger networks will be used when the task becomes more difficult.

We show that this adaptive model can be learned end-to-end by optimizing a trade-off between the prediction performance and the average computational cost per unit of time. We empirically validate the proposed model on the Speech Commands dataset (Warden, 2018), showing that our approach is able to strike the right balance between maintaining a high recognition performance while

better allocating its budget, outperforming classical approaches which are all using static network architectures.

The work in this chapter has led to the publication of a conference paper:

- Tom Véniat, Olivier Schwander, and Ludovic Denoyer (2019). "Stochastic Adaptive Neural Architecture Search for Keyword Spotting". In: *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2019, Brighton, United Kingdom, May 12-17, 2019*. IEEE, pp. 2842–2846. DOI: 10.1109/ICASSP.2019.8683305. URL: https://doi.org/10.1109/ICASSP.2019.8683305.

Resources to reproduce the work in this chapter are publicly available:

- Source code of the experiments: https://github.com/TomVeniat/SANAS.

4.1 Introduction

With sight, hearing is one of the most important senses of living beings. It allows one to perceive a large amount of information about the surrounding environment and, for most animal species, plays a major role in communication. Given its importance, it is natural to want to be able to automatically process audio signals to extract relevant information. For example, Acoustic Scene Classification is the problem of identifying the environment in which audio samples has been recorded (e.g. busy office, street, forest, ...) (Mesaros et al., 2018) and Sound Event Detection can be used to monitor the environment for specific events (e.g. car passing by, crying babies, knocks on a door, ...)(Turpault et al., 2019; Serizel et al., 2020). Since hearing is also used when communicating, we can also define tasks aiming at extracting information from speech signals. For example, we can use speech to automatically detect a range of psychiatric disorders (Low et al., 2020) or perform Automatic Speech Recognition (also known as speech-to-text) which is now widely used in connected devices such as smartphones or virtual assistants.

Let us focus on this last use-case, which is by design aimed to be used on small devices and therefore a natural application for budgeted algorithms. The traditional voice assistant pipeline is two-fold: (i) the first part is to continuously listen to sounds from the environment to detect some particular key-words and (ii) once a keyword is detected, identify the specific request of the user. While this second part can support some latency, this is not the case of the keyword spotting part, which furthermore may have to run on-device for latency, privacy, or energy consumption reasons. This task, illustrated in Figure 4.1, consists in listening continuously to an audio stream and being able to detect in real-time a

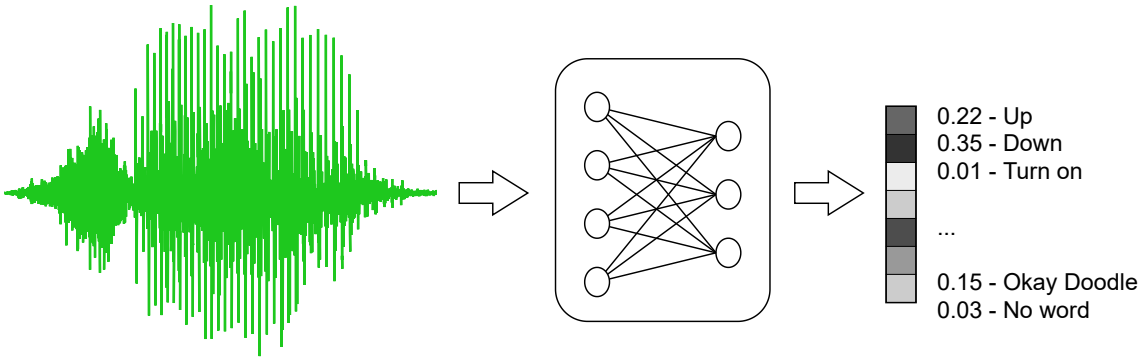


Figure 4.1. – The keyword spotting problem aims at identifying a set of predetermined keywords in an audio stream. This problem is usually tackled by feeding successive chunks of 1 second of audio signal in a neural network trained to recognize these specific words. The result is then a sequence of probability distributions over our vocabulary, which can be post-processed to make application-specific decisions.

word in a set of provided keywords. This problem thus implies detecting when a word is pronounced, but also which word has been pronounced.

Some recent works Sainath and Parada, 2015; Arik et al., 2017; Tang and Lin, 2018 proposed to use convolutional neural networks (CNN) in this streaming context, applying a particular model to successive sliding windows Sainath and Parada, 2015; Tang and Lin, 2018; Rybakov et al., 2020 or combining CNNs with recurrent neural networks (RNN) to keep track of the context Arik et al., 2017. In such cases, the resulting system spends the same amount of time processing each audio frame, irrespective of the content of the frame or its context. The resulting system is, in that case, quite slow and consumes a lot of energy, the same big network being applied every 20 or 50 ms. If the application requires close to real-time inference, this method also gives a hard constraint on the size of the model that can be used: if the time taken to evaluate a frame is longer than the time hop between successive frames, the predictor will inevitably accumulate delay and won't be able to keep pace with the stream.

Current methods either only focus on prediction performance, overlooking real-world constraints like limited computation resources or strong latency requirements, or try to take into account these constraints by using a carefully tuned model that will respect the hard *by-timestep* constraint. While this approach effectively solves the budget constraint for each step (and on average), it results in inefficient usage of computation resources in real-world conditions where the complexity of the samples are different in the same sequence. In this chapter, we propose an adaptive method allowing to dynamically select the neural network to use at each timestep, allowing to spend more resources on harder examples while making predictions on average faster than static Neural Networks

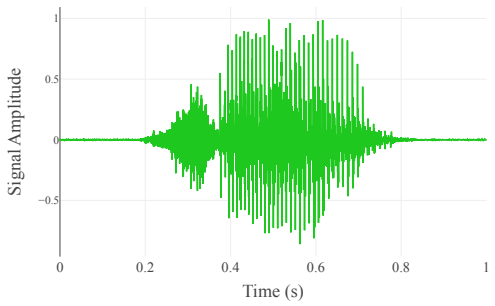


Figure 4.2. – Raw audio recording of the word "Sheila".

4.2 Audio and speech processing background

When dealing with audio signals, the data usually comes in the raw time/amplitude form as illustrated in Figure 4.2. When processing it with neural networks, the most standard route is to apply a fixed set of pre-processing operations instead of directly using the raw representation of the signal. The objective of this pre-processing is to represent the signal in a way similar to how the human ear perceives it.

Starting from this raw signal, illustrated in Figure 4.2, the usual preprocessing pipeline for speech processing is the following:

1. The first step is to slice it into a series of short chunks, short enough to consider that the frequency distributions will no change much during each timeframe. In our experiments, we use overlapping 30ms long chunks every 10ms.
2. We compute the periodogram of each timeframe to get an estimate of how much power there is around each frequency.
3. Using triangular overlapping windows, map the power of the frequencies to the Mel scale. Since the human ear doesn't perceive the frequencies linearly (the perceived difference between 20Hz and 120Hz will be much bigger than the perceived difference between 10kHz and 10.1kHz), the Mel scale is conceived in such a way that the increment between pitches will be judged to be perceptually similar for humans.
4. Take the log of the energies obtained above. Here again, this is based on human perception, since we don't perceive "loudness" linearly either.
5. The last step is to apply a Discrete Cosine Transform (DCT) to the log energies to decorrelate them. The resulting coefficients are called Mel Frequency Cepstral Coefficient (MFCC) (we usually keep the first 40 DCT coefficients) and are the features we will feed to the algorithm for the corresponding timeframe.

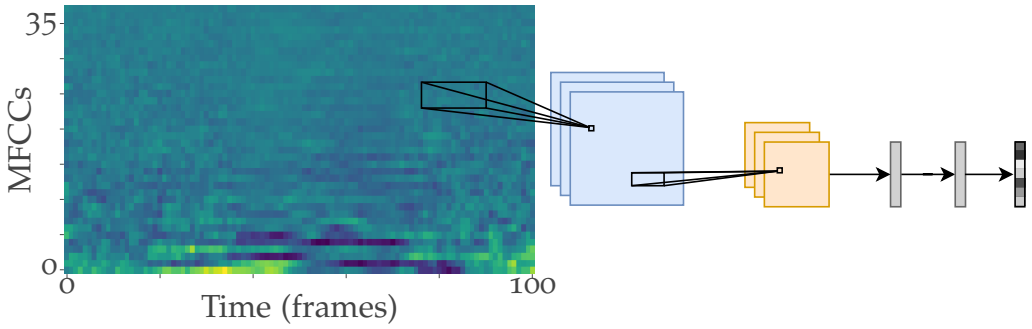


Figure 4.3. – Example of speech processing using a ConvNet composed of 2 convolutional layers and 3 linear layers. Features corresponds to the raw signal of Figure 4.2. The Feature extraction procedure is detailed in Section 4.2.

Once the MFCCs for each chunk of the audio signal are computed, we can stack them to create a 2D time/MFCC feature map that can be fed to a standard Convolutional Neural Network (ConvNet). In our experiments, each feature map is computed using 1s of audio signal at a time, corresponding to 100 successive timeframes and resulting in 100x40 feature maps.

4.3 Stochastic Adaptive Neural Architecture Search

The main idea behind our proposition is that, when dealing with streams of information, a model able to adapt its architecture to the difficulty of the prediction problem at each timestep would be more efficient than a static model. Such a dynamic model would for example decide to use a small architecture when the current step is easy and a larger architecture when the prediction is more difficult. If done correctly, this strategy would allow the model to save a lot of computation or energy consumption while retaining most of the performance of the static version. Figure 4.4 presents the static approach commonly used in this setting while Figure 4.5 presents a toy illustration of how the model we propose in this chapter should behave.

The Stochastic Adaptive Architecture Search (SANAS) method presented in this chapter (detailed in Section 4.3.3) is built to achieve this goal: it is, as far as we know, the first architecture search method producing a system that adapts the architecture of a neural network on-the-fly during prediction such that small architectures will be used on the simple steps of the streams (e.g., silence, low noise, ...) and more powerful networks will be used when facing difficult samples.

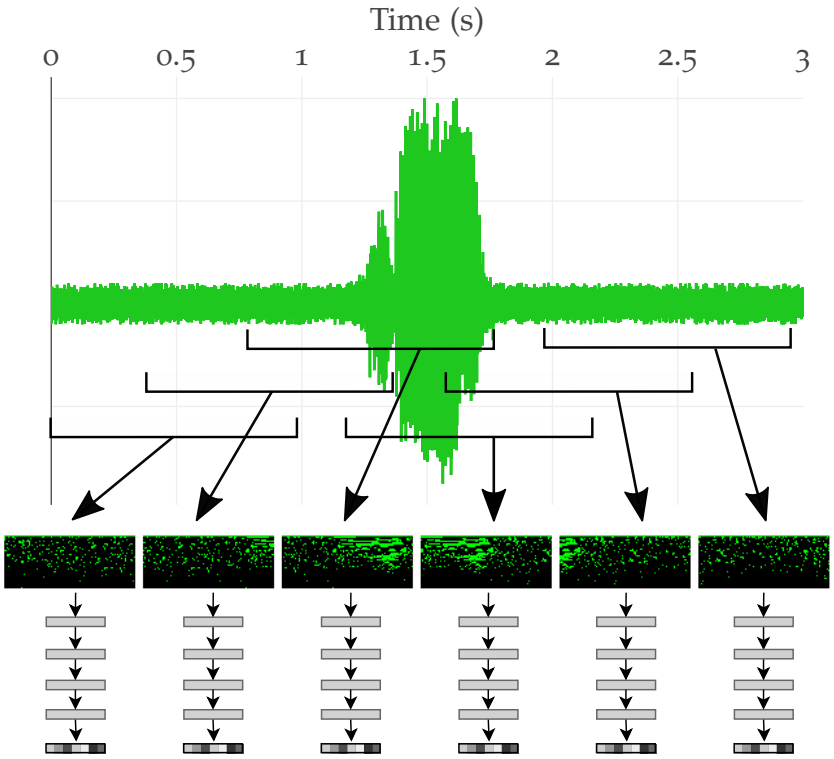


Figure 4.4. – Standard workflow for the keyword spotting use-case, using the same model to sequentially infer the different timeframes of the sequence.

After learning, SANAS can process audio streams at a higher speed than classical static methods while keeping a high recognition rate, spending more prediction time on complex signal windows and less time on easier ones (see Section 4.4). First, we formally define the setting in Section 4.3.1.

Then, we present in detail the main contribution of the chapter: a stochastic model which adapts itself to the sequence it is processing and can be learned end-to-end by optimizing a trade-off between the prediction performance and the average amount of Floating Point Operation (FLOP) spent per unit of time.

Finally, we experimentally validate our model on the Speech Commands dataset (Warden, 2018) in Section 4.4. Our quantitative results show that this approach leads to a high recognition level while being much faster (and/or energy saving) than classical approaches where the network architecture is static. The final section analyses the model qualitatively, probing the network to observe how it behaves and what strategies are implemented by the controller to save time.

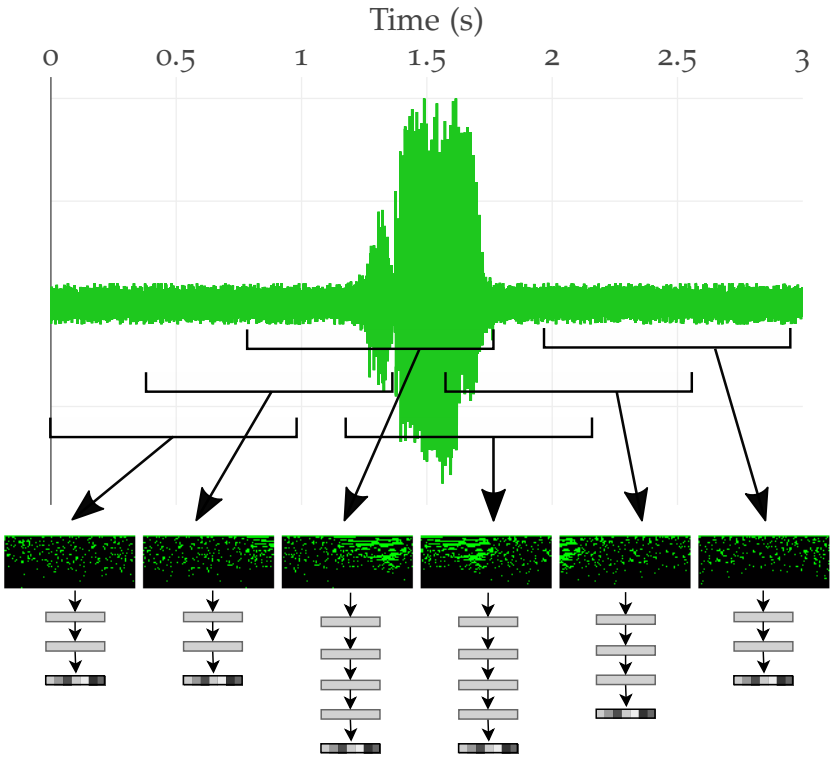


Figure 4.5. – Example of a dynamic system on the keyword spotting problem, the model is adapted to the amount of signal present in each frame.

4.3.1 Problem Definition

We consider the generic problem of stream labeling where, at each timestep, the system receives a datapoint denoted x_t and produces an output label y_t. In the case of audio streams, x_t is usually a time-frequency feature map, and y_t is the presence or absence of a given keyword. In classical approaches, the output label y_t is predicted using a neural network whose architecture [1] is denoted $\mathcal{A}$ and whose parameters are θ. We consider in this paper the recurrent modeling scheme where the context $x_1, y_1, ..., x_{t-1}, y_{t-1}$ is encoded using a latent representation z_t, such that the prediction at time t is made computing $f(z_t, x_t, \theta, \mathcal{A})$, z_t being updated at each timestep such that $z_{t+1} = g(z_t, x_t, \theta, \mathcal{A})$ - note that g and f can share some common computations.

For a particular architecture $\mathcal{A}$, the parameters are learned over a training set of labeled sequences $\{(x^i, y^i)\}_{i\in[1,N]}$, N being the size of the training set, by solving:

1. a precise definition of the notion of architecture is given further.

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \big[\sum_{t=1}^{\#x^i} \Delta(f(z_t, x_t, \theta, \mathcal{A}), y_t) \big]. \tag{4.1}$$

where $\#x^i$ is the length of sequence x^i, and Δ a differentiable loss function. At inference, given a new stream x, each label $\hat{y_t}$ is predicted by computing $f(x_1, \hat{y_1}, ..., \hat{y_{t-1}}, x_t, \theta^*, \mathcal{A})$, where $\hat{y_1} \dots \hat{y_{t-1}}$ are the predictions of the model at previous timesteps. In that case, the computation cost of each prediction step solely depends on the architecture and is denoted $C(\mathcal{A})$.

4.3.2 Stochastic Adaptive Architecture Search: Principles

We propose now a different setting where the architecture of the model can change at each timestep depending on the context of the prediction z_t. At time t, in addition to producing a distribution over possible labels, our model also maintains a distribution over possible architectures denoted $P(\mathcal{A}_t|z_t, \theta)$. The prediction y_t being now made following the distribution $f(z_t, x_t, \theta, \mathcal{A}_t)$ and the context update being $z_{t+1} = g(z_t, x_t, \theta, \mathcal{A}_t)$. In that case, the cost of a prediction at time t is now $C(\mathcal{A}_t)$, which also includes the computation of the architecture distribution $P(\mathcal{A}_t|z_t, \theta)$. It is important to note that, since the architecture $\mathcal{A}_t$ is chosen by the model, it has the possibility to learn to control this cost itself. A budgeted learning problem can thus be defined as minimizing a trade-off between prediction loss and average cost. Considering a labeled sequence (x, y), this trade-off is defined as :

$$\mathcal{L}(x, y, \theta) = \mathbb{E}_{\{\mathcal{A}_t\}} \big[\sum_{t=1}^{\#x} [\Delta(f(z_t, x_t, \theta, \mathcal{A}_t), y_t) + \lambda C(\mathcal{A}_t)] \big]. \tag{4.2}$$

where $\mathcal{A}_1, ..., \mathcal{A}_{\#x}$ are sampled following $P(\mathcal{A}_t|z_t, \theta)$ and λ controls the trade-off between cost and prediction efficiency. Considering that $P(\mathcal{A}_t|z_t, \theta)$ is differentiable, and following the derivation schema proposed in Chapter 3, this cost can be minimized using the Monte-Carlo estimation of the gradient. Given one sample of architectures $\mathcal{A}_1, ..., \mathcal{A}_{\#x}$, the gradient can be approximated by:

$$\begin{aligned} \nabla_\theta \mathcal{L}(x, y, \theta) \approx \big(\sum_{t=1}^{\#x} \nabla_\theta \log P(\mathcal{A}_t|z_t, \theta) \big) \mathcal{L}(x, y, \mathcal{A}_1, ..., \mathcal{A}_{\#x}, \theta) \\ + \sum_{t=1}^{\#x} \nabla_\theta \Delta(f(z_t, x_t, \theta, \mathcal{A}_t), y_t). \end{aligned} \tag{4.3}$$

where

$$\mathcal{L}(x, y, \mathcal{A}_1, ..., \mathcal{A}_{\#x}, \theta) = \sum_{t=1}^{\#x} [\Delta(f(z_t, x_t, \theta, \mathcal{A}_t), y_t) + \lambda C(\mathcal{A}_t)] \tag{4.4}$$

In practice, a variance correcting value is used in this gradient formulation to accelerate the learning as explained in Williams (1992) and Wierstra et al. (2007).

4.3.3 The SANAS Model

We now instantiate the previous generic principles in a concrete model using the concept of *Super-Network* introduced in Section 3.3: the architecture search is reformulated into a sub-graph discovery problem in a large graph representing the search space.

NAS with Super-Networks (static case)

We propose here a quick reminder of Chapter 3, allowing us to introduce the formalism that we will need in the next sections: a Super-Network is a directed acyclic graph of layers $L = \{l_1, ...l_n\}$, of edges $E \in \{0, 1\}^{n \times n}$ and where each existing edge connecting layers i and j ($e_{i,j} = 1$) is associated with a (small) neural network $f_{i,j}$. The layer l_1 is the input layer while l_n is the output layer. The inference of the output is made by propagating the input x over the edges, and by summing, at each layer level, the values coming from incoming edges. Given a Super-Network, the architecture search can be made by defining a distribution matrix $\Gamma \in [0, 1]^{n \times n}$ that can be used to sample edges in the network using a Bernoulli distribution. Indeed, let us consider a binary matrix H sampled following Γ, the matrix $E \circ H$ defines a sub-graph of E and corresponds to a particular neural-network architecture which size is smaller than E ($\circ$ being the Hadamard product). Learning Γ thus results in doing architecture search in the space of all the possible neural networks contained in Super-Network. At inference, the architecture with the highest probability is chosen.

SANAS with Super-Networks

Concretely, our architecture can be decomposed into 3 major components as depicted in Figure 4.6:

The first one is a Recurrent Neural Network (RNN) g which will encode in its state z information about the global dynamics of the sequence. The second

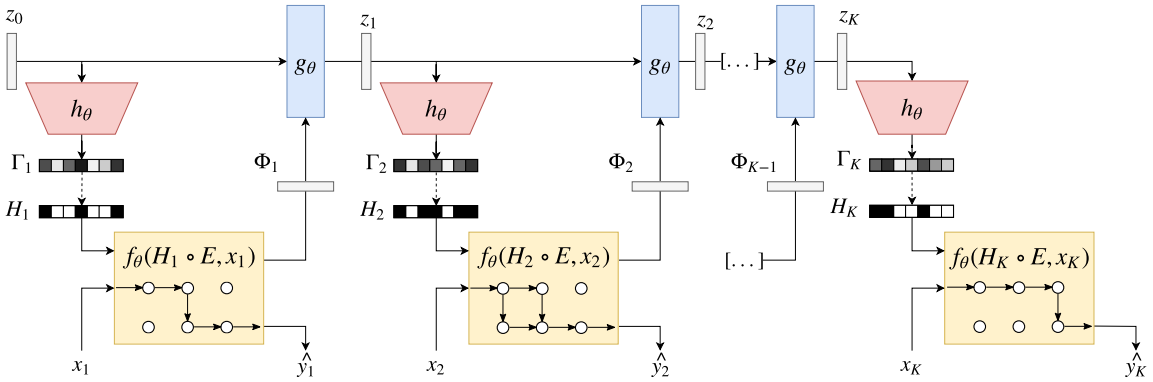


Figure 4.6. – **SANAS Architecture**. At timestep t, the distribution Γ_t is generated from the previous hidden state, $\Gamma_t = h(z_t, \theta)$. A discrete architecture H_t is then sampled from Γ_t and evaluated over the input x_t. This evaluation gives both a feature vector $\Phi(x_t, \theta, E \circ H_t)$ to compute the next hidden state, and the prediction of the model $\hat{y}_t$ using $f(z_t, x_t, \theta, E \circ H_t)$. Dashed edges represent sampling operations. At inference, the architecture which has the highest probability is chosen at each timestep.

component is the generator function $h(z)$ in charge of generating a distribution over architectures $\Gamma_t = h(z_t, \theta)$ from the hidden state z_t at each timestep. The last component is a stochastic super network f used for inference and feature encoding.

At a given timstep t, the RNN takes as input the feature vector from previous step ϕ_{t-1} and the previous latent state h_{t-1}. It then generates a distribution Γ_t from which a particular sub-graph represented by $H_t \sim \mathcal{B}(\Gamma_t)$ is sampled, $\mathcal{B}$ being a Bernoulli distribution. This particular sub-grap $E \circ H_t = \mathcal{A}_t$ corresponds to the architecture to use at time t.

Then the selected architecture is evaluated on sample x_t using the stochastic super network $f(z_t, x_t, \theta, E \circ H_t)$ to (i) extract the feature vector Φ_t and (ii) make a prediction y_t for the current timestep.

We can now use Φ_t to compute the next state using the RNN $z_{t+1} = g(z_t, \Phi_t, \theta)$ of the system. Different kind of RNN could be used here, we chose to use the Gated Recurrent Unit (GRU) (Cho et al., 2014) for its simplicity.

The learning of the parameters of the proposed model relies on a gradient-descent method based on the approximation of the gradient provided previously, which simultaneously updates the parameters θ and the conditional distribution over possible architectures.

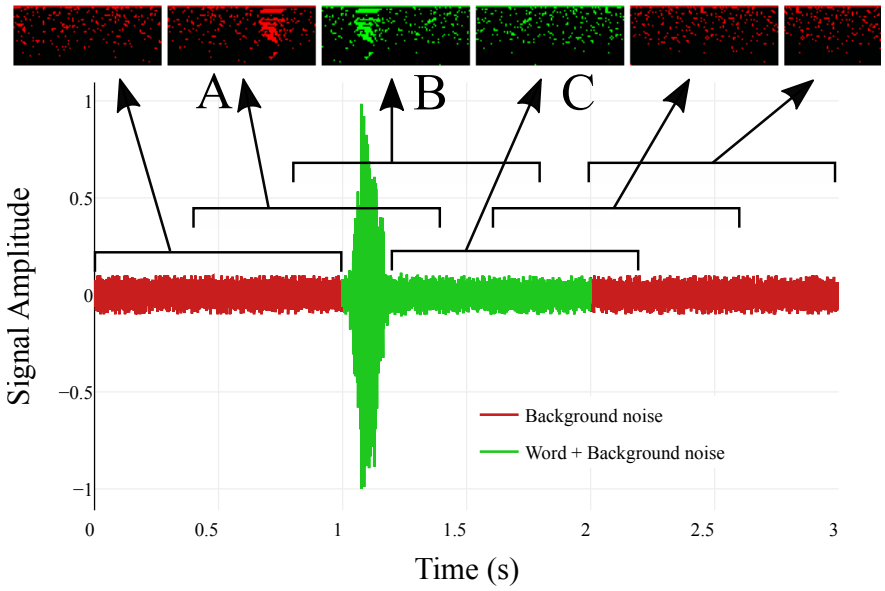


Figure 4.7. – Example of labeling using the method presented in Section 4.4. To build the dataset, a ground noise (red) is mixed with randomly located words (green). The signal is then split in 1s frames every 200ms. When a frame contains at least 50% of a word signal, it is labeled with the corresponding word (frame B and C – frame A is labeled as *bg-noise*). Note that this labeling could be imperfect (see frame A and C).

4.4 Experiments

We train and evaluate our model using the Speech Commands dataset Warden, 2018. It is composed of 65000 short audio clips of 30 common words. As done in Tang and Lin, 2018; Tang et al., 2018; Zhang et al., 2017, we treat this problem as a classification task with 12 categories: 'yes', 'no', 'up', 'down', 'left', 'right', 'on', 'off', 'stop', 'go', '*bg-noise*' for background noise and '*unknown*' for the remaining words.

4.4.1 Methodology

Instead of directly classifying 1-second samples, we use this dataset to generate between 1 and 3 second long audio files by combining a background noise coming from the dataset with a randomly located word (see Figure 4.7), the signal-to-noise ratio being randomly sampled with a minimum of 5dB.

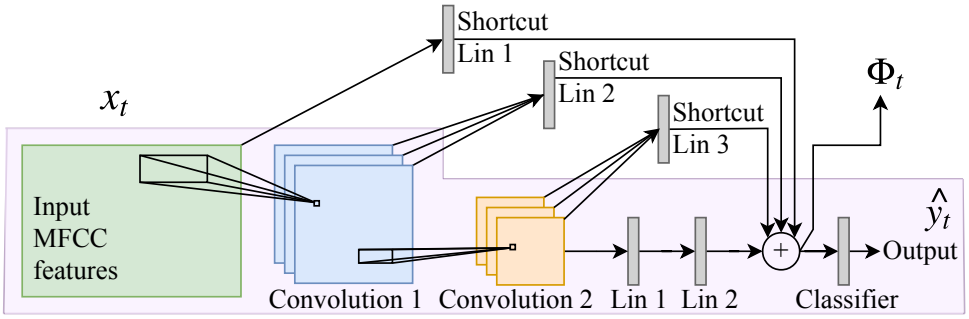


Figure 4.8. – SANAS architecture based on *cnn-trad-fpool3* (Sainath and Parada, 2015). Edges between layers are sampled by the model. The highlighted architecture is the base model on which we have added shortcut connections. Conv1 and Conv2 have filter sizes of (20,8) and (10,4). Both have 64 channels and Conv1 has a stride of 3 in the frequency domain. Linear 1,2 and the Classifier have 32, 128, and 12 neurons respectively. Shortcut linear layers all have 128 neurons to match the dimension of the classifier.

We thus obtain a dataset of about 30,000 small files [2] and then split this dataset into the train, validation, and test sets using an 80:10:10 ratio. The sequence of frames is created by taking overlapping windows of 1 second every 200ms.

As described in Section 4.2, the input features for each window are computed by extracting 40 MFCC on 30ms frames every 10ms and stacking them to create the 2D time/frequency maps.

For the evaluation, we use both the prediction accuracy and the number of operations per frame (FLOPs) value. Similar to how we did it in Chapter 3 (Figure 3.4), model selection is made by training multiple models, selecting the best models on the validation set, and computing their performance on the test set. This is because here again, the 'best models' in terms of both accuracy and FLOPs are the models located on the Pareto front of the accuracy/cost validation curve.

These models are also evaluated using the *matched*, *correct*, *wrong* and *false alarm* (FA) metrics as proposed in Warden, 2018 and computed over the one hour stream provided with the original dataset. Note that these last metrics are computed after using a post-processing method that ensures a labeling consistency as described in the reference paper.

2. tools for building this dataset are available at `http://github.com/TomVeniat/SANAS` with the open-source implementation.

Match	Correct	Wrong	FA	FLOPs per frame
		cnn-trad-fpool3		
81.7%	72.8%	8.9%	0.0%	124.6M
		***cnn-trad-fpool3* + shortcut connections**		
82.9%	77.9%	5.0%	0.3%	137.3M
		SANAS		
0.0%	0.0%	0.0%	0.0%	0.0
61.2%	53.8%	7.3%	0.7%	519.2K
48.0%	38.3%	9.7%	0.4%	526.4K
62.0%	57.3%	4.8%	0.1%	22.9M
86.5%	80.7%	5.8%	0.3%	37.7M
85.0%	79.6%	5.4%	0.2%	38.6M
83.7%	79.3%	4.4%	0.3%	44.2M
83.5%	78.2%	5.3%	0.0%	44.8M
85.8%	80.6%	5.3%	0.2%	45.2M
86.3%	80.6%	5.7%	0.2%	48.8M
84.1%	78.0%	6.1%	0.0%	51.4M
81.7%	76.4%	5.3%	0.1%	94.0M
80.7%	75.6%	5.1%	0.1%	101.9M
81.4%	76.3%	5.2%	0.2%	105.4M

Table 4.1. – Evaluation of models on 1h of audio from Warden (2018) containing words roughly every 3 seconds with different background noises. We use the label post processing and the streaming metrics proposed in Warden, 2018 to avoid repeated and noisy detections. We report the performance of SANAS for different budget constraint levels. Matched % corresponds to the portion of words detected, either correctly (Correct %) or incorrectly (Wrong %). FA is *False Alarm*.

4.4.2 Baselines and Training Details

As baseline static model, we use a standard neural network architecture used for Keyword Spotting aka the *cnn-trad-fpool3* architecture proposed in Sainath and Parada, 2015 which consists of two convolutional layers followed by 3 linear layers. We then proposed a SANAS extension of this model (see Figure 4.8) with additional connections that will be adaptively activated (or not) during the audio stream processing.

In the SANAS models, the recurrent layer g is a one-layer GRU Cho et al., 2014 and the function h mapping from the hidden state x_t to the distribution over architecture Γ_t is a one-layer linear module followed by a sigmoid activation.

The models are learned using the ADAM Kingma and Ba, 2015 optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, gradient steps between 10^{-3} and 10^{-5} and λ in range

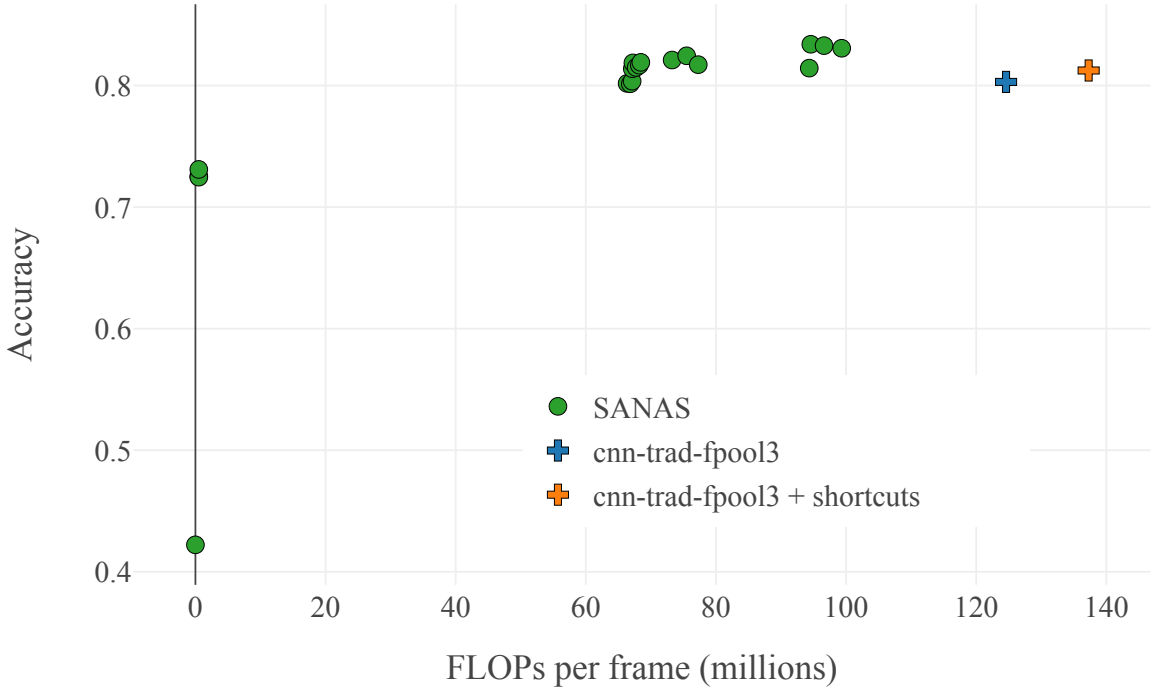


Figure 4.9. – **Cost accuracy curves.** Reported results are computed on the test set using models selected by computing the Pareto front over the validation set. Each point represents a model.

$[10^{-(m+1)}, 10^{-(m-1)}]$ with m the order of magnitude of the cost of the full model. Training time is reasonable and corresponds to about 1 day on a single GPU computer.

4.4.3 Results

Quantitative Results

The first results obtained by various models are presented in Table 4.1 for the one-hour test stream, and in Figure 4.9 on the test evaluation set. It can be seen that, at a given level of accuracy, the SANAS approach is able to greatly reduce the number of FLOPs, resulting in a model which is much more power-efficient.

For example, with an average cost of 37.7M FLOPs per frame, our model is able to match 86.5% of the words, (80.7% correctly and 5.8% wrongly) while the baseline models match 81.7% and 82.9% of the words with 72.8% and 77.9%

correct predictions while having a higher budget of 124.6M and 137.3 FLOPs per frame respectively.

Moreover, it is interesting to see that our model also outperforms both baselines in terms of accuracy, or regarding the metrics in Table 4.1. This is because, knowing that we have added shortcut connections in the base architecture, our model has better expressive power. Note that in our case, over-fitting is avoided by the cost minimization term in the objective function, while it occurs when using the complete architecture with shortcuts (see Figure 4.9).

Qualitative Results

To better understand what happens in SANAS. We measure how and when most of the budget is spent. Figure 4.10 illustrates the average cost per possible prediction during training. It is not surprising to show that our model automatically 'decides' to spend less time on frames containing background noise and much more time on frames containing words. Moreover, at convergence, the model also divides its budget differently on the different words, for example spending less time on the *yes* words that are easy to detect.

Figure 4.11 uses the data creation procedure explained in Figure 4.7 to see whether the amount of "true" signal present in a frame correlates with the share of the budget attributed to the frame. To do so we plot the budget against the share of a frame that is occupied by the real word (e.g., the budget allocated to frame B of Figure 4.7 would be plotted at $x \approx 0.3$ while frame C would be plotted at $x \approx 0.9$). The observed curves show that there is a strong correlation between the amount of real signal present in a frame and the time that SANAS decides to spend on it, confirming that the model behaves as expected and truly learns to look at the signal before selecting the architectures.

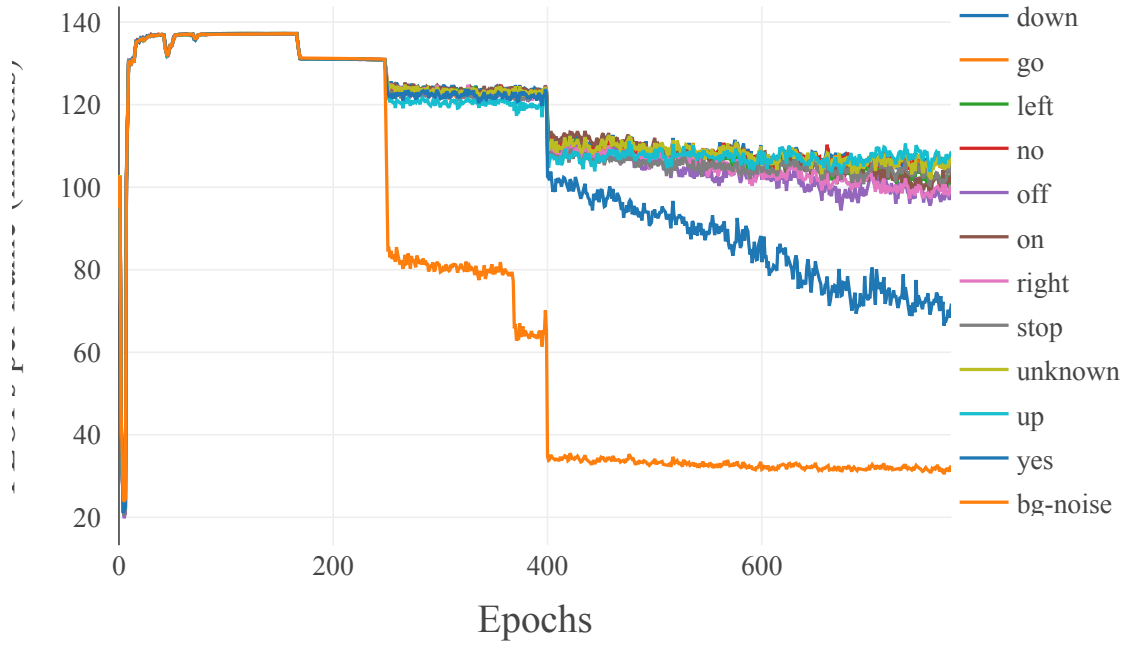


Figure 4.10. – **Training dynamics.** Average cost of the architecture sampled for each word (on the test set). The training dynamics show us that the network first finds an architecture able to solve the task while sampling notably cheaper architectures when only background noise is present in the frames (note the difference between the 11 curves for the classes containing actual words and the one for background noise). When we continue training past this point, forcing the network to save more energy in order further decrease the loss, we notice that the network starts making differences between words, first sampling notably cheapest architecture for words from the dataset less susceptible to be mixed up ("yes" or "left" for example) and keeping high cost for closer words ("no", "go", "on", "down") or the more complex class "unknown". The last part shows that the network is able to close the gap between the costs of the architectures used for the different "categories" of words. These dynamics illustrate the advantage of our method over "hard-coded" usage of several networks of different costs used in cascade or selected via some gating function. Y-axis is FLOPs per frame (millions)

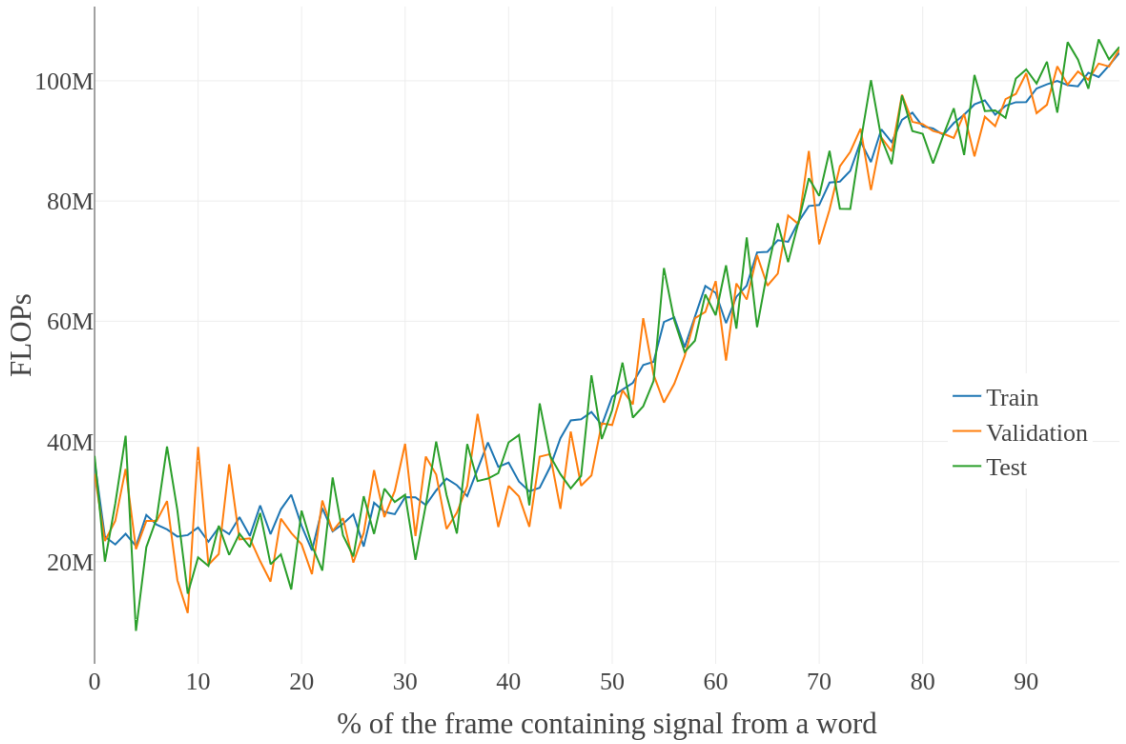


Figure 4.11. – Average number of FLOPs required to evaluate the sampled models in function of the signal contained in the corresponding frame. We observe that model is able to effectively adapt its size, sampling simple models when the frame contains only background noise and leveraging more computation power when the frame contains an actual word.

4.5 Conclusion

In this chapter, we have proposed a new model for keyword spotting where the recurrent network can automatically adapt its size during inference depending on the difficulty of the prediction problem at time t. This model is learned end-to-end based on a trade-off between prediction efficiency and computation cost and is able to find solutions that keep a high prediction accuracy while minimizing the average computation cost per timestep. Ongoing research includes using these methods on larger super-networks and investigating other types of budgets like memory footprint or electricity consumption on connected devices.

CHAPTER 5

NAS WITH ADAPTIVE SEARCH-SPACES FOR LIFELONG LEARNING

Contents

Chapter abstract

Existing literature in budgeted Neural Architecture Search (NAS) has mainly focused on properties of the learner such as its size and speed, overlooking other considerations like the amount of data available and whether external knowledge could be incorporated in the search procedure to reach better performances. Although such setups are currently under-explored, we remark that it bears some similarities with the very active domain of Continual Learning (CL), in which a learner faces a stream of tasks. In this setting, the learner is allowed to experience each task only once before being evaluated on all of

them. We propose to consider each task as a local NAS problem. The main idea is that, instead of starting from scratch every time as usually done, the learner can use its past experiences to improve the search procedure on subsequent tasks. For example, it could add pre-trained layers from past tasks to the current search space to save time and/or reach better performance on highly data-restricted tasks. Since no current benchmarks in CL focus on this transfer ability, we first propose a new suite of benchmarks to probe CL algorithms on carefully selected transfer scenarios. Equipped with this tool, we introduce a new modular NAS algorithm, whose modules represent atomic skills that can be composed to perform a certain task. Learning a task reduces to figuring out which past modules to re-use, and which new modules to instantiate to solve the current task. Our experiments show that this modular architecture and learning algorithm perform competitively on widely used CL benchmarks while yielding superior performance on the more challenging benchmarks we introduce in this work.

The work in this chapter has led to the acceptation of a conference paper to appear in May this year:

- Tom Veniat, Ludovic Denoyer, and Marc'Aurelio Ranzato (2021). "Efficient Continual Learning with Modular Networks and Task-Driven Priors". In: *9th International Conference on Learning Representations, ICLR 2021* abs/2012.12631. arXiv: 2012.12631. URL: https://arxiv.org/abs/2012.12631.

Resources to reproduce the work in this chapter are publicly available:

- CTrL benchmark: https://github.com/facebookresearch/CTrLBenchmark.
- Source code of the experiments: https://github.com/TomVeniat/MNTDP.

5.1 Introduction

After observing that NAS can improve the design of architectures on a single task, being able to greatly reduce the cost of inference models both on static (Chapter 3) and dynamic (Chapter 4) problems. One can wonder if NAS could operate using a new kind of constraints, not coming from inference infrastructure restrictions but coming from the task itself. For example, we can compare the standard ways used in supervised learning to solve small problems, only coming with a handful of annotated examples vs. large-scale tasks coming with thousand or even millions of examples. A seasoned machine learning practitioner facing

the first problem would generally use small neural networks with a carefully tuned architecture and training procedure (duration, which parameters to train, regularization, ...) to prevent over-fitting, they would also spend a considerable amount of time searching for models already trained on similar tasks, hoping that it will contain some related knowledge transferable to the current problem when finetuned. Their approach to the second problem would be very different, most probably using massive models having millions to billions of free parameters and training them from scratch for a long period of time. It would be interesting to see whether an automatic architecture search procedure could be extended to this level of complexity. Not only solving local problems but also operating at a "meta" level, automatically deciding which search space to consider or which task we could try benefiting from. The result of such an algorithm would be a large "system" evolving over time, adding more and more knowledge as it encounters new tasks and able to scale efficiently in order to exploit this knowledge when necessary to solve new problems.

This is the general problem we tackle in this last chapter. The first step is to note that, although not pursuing a strictly identical objective, there is some commonality between this problem and the CL framework – presented in details in Section 5.2. Indeed, both settings consider a single agent facing a theoretically never-ending stream of tasks. In both settings, the agent is required to carry knowledge across the stream to perform well. In the case of a continual learner, this is because it will be evaluated on the same task again at some point. In the case of the system described above it is to help the learning procedure and improve the performance on later tasks. In both settings, the learner should scale memory and compute sub-linearly w.r.t. the number of tasks in order to be usable in real-world scenarios.

As a first step in this direction, we therefore decide to focus on using NAS approaches on the CL problem. Existing literature in CL has focused on overcoming catastrophic forgetting (i.e., the inability of the learner to recall how to perform tasks observed in the past) and not on maximizing transfer across tasks. We first discuss the current methods used to evaluate CL models in Section 5.3. Starting from the observation that these methods fall short of measuring the specific properties we are interested in, we formally define them in Section 5.3.1 and propose a new benchmark based on real-world datasets able to evaluate any model along these axes in Section 5.3.2. Then, we present a new model of NAS-based CL in Section 5.5. This modular model is specifically designed to leverage knowledge acquired during past experiences to help the learning procedure by finding the best combination of pretrained and blank modules. We present the general framework as well as two implementations of the search algorithm. Our learning algorithm also leverages a task-driven prior over the exponential search space of all possible ways to combine modules, enabling efficient learning on

long streams of tasks. Finally, we demonstrate that it significantly outperforms existing approaches on most of these dimensions in the experiments presented in Section 5.6.

5.2 Continual Learning Background

Continual Learning (CL) is a learning framework whereby an agent learns through a sequence of tasks (Ring, 1994; Thrun, 1994; Thrun, 1998), observing each task once and only once. Much of the focus of the CL literature has been on avoiding *catastrophic forgetting* (McClelland et al., 1995; McCloskey and Cohen, 1989; Goodfellow et al., 2013), the inability of the learner to recall how to perform a task learned in the past. In our view, remembering how to perform a previous task is particularly important because it promotes knowledge accrual and transfer. CL has then the potential to address one of the major limitations of modern machine learning: its reliance on large amounts of labeled data. An agent may learn well a new task even when provided with little labeled data if it can leverage the knowledge accrued while learning previous tasks, assuming the current task bears some similarity to previously encountered tasks.

Continual Learning methods can be categorized into three main families of approaches:

Regularization

Methods falling in this category use a single shared predictor across all tasks with the only exception that there can be a task-specific classification head depending on the setting. They rely on various regularization methods to prevent forgetting. Kirkpatrick et al. (2016) and Schwarz et al. (2018) use an approximation of the Fisher Information matrix while Zenke et al. (2017) using the distance of each weight to its initialization as a measure of importance. These approaches work well in streams containing a limited number of tasks but will inevitably either forget or stop learning as streams grow in size and diversity (Ven and Tolias, 2019), due to their structural rigidity and fixed capacity.

Memory

Similarly, **rehearsal** based methods also share a single predictor across all tasks but attack forgetting by using rehearsal on samples from past tasks. For instance, Lopez-Paz and Ranzato (2017), Chaudhry et al. (2019a), and Rolnick et al. (2019)

store past samples in a replay buffer, while Shin et al. (2017) learn to generate new samples from the data distribution of previous tasks and Zhang et al. (2019) computes per-class prototypes. These methods share the same drawback of regularization methods: Their capacity is fixed and pre-determined which makes them ineffective at handling long streams.

Dynamic Architectures

Finally, approaches based on **evolving architectures** directly tackle the issue of the limited capacity by enabling the architecture to grow over time.

Within this family, there is a subclass of approaches that are *modular*, in the sense that the architecture is allowed to grow by adding modules to layers as opposed to individual neurons, and at run time only a sparse subset of modules is executed. This leads to much more efficient inference and RAM utilization, but learning is usually harder because of the combinatorial nature of the optimization problem. Our approach belongs to this sub-class.

Rusu et al. (2016) introduce a new network on each task, with connection to all previous layers, resulting in a network that grows super-linearly with the number of tasks. This issue was later addressed by Schwarz et al. (2018) who propose to distill the new network back to the original one after each task, henceforth yielding a fixed capacity predictor which is going to have severe limitations on long streams. Yoon et al. (2018) and Hung et al. (2019) propose a heuristic algorithm to automatically add and prune weights. Li et al. (2019) propose to softly select between reusing, adapting, and introducing a new module at every layer. Similarly, Xu and Zhu (2018) propose to add filters once a new task arrives using REINFORCE (Williams, 1992), leading to larger and larger networks even at inference time as time goes by. These two works are the most similar to ours, with the major difference that we restrict the search space over architectures, enabling much better scaling to longer streams. While their search space (and RAM consumption) grows over time, ours is constant. Our approach is *modular*, and only a small (and constant) number of modules is employed for any given task both at training and test time. Non-modular approaches, like those relying on individual neuron gating (Adel et al., 2020; Serrà et al., 2018; Kessler et al., 2019), lack such runtime efficiency which limits their applicability to long streams. Rajasegaran et al. (2019) and Fernando et al. (2017) propose to learn a modular architecture, where each task is identified by a path in a graph of modules like we do. However, they lack the prior over the search space. They both use random search which is rather inefficient as the number of modules grows.

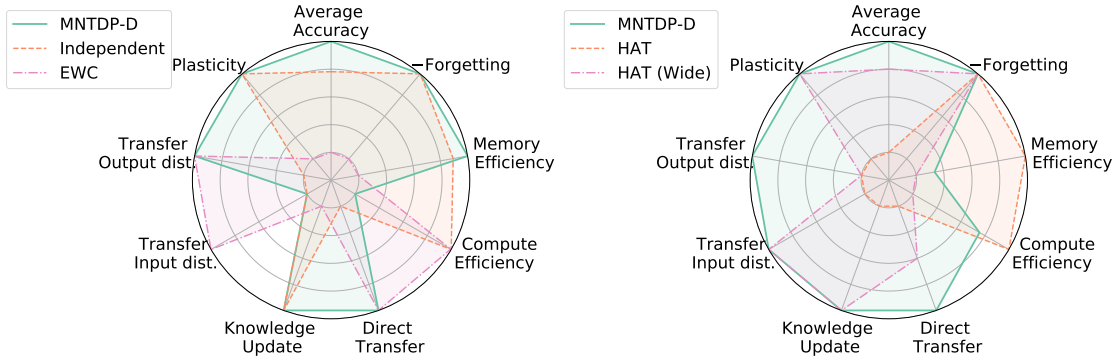


Figure 5.1. – Comparison of various CL methods on the CTrL benchmark using Resnet (left) and Alexnet (right) backbones. MNTDP-D is our method. See Table 5.7 of Section 5.6.3 for details.

There are of course other works that introduce new benchmarks for CL. Most recently, Wortsman et al. (2020) have proposed a stream with 2500 tasks all derived from MNIST permutations. Unfortunately, this may provide little insight in terms of how well models transfer knowledge across tasks. Other benchmarks like CORe50 (Lomonaco and Maltoni, 2017) and CUB-200 (Wah et al., 2011) are more realistic but do not enable precise assessment of how well models transfer and scale.

CL is also related to other learning paradigms, such as meta-learning (Finn et al., 2017b; Nichol et al., 2018b; Duan et al., 2016), but these only consider the problem of quickly adapting to a new task while in CL we are also interested in preventing forgetting and learning better over time. For instance, Alet et al. (2018) proposed a modular approach for robotic applications. However, only the performance on the last task was measured. There is also a body of literature on modular networks for multi-task and multi-domain learning (Ruder et al., 2019; Rebuffi et al., 2017; Zhao et al., 2020). The major differences are the static nature of the learning problem they consider and the lack of emphasis on scaling to a large number of tasks.

5.3 Evaluating Continual Learning Models

Let us start with a general formalization of the CL framework. We assume that tasks arrive in sequence and that each task is associated with an integer task descriptor $t = 1, 2, ...$ which corresponds to the order of the tasks. Task descriptors are provided to the learner both during training and test time. Each task is defined by a labeled dataset $\mathcal{D}^t$. We denote with $\mathcal{S}$ a sequence of such tasks. A predictor for a given task t is denoted by $f : \mathcal{X}^t \times \mathbb{Z} \rightarrow \mathcal{Y}^t$. The predictor has internally

some trainable parameters whose values depend on the stream of tasks $\mathcal{S}$ seen in the past, therefore the prediction is: $f(x, t|\mathcal{S})$. Notice that in general, different streams lead to different predictors for the same task: $f(x, t|\mathcal{S}) \neq f(x, t|\mathcal{S}')$.

5.3.1 Desirable Properties of CL models and Metrics:

Since we are focusing on supervised learning tasks, it is natural to evaluate models in terms of accuracy. We denote the prediction accuracy of the predictor f as $\Delta(f(x, t|\mathcal{S}), y)$, where x is the input, t is the task descriptor of x, $\mathcal{S}$ is the stream of tasks seen by the learner and y is the ground truth label.

In this context, we consider four major properties of a CL algorithm:

Average Accuracy First, the algorithm has to yield predictors that are accurate by the end of the learning experience. This is measured by their **average accuracy** at the end of the learning experience:

$$\mathcal{A}(\mathcal{S}) = \frac{1}{T}\sum_{t=1}^{T} \mathbb{E}_{(x,y)\sim\mathcal{D}^t}[\Delta(f(x, t|\mathcal{S} = 1, \ldots, T), y)]. \tag{5.1}$$

Forgetting Second, the CL algorithm should yield predictors that do not forget, i.e. that are able to perform a task seen in the past without significant loss of accuracy. **Forgetting** is defined as:

$$\mathcal{F}(\mathcal{S}) = \frac{1}{T-1}\sum_{t=1}^{T-1} \mathbb{E}_{(x,y)\sim\mathcal{D}^t}[\Delta(f(x, t|\mathcal{S} = 1, \ldots, T), y) - \Delta(f(x, t|\mathcal{S}' = 1, \ldots, t), y)]. \tag{5.2}$$

This measure of forgetting has been called backward transfer (Lopez-Paz and Ranzato, 2017), and it measures the average loss of accuracy on a task at the end of training compared to when the task was just learned. Negative values indicate the model has been forgetting. Positive values indicate that the model has been improving on past tasks by learning subsequent tasks.

Transfer Third, the CL algorithm should yield predictors that are capable of transferring knowledge from past tasks when solving a new task. **Transfer** can be measured by:

$$\mathcal{T}(\mathcal{S}) = \mathbb{E}_{(x,y)\sim\mathcal{D}^T}[\Delta(f(x, T|\mathcal{S} = 1, \ldots, T), y) - \Delta(f(x, T|\mathcal{S}' = T), y)]. \tag{5.3}$$

which measures the difference of performance between a model that has learned through a whole sequence of tasks and a model that has learned the last task in isolation. We would expect this quantity to be positive if there exist previous tasks that are related to the last task. Negative values imply the model has suffered some form of interference or even lack of plasticity when the predictor has too little capacity left to learn the new task.

Scalability The CL algorithm also has to yield predictors that **scale sub-linearly** with the number of tasks both in terms of memory and compute. In order to quantify this, we simply report the total memory usage and compute by the end of the learning experience during training. We, therefore, include in the memory consumption everything a learner has to keep around to be able to continually learn (e.g., regularization parameters of EWC or the replay buffer for experience replay).

Learning Speed Finally, we report the area under the learning curve after β optimization steps as proposed in Chaudhry et al. (2019b) as a metric to assess learning speed:

$$\text{LCA@}\beta = \frac{1}{T}\sum_{t=1}^{T}\left[\frac{1}{\beta+1}\sum_{b=0}^{\beta} A_{b,t}(f)\right]. \tag{5.4}$$

Where $A_{b,t}(f)$ corresponds to the test set accuracy on task t of learner f after having seen b batches from task t. While interesting, this metric isn't essential to our evaluation since we focus on this chapter on obtaining the best possible model for each task. Therefore, we only present it in the detailed results of each stream while omitting it from summary tables for clarity.

5.3.2 Streams

The metrics introduced above can be applied to any stream of tasks. While current benchmarks are constructed to assess forgetting, they fall short at enabling a comprehensive evaluation of *transfer* and *scalability* because they do not control for task relatedness and they are composed of too few tasks. Therefore, we propose a new suite of streams. If t is a task in the stream, we denote with t^- and t^+ a task whose data is sampled from the same distribution as t, but with a much smaller or larger labeled dataset, respectively. Finally, t' and t'' are tasks that are similar to task t, while we assume no relation between t_i and t_j, for all $i \neq j$.

We consider five axes of transfer and define a stream for each of them. While other dimensions certainly exist, here we are focusing on basic properties that any desirable model should possess.

Direct Transfer

We define the stream $\mathcal{S}^{\text{-}} = (t_1^+, t_2, t_3, t_4, t_5, t_1^-)$ where the last task is the same as the first one but with much less data to learn from. This is useful to assess whether the learner can remember an experience and directly transfer knowledge from the relevant task without having to retrain anything.

A classic real-world example of such a scenario is how we learn to ride a bicycle: the first experience of it is long and painful, it requires lots of trials to reach the required skills of balance and coordination (lots of data). Yet once this skill is acquired, it only takes a few minutes to ride again even without any practice for several years.

Knowledge Update

We define the stream $\mathcal{S}^{+} = (t_1^-, t_2, t_3, t_4, t_5, t_1^+)$ where the last task has much more data than the first task with intermediate tasks that are unrelated. In this case, there should not be much need to transfer anything from previous tasks, and the system can just use the last task to update its knowledge of the first task.

Transfer to similar Input Distributions

In order to evaluate the ability to recognize high-level similarities, we consider the case where two tasks are very similar but with very different input distributions. For example, different tasks could aim at classifying the same objects but using black and white vs color photos, or the same signal could be observed in different tasks, but each time observed by different captors each having different features and noise levels. We define this streams with the last task similar to the first task but the input distribution changes $\mathcal{S}^{\text{in}} = (t_1, t_2, t_3, t_4, t_5, t_1')$.

Transfer to similar Output Distributions

On the opposite, we also define a streams where the last task is similar to the first task but the with the same input distribution and a changed output distribution: $\mathcal{S}^{\text{out}} = (t_1, t_2, t_3, t_4, t_5, t_1'')$.

This corresponds to the setting in which standard transfer learning usually shines. When interested in a task for which few annotated samples are available, a good approach is to first pre-train on a large task using the same modality and then fine-tuning on the desired task (e.g., classification to segmentation, translation to sentiment classification).

Plasticity

To assess plasticity, we use a stream where all tasks are unrelated, $\mathcal{S}^{\text{pl}} = (t_1, t_2, t_3, t_4, t_5)$, which is useful to measure the "ability to still learn" and potential interference (erroneous transfer from unrelated tasks) when learning the last task.

One important thing to note is that approaches using fixed architectures are doomed to fail on this dimension. Indeed, if an almost saturated learner faces a task of a totally new kind, it will be unable to solve it without major alterations to the accumulated knowledge. This dimension is critical for truly lifelong learners having to deal with a never-ending stream of tasks.

Scalability

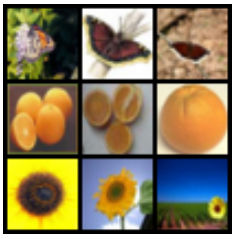
Finally, we evaluate scalability using $\mathcal{S}^{\text{long}}$, a stream with 100 tasks of varying degrees of relatedness and with varying amounts of training data. See Section 5.4 for more details.

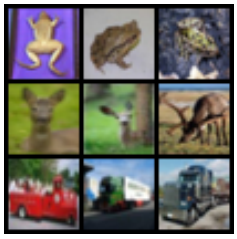
All these tasks are evaluated using $\mathcal{T}$ in Equation 5.3. Other dimensions of transfer (e.g., transfer with compositional task descriptors or under noisy conditions) are avenues of future work.

5.4 The CTrL Benchmark

The Continual Transfer Learning (CTrL) benchmark is a collection of streams of tasks built over seven popular computer vision datasets, namely:

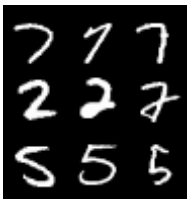
- CIFAR10 and CIFAR100 (Krizhevsky, 2009a), which have already been presented in Chapter 3, Section 3.5.2.1.
- MNIST (LeCun et al., 1998), also presented in Section 3.5.2.1.
- Describable Textures Dataset (DTD) (Cimpoi et al., 2014), composed of images of 40 different textures. This dataset is interesting because while also com-


(a) Cifar100


(b) Cifar10


(c) DTD


(d) SVHN


(e) MNIST


(f) Fashion-MNIST


(g) Rainbow-MNIST

Figure 5.2. – Samples from the datasets used in the Continual Transfer Learning (CTrL) benchmark.

Dataset	no. classes	training	validation	testing
CIFAR-100	100	40000	10000	10000
CIFAR-10	10	40000	10000	10000
D. Textures	47	1880	1880	1880
SVHN	10	47217	26040	26032
MNIST	10	50000	10000	10000
Fashion-MNIST	10	50000	10000	10000
Rainbow-MNIST	10	50000	10000	10000

Table 5.1. – Datasets used in the CTrL benchmark.

posed of pictures, its semantic is far from natural images. it is also relatively small with only 40 training examples per class.

- Street View House Number (SVHN) (Netzer et al., 2011) which is also composed of pictures but this time of house numbers. The objective being to recognize digits in the wild, this problem is very similar to MNIST but using a different input domain.

- Rainbow MNIST (R-MNIST) is a variant of the Rainbow-MNIST dataset proposed by Finn et al. (2019), using only different background colors and keeping the original scale and rotation of the digits.

- Fashion MNIST (F-MNIST) (Xiao et al., 2017) is similar in shape and size to MNIST, intending to classify clothes rather than digits.

		T_1	T_2	T_3	T_4	T_5	T_6
$\mathcal{S}^-$	**Datasets**	Cifar-10	MNIST	DTD	F-MNIST	SVHN	Cifar-10
	# Train Samples	4000	400	400	400	400	400
	# Val Samples	2000	200	200	200	200	200
$\mathcal{S}^+$	**Datasets**	Cifar-10	MNIST	DTD	F-MNIST	SVHN	Cifar-10
	# Train Samples	400	400	400	400	400	4000
	# Val Samples	200	200	200	200	200	2000
$\mathcal{S}^{\text{in}}$	**Datasets**	R-MNIST	Cifar-10	DTD	F-MNIST	SVHN	R-MNIST
	# Train Samples	4000	400	400	400	400	50
	# Val Samples	2000	200	200	200	200	30
$\mathcal{S}^{\text{out}}$	**Datasets**	Cifar-10	MNIST	DTD	F-MNIST	SVHN	Cifar-10
	# Train Samples	4000	400	400	400	400	400
	# Val Samples	2000	200	200	200	200	200
$\mathcal{S}^{\text{pl}}$	**Datasets**	MNIST	DTD	F-MNIST	SVHN	Cifar-10	
	# Train Samples	400	400	400	400	4000	
	# Val Samples	200	200	200	200	2000	

Table 5.2. – Details of the streams used to evaluate the transfer properties of the learner. The provided number of samples is per class.

Please, refer to Table 5.1 for basic statistics about the datasets.

These datasets are desirable because they are diverse (hence tasks derived from some of these datasets can be considered unrelated), they have a fairly large number of training examples to simulate tasks that do not need to transfer, and they have low spatial resolution enabling fast evaluation.

CTrL is designed according to the methodology described in Section 5.3.2, to enable evaluation of the various transfer learning properties presented above and the ability of models to scale to a large number of tasks.

The composition of the different tasks is given in Table 5.2 and an instance of the long stream is presented in Appendix A.

The tasks in $\mathcal{S}^-$, $\mathcal{S}^+$, $\mathcal{S}^{\text{in}}$, $\mathcal{S}^{\text{out}}$ and $\mathcal{S}^{\text{pl}}$ are all 10-way classification tasks.

In $\mathcal{S}^-$, the first task has 4000 training examples while the last one which is the same as the first task has only 400. The vice versa is true for $\mathcal{S}^+$ instead.

The last task of $\mathcal{S}^{\text{in}}$ is the same as the first task, except that the background color of the MNIST digit is different. The last task of $\mathcal{S}^{\text{out}}$ is the same as the first task, except for label ids that have been shuffled, therefore, if "horse" was associated to label id 3 in the first task, it is now associated to label id 5 in the last task. Special care is taken to avoid that any class is associated to the same label id in the first and last tasks.

The $\mathcal{S}^{\text{long}}$ stream is composed of both *large* and *small* 5-way classification tasks that have 5000 (or whatever is the maximum available) and 25 training examples, respectively. Each task is built by choosing one of the datasets at random, and 5 categories at random in this dataset. During task 1-33, the fraction of small tasks is 50%, this increases to 75% for tasks 34-66, and to 100% for tasks 67-100. This schedule is designed to simulate the desired evolution of a lifelong learning agent. Indeed, when freshly released in a new environment without any prior on the tasks it faces, a significant amount of data is required to reach an acceptable performance. On the contrary, as the number of tasks solved increases, the agent should rely more and more on the prior knowledge accumulated over time, therefore requiring orders of magnitude less data toward the end of the stream. This is a challenging setting allowing to assess not only scalability but also transfer ability and sample efficiency of a learner. Full details about one version of the $\mathcal{S}^{long}$ stream used in our experiments are presented in Section A.1.

Each task comes with a training, validation, and test datasets. Since the learner has access to both the training and validation sets, the number of validation samples is always around half the size of the training set. On the contrary, since the test set is only used for reporting results and never for training, it contains the maximum number of samples available to allow the best evaluation possible.

All images have been rescaled to 32x32 pixels in RGB color format, and per-channel normalized using statistics computed on the training set of each task. During training, we perform data augmentation by using random crops (4 pixels padding and 32x32 crops) and random horizontal reflection.

5.5 Modular Networks with Task Driven Prior

Now that the desired properties of a continual learning agent have been identified and can be evaluated, we can come back to the objective discussed in Section 5.1.

The overall idea is to decompose the ever-growing machine learning system into two main components:

- A growing strategy, in charge of maintaining the ability of the system to keep learning as the number of encountered tasks increase and archiving existing knowledge for future usage once a problem is solved. A poorly balanced growing strategy would greatly reduce the practical interest of such a system. Indeed, if not enough capacity is added over time the system will quickly saturate and stop learning. On the other end of the spectrum, adding too much capacity and/or archiving unnecessary information will result in an inefficient bloated system growing out of control and unable to retrieve relevant information from what it decided to archive. The performance of this component can be measured using $\mathcal{S}^{\text{pl}}$ and $\mathcal{S}^{\text{long}}$.

- A search algorithm, in charge of selecting what and how much knowledge should be transferred when facing a new task. Here again, the balance is difficult to find since transferring too much knowledge to an unrelated task or, on the contrary, not transferring anything to a task already solved could prevent the system from reaching the optimal performance. The performance of this component can be measured using $\mathcal{S}^{+}$, $\mathcal{S}^{-}$, $\mathcal{S}^{\text{in}}$ and $\mathcal{S}^{\text{out}}$.

The proposed approach, dubbed Modular Networks with Task Driven Prior (MNTDP), is designed following this general idea. Its growing strategy is presented in Section 5.5.1 and its NAS-based search algorithm is described in Section 5.5.2. The Task-Driven Prior, an additional component whose objective is to further improve the scalability of the system is presented in Section 5.5.3.

In this model, the class of predictor functions $f(x, t|\mathcal{S})$ we consider is *modular*, in the sense that predictors are composed of modules that are potentially shared across different (but related) tasks. A module can be any parametric function, for instance, a ResNet block (He et al., 2016). The only restriction we use is that all modules in a layer should differ only in terms of their actual parameter values, while modules across layers can implement different classes of functions. For instance, in the illustrations of Figure 5.3-A and Figure 5.4e (see captions for details), there are two predictors, each composed of three modules (all ResNet blocks), the first one being shared.

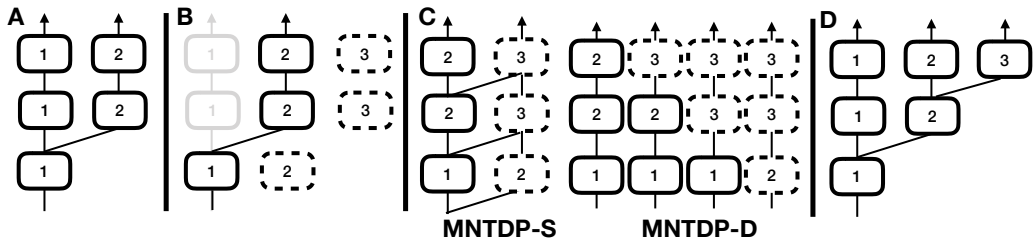


Figure 5.3. – Toy illustration of the approach when each predictor is composed of only three modules and only two tasks have already been observed. **A)**: The predictor of the first task uses modules (1,1,1) (listing modules by increasing depth in the network) while the predictor of the second task uses modules (1,2,2); the first layer module is shared between the two predictors. **B)**: When a new task arrives, first we add one new randomly initialized module at each layer (the dashed modules). Second, we search for the most similar past task and retain only the corresponding architecture. In this case, the second task is most similar and therefore we remove (gray out) the modules used only by the predictor of the first task. **C)**: We train on the current task by learning both the best way to combine modules and their parameters. However, we restrict the search space. In this case, we only consider four possible compositions, all derived by perturbing the predictor of the second task. In the stochastic version (MNTDP-S), for every input a path (sequence of modules) is selected stochastically. Notice that the same module may contribute to multiple paths (e.g., the top-most layer with id 3). In the deterministic version instead (MNTDP-D), we train in parallel all paths and then select the best. Note that only the parameters of the newly added (dashed) modules are subject to learning. **D)**: Assuming that the best architecture found at the previous step is (1,2,3), module 3 at the top layer is added to the current library of modules.

5.5.1 Growing Strategy

Once a new task t arrives, the algorithm follows three steps.

First, it temporarily adds new randomly initialized modules at every layer (these are denoted by blue edges in Figure 5.4). The graph composed of existing modules and the new added ones defines a search space over all possible ways to combine modules (Figure 5.4a, Figure 5.4c and Figure 5.4f)).

Second, it makes use of the search algorithm to minimize a loss function, which in our case is the cross-entropy loss. This step learns ways to combine modules and module parameters at the same time, see Figure 5.4d. Note that only the parameters of the newly added modules are subject to training, de facto preventing forgetting of previous tasks by construction but also preventing positive backward transfer. This step is described in more detail in Section 5.5.2.

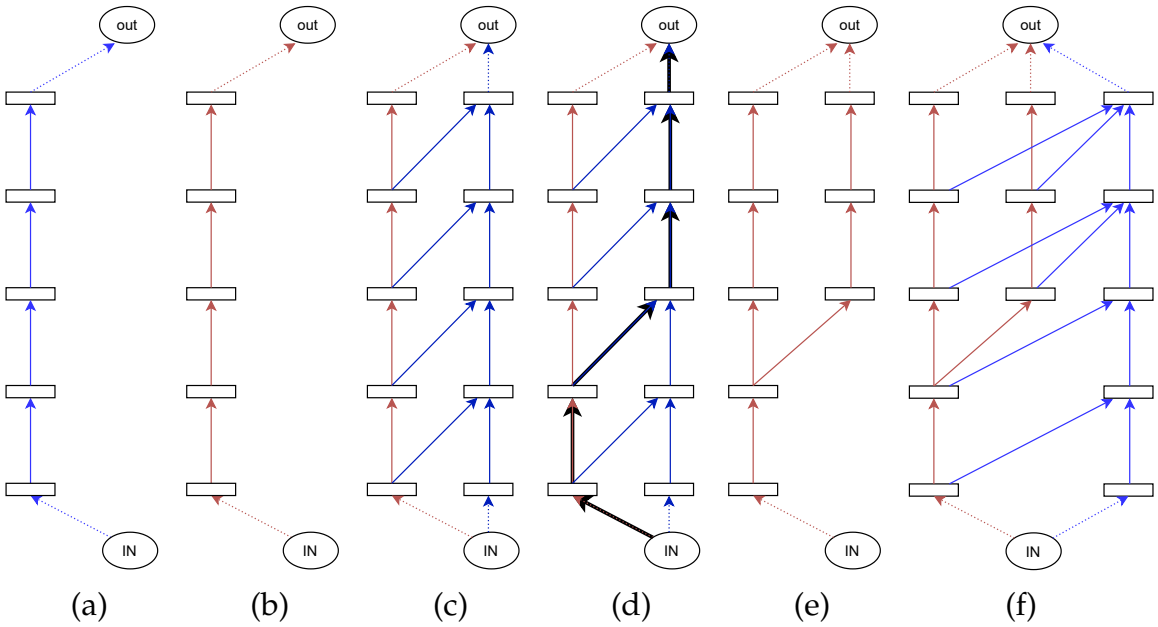


Figure 5.4. – Expansion and pruning strategy of MNTDP. Blue edges correspond to trainable modules while red edges are frozen. (a) is the initial step, all modules are randomly initialized and then fully trained on the first task. Once the training is finished, newly introduces edges are frozen (b) before moving to the next task and expanding the network to create the new search space (c). Starting from task 2, MNTDP learns at the same time the optimal path and the corresponding parameters that needs to be retained (d). When the path is identified, selected new modules are frozen and unused modules are discarded (e). The model can now expand again and repeat the procedure for the next task (f).

Finally, it takes the resulting predictor for the current task and adds the parameters of the (selected) newly added modules (if any) back to the existing library of module parameters, see Figure 5.4b and Figure 5.4e, discarding modules added during the first step but not selected at the second step.

5.5.2 Search procedure

Since predictors are uniquely identified by which modules compose them, they can also be described by the *path* in the grid of module parameters. We denote the j-th path in the graph by π_j. The parameters of the modules in path π_j are denoted by $\theta(\pi_j)$. Note that in general $\theta(\pi_j) \cap \theta(\pi_i) \neq \emptyset$, for $i \neq j$ since some modules may be shared across two different paths.

Let Π be the set of all possible paths in the graph. This has a size equal to the product of the number of modules at every layer, after adding the new randomly initialized modules. If Γ is a distribution over Π which is subject to learning (and initialized uniformly), then the loss function is:

$$\Gamma^*, \theta^* = \arg\min_{\theta,\Gamma} \mathbb{E}_{j\sim\Gamma,(x,y)\sim\mathcal{D}^t} \mathcal{L}(f(x,t|\mathcal{S},\theta(\pi_j)),y) \tag{5.5}$$

where $f(x,t|\mathcal{S},\theta(\pi_j)$ is the predictor using parameters $\theta(\pi_j)$, $\mathcal{L}$ is the loss, and the minimization over the parameters is limited to only the newly introduced modules. The resulting distribution Γ^* is a delta distribution, assuming no ties between paths. Once the best path has been found and its parameters have been learned, the corresponding parameters of the new modules in the optimal path are added to the existing set of modules while the other ones are disregarded (Figure 5.4).

We propose now two instances of the learning problem in Equation 5.5, which differ in a) how they optimize over paths and b) how they share parameters across modules.

Stochastic version:

Comparing Equation 5.5 and Equation 3.10, we observe that the problem we are now facing is very similar to the one solved by the Budgeted Super Networks (BSN) algorithm. The only difference being the absence of the explicit cost constraint. We can therefore take inspiration from what we did in Chapter 3 to propose an algorithm solving the problem at hand.

This algorithm alternates between one step of gradient descent over the paths via REINFORCE (Williams, 1992) as in Chapters 3 and 4, and one step of gradient descent over the parameters for a given path. The distribution Γ is now modeled by a product of multinomial distributions, one for each layer of the model. These select one module at each layer, ultimately determining a particular path. Newly added modules may be shared across several paths which yields a model that can support several predictors while retaining a very parsimonious

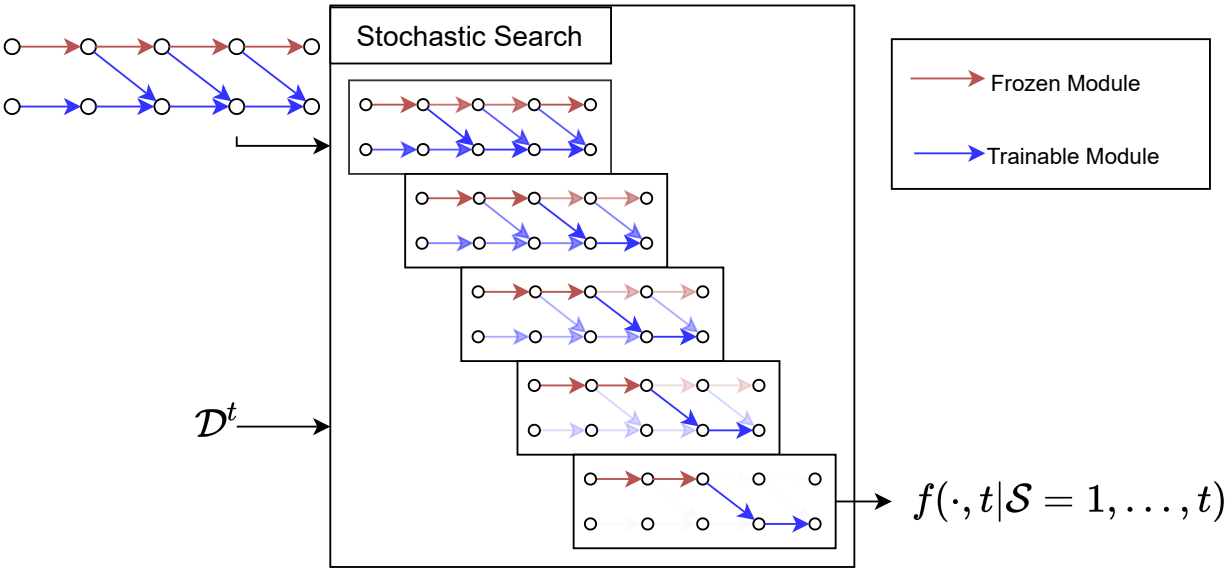


Figure 5.5. – Stochastic version of MNTDP search algorithm. A distribution over architectures and the corresponding parameters are learned jointly using the NAS procedure presented in Chapter 3.

memory footprint thanks to parameter sharing. This version of the model, dubbed MNTDP-Stochastic (MNTDP-S), is outlined in Figure 5.5 and in algorithm 3 (as well as in the left part of Figure 5.3-C). To encourage the model to explore multiple paths, we use an entropy regularization on Γ during training.

The alternation of gradient steps over paths and layer parameters is an addition to the method presented in previous chapters, where both sets of parameters were jointly updated on each batch. The objective is to avoid overfitting on small tasks.

To implement this strategy, we split the training set into two halves $\mathcal{D}_1^t$ and $\mathcal{D}_2^t$. The first part is used to update the module parameters θ, while the second is used to update the parameters in the distribution over paths, Γ. If both sets of parameters were trained on the same dataset, Γ would favor paths prone to overfitting since they will results in a large decrease of its training loss and therefore a larger reward. When they are trained on different sets, Γ has to select paths with a reasonable amount of free parameters, allowing θ to learn and generalize enough to decrease the loss on both training sets. Then, the most promising architecture is selected based on $\arg\max \Gamma$, and fine-tuned over the whole $\mathcal{D}^t$. In this case, the validation set is used only for hyper-parameters selection.

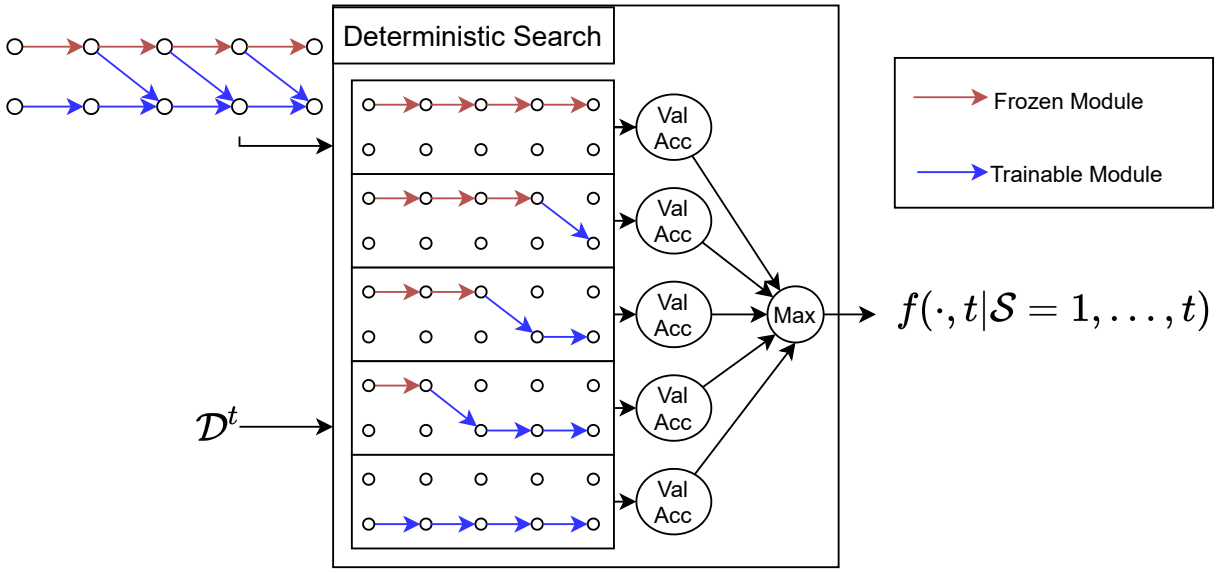


Figure 5.6. – Deterministic version of MNTDP search algorithm.

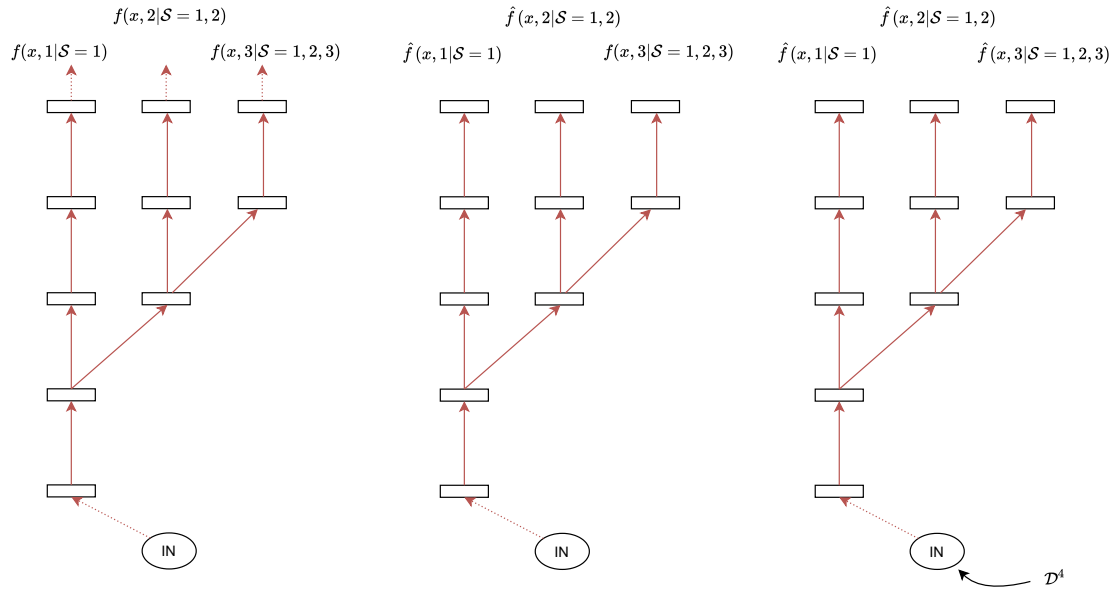
Deterministic version:

This algorithm, dubbed MNTDP-Deteministic (MNTDP-D) minimizes the objective over paths in Equation 5.5 via exhaustive search, see algorithm 4. It is illustrated in the right part of Figure 5.3-C and a detailed version in Figure 5.6. Here, paths do not share any newly added module and we train one independent network per path, and then select the path yielding the lowest loss on the validation set.

While this requires much more memory, it may also lead to better overall performance because each new module is cloned and trained just for a single path. Moreover, training predictors on each path can be trivially and cheaply parallelized on modern GPU devices. All the architectures are trained over the training set, and the best path is retained based on its score on the validation set.

5.5.3 Scaling with Data-Driven Prior

Unfortunately, the algorithms as described above do not scale to a a large number of tasks (and henceforth modules) because the search space grows exponentially. This is also the case for other evolving architecture approaches proposed in the literature (Li et al., 2019; Rajasegaran et al., 2019; Yoon et al., 2018; Xu and Zhu, 2018) as discussed in Section 5.2. We thus propose to reduce the search space, hence enabling lower memory and computational footprint.



(a) (b) (c)

Figure 5.7. – **Data-Driven Priors** are used to restrict the search space. When learning on a new task, (a) we consider classifiers obtained so far. (b) More specifically, we consider the feature space in which the classification layers operate. (c) Finally, we feed the current training set through this set of feature extractors and run K-Nearest Neighbors (KNN) to identify which feature space has a structure allowing to perform the best on the current task.

If there were N modules per layer and L layers, the search space would have size $\mathcal{O}(N^L)$. As illustrated in Figures 5.3 and 5.4, we already restrict the search space to paths that branch to the right: A newly added module at layer l can only connect to another newly added module at layer $l + 1$, but it cannot connect to an already trained module at layer $l + 1$. The underlying assumption is that for most tasks we expect changes in the output distribution as opposed to the input distribution, and therefore if tasks are related, the base trunk is a good candidate for being shared. We will see in Section 5.6.4 what happens when this assumption is not satisfied, e.g., when applying this to $\mathcal{S}^{\text{in}}$.

To further restrict the search space we employ a *data-driven prior*. The intuition is to limit the search space to perturbations of the path corresponding to the past task (or to the top-k paths) that is the most similar to the current task. There are several methods to assess which task is the closest to the current task without accessing data from past tasks and also different ways to perturb a path. We propose a very simple approach, but others could have been used.

On task t, we take the predictors from all the past tasks (Figure 5.7a)

$$f(x, 1|\mathcal{S} = 1)$$
$$\dots$$
$$f(x, t-1|\mathcal{S} = 1, \dots, t-1)$$

and consider the feature space in which their classification layers operate. The feature space of $f(x, 1|\mathcal{S} = 1)$ is denoted $\hat{f}(x, 1|\mathcal{S} = 1)$ and is obtained by removing the head the corresponding classifier (Figure 5.7b). The last step is to feed the training set $\mathcal{D}^t$ of t to these feature extractors, yielding a set of features for each past predictor(Figure 5.7c). The k paths whose feature space yields the best nearest neighbors classification accuracy are then selected to branch from and the other will not be considered when learning the architecture for task t. The intuition is that if features from the predictor of task t_i allow reaching a high accuracy using KNN on task t_j, it means that the feature space structure corresponds well to both these tasks and therefore t_i and t_j must have some level of semantic similarity.

This process is shown in Figure 5.3-B. The search space is reduced from T^L to only L, and Γ of Equation 5.5 is allowed to have non-zero support only in this restricted search space, yielding a much lower computational and memory footprint which is *constant* with respect to the number of tasks. The designer of the model has now direct control (for instance, by varying k) over the trade-off between accuracy and computational/memory budget.

As long as the restricted search space contains the optimal path or paths close to the optimal, we should expect the modular network to satisfy all our desired properties: by construction, the model does not forget, because we do not update modules of previous tasks. The model can transfer well because it can re-use modules from related tasks encountered in the past while not being constrained in terms of its capacity. And finally, the model scales sub-linearly in the number of tasks because modules can be shared across similar tasks. We will validate empirically in Section 5.6.3 whether the choice of the restricted search space works well in practice.

Moreover, the designer of the algorithm can carefully tune the appropriate search space, leveraging application-specific priors as well as any constraints coming from their computational or memory budget. For instance, when very limited compute and memory are available, the designer may decide to further limit the search space to paths that contain only a few new randomly initialized modules as opposed to L.

Algorithm 3: MNTDP-S algorithm.

Data: Dataset of task t: $\mathcal{D}^t$.;

Past predictors: $f(x, j|\mathcal{S})$ for $j = 1, \ldots, t-1$;

Find closest task: $j^* = \arg\max_j \mathbb{E}_{(x,y)\sim\mathcal{D}^t}[\Delta(\text{NN}(f(x, j|\mathcal{S})), y)]$, where NN is the 5-nearest neighbor classifier in feature space;

Define search space: Take path corresponding to predictor of task j^* and add a new randomly initialized module at every layer. Γ: distribution over paths; π_i: i-th path;

Split train set in two halves: $\mathcal{D}_1^t$ and $\mathcal{D}_2^t$;

while *loss in eq. 5.5 has not converged* **do**

  get sample $(x, y) \sim \mathcal{D}^t[iteration \bmod 2]$;

  sample path $\pi_k \sim \Gamma$;

  if *odd iteration* ***or*** $\max\Gamma > 0.99$ **then**

    forward/backward and update $\theta(\pi_k)$ (only newly added modules)

  else

    forward/backward and update Γ;

  end

end

Let i^* be the path with largest values in Γ, then set $f(x, t|\mathcal{S} \cup t)$ to π_{i^*}.

Algorithm 4: MNTDP-D algorithm.

Data: Dataset of task t: $\mathcal{D}^t$.;

Past predictors: $f(x, j|\mathcal{S})$ for $j = 1, \ldots, t-1$;

Find closest task: $j^* = \arg\max_j \mathbb{E}_{(x,y)\sim\mathcal{D}^t}[\Delta(\text{NN}(f(x, j|\mathcal{S})), y)]$, where NN is 5-nearest neighbor classifier in feature space;

Define search space: Take path corresponding to predictor of task j^* and add a new randomly initialized module at every layer. π_i: i-th path, $i = 1, \ldots, N$ where N is the total number of paths;

for $i = 1, \ldots, N$ **do**

  while *loss in eq. 5.5 has not converged* **do**

    get sample $(x, y) \sim \mathcal{D}^t$;

    forward/backward, update parameters $\theta(\pi_i)$ (only newly added modules)

  end

  compute accuracy A_i on validation set.

end

Let $i^* = \arg\max_i A_i$, then set $f(x, t|\mathcal{S} \cup t)$ to π_{i^*}.

5.6 Experiments

In this section we first present the modeling details in Section 5.6.1, then we compare the complexity of different approaches in Section 5.6.2 before reporting their results on standard benchmarks as well as on CTrL in Section 5.6.3.

5.6.1 Methodology and Modeling Details

Models learn over each task in sequence; data from each task can be replayed several times but *each stream is observed only once*. Since each task has a validation dataset, hyper-parameters (e.g., learning rate and number of weight updates) are task-specific and they are cross-validated on the validation set of each task. Once the learning experience ends, we test the resulting predictor on the test sets of all the tasks. Notice that this is a stricter paradigm than what is usually employed in the literature (Kirkpatrick et al., 2016), where hyper-parameters are set at the stream level (by replaying the stream several times). Our model selection criterion is more realistic because it does not assume that the learner has access to future tasks when cross-validating on the current task, and this is more consistent with the CL's assumptions of operating on a stream of data.

All models use the same backbone. Unless otherwise specified, this backbone is a small fully-connected network divided into 3 blocks for the Permuted-MNIST task and a small variant of the ResNet-18 architecture which is divided into 7 module. Please, refer to Table 5.3 for further details about the specifics of the architectures, with Table 5.3a for the model used on Permuted-MNIST stream and Table 5.3b and Table 5.3c for all the other tasks.

Each predictor is trained by minimizing the cross-entropy loss with a small L2 weight decay on the parameters. In our experiments, MNTDP adds 7 new randomly initialized modules, one for every block. The search space does not allow connecting old blocks from new blocks, and it considers two scenarios: using old blocks from the past task that is deemed most similar ($k = 1$, the default setting) or considering the whole set of old blocks ($k =$ all) resulting in a much larger search space.

As presented in Section 5.6.1, the stream can be visited only once, preventing stream-level hyper-parameters tuning. Exceptions are made for HAT (Serrà et al., 2018) because we have been using authors' implementation, and for EWC and Online-EWC since these approaches fail in the proposed setting. The constraint strength hyper-parameter λ must be tuned at the stream level since a task-level tuning of λ results in little or no constraint at all, leading to severe catastrophic forgetting. The stream-level hyper-parameters optimization considers 9 values for

Block	# layers	#params	# hidden units
1	1	785000	1000
2	1	1001000	1000
3	1	10010	num. classes

(a) Permuted MNIST Model

Block	# layers	#params	# out channels
1	1	1856	64
2	4	147968	64
3	4	152192	64
4	4	152192	64
5	2	78208	64
6	2	73984	64
7	1	650	num. classes

(b) Resnet architecture used throughout our experiments.

Block	# layers	#params	# out channels/hidden units
1	1	3136	64
2	1	73856	128
3	1	131328	256
4	1	2099200	2048
5	1	4196352	2048
7	1	10245	num. classes

(c) Details of the Alexnet architecture used in Serrà et al. (2018) and in our experiments.

Table 5.3. – Descriptions of all the base architectures used in our experiments. Details of how many layers and parameters each block contains are only relevant for modular approaches.

λ {1, 5, 10, 50, 100, 500, 10^3, 5×10^3, 10^4}. Note that this gives an unfair advantage to EWC, Online-EWC, and HAT, as all other methods including MNTDP use task-level cross-validation as described in Section 5.6.1.

For all methods and experiments, we use the Adam optimizer (Kingma and Ba, 2015) with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

For each task and each baseline, two learning rates $\{10^{-2}, 10^{-3}\}$ and 3 weight decay strengths $\{0, 10^{-5}, 10^{-4}\}$ are considered. Early stopping is performed on each task to identify the best step to stop training. When the current task validation accuracy stops increasing for 300 iterations, we restore the learner to its state after the best iteration and stop training on the current task.

For MNTDP-S, we consider two additional learning rates for the Γ optimization $\{10^{-2}, 10^{-3}\}$. An entropy regularization term on Γ is added to the loss to encourage

exploration, preventing an early convergence towards a sub-optimal path. The weight for this regularization term is set to 1 throughout our experiments

Finally, since small tasks in $\mathcal{S}^{\text{long}}$ have very few examples in the validation sets, we use test-time augmentation to prevent overfitting during the grid search. For each validation sample, we add four augmented copies following the same data augmentation procedure used during training.

We compare to several baselines:

1. **Independent Models** which instantiates a randomly initialized predictor for every task (as many paths as tasks without any module overlap).
2. **Finetuning** which trains a single model to solve all the tasks without any regularization and initializes from the model of the previous task (a single path shared across all tasks)
3. **New-Head** which also shares the trunk of the network across all tasks but not the classification head which is task-specific
4. **New-Leg** which shares all layers across tasks except for the very first input layer which is task-specific
5. **Elastic Weight Consolidation (EWC)** (Kirkpatrick et al., 2016) which is like "finetuning" but with a regularizer to alleviate forgetting
6. **Experience Replay (ER)** (Chaudhry et al., 2019a) which is like finetuning except that the model has access to some samples from the past tasks to rehearse and alleviate forgetting (we use 15 samples per class to obtain a memory consumption similar to other baselines)
7. **Progressive Neural Networks (PNN)** (Rusu et al., 2016) which adds both a new module at every layer as well as lateral connections once a new task arrives.
8. **Hard Attention to Tasks (HAT)** (Serrà et al., 2018): which learns an attention mask over the parameters of the backbone network for each task. Since HAT's open-source implementation uses AlexNet (Krizhevsky et al., 2012) as a backbone, we also implemented a version of MNTDP using AlexNet for a fair comparison. Moreover, we considered two versions of HAT, the default as provided by the authors and a version, dubbed HAT-wide, that is as wide as our final MNTDP model (or as wide as we can fit into GPU memory).
9. **DEN** (Yoon et al., 2018) which grows the base network using a set of heuristics, adding neurons and tuning them when the desired performance can't be reached on the new task.
10. **RCL** (Xu and Zhu, 2018) which uses Reinforcement Learning to grow the architecture.

Model	Train complexity	Test complexity
Inde, fintune, new-head	N	N
EWC	$N + 2TN$	N
ER	$N + rT$	N
MNTDP-D	kbN	N
MNTDP-S	$N + 2kN$	N
HAT	N	N
Wide HAT	SN	SN
DEN	$N + pT$	$N + pT$
RCL	$N + pT$	$N + pT$
Lean to Grow	$N + 2Tp$	N

Table 5.4. – Memory complexity of the different baselines at train time and at test time, where N is the size of the backbone, T the number of tasks, r the size of the memory buffer per task, k the number of source columns used by MNTDP, b the number of blocks in the backbone, S the scale factor used for wide-HAT and p the average number of new parameters introduced per task. Note that while Wide HAT and MNTDP-D are using a similar amount of memory on CTrL (Table 5.7), the inference model used by MNTDP on each task only uses the memory of the narrow backbone, resulting in more than 6 times smaller inference models.

11. Finally, both versions of **MNTDP** (Stochastic and Deterministic) defining the search space by taking the most similar previous task (i.e.,$k = 1$) and by adding a new module at every block depth, unless otherwise specified.

For baselines 3 (new-head) and 4 (new-leg), we propose two variants. In the "freeze" variant the modules shared across tasks are not updated after the first task, while in the "finetune" version they are updated throughout the whole stream. For baselines 9 (Dynamically Expandable Networks (DEN)) and 10 (Reinforced Continual Learning (RCL)), since there is no publicly available implementation using Convolutional Neural Network (ConvNet), we only report their performance on Permuted-MNNIST using Multi-Layer Perceptron (MLP) modules (see Figure 5.8 and Table 5.5).

We report performance across different axes as discussed in Section 5.3: Average accuracy as in Equation 5.1, forgetting as in Equation 5.2, transfer as in Equation 5.3 and applicable only to the transfer streams, "Mem." [MB] which refers to the average memory consumed by the end of the training, and "Flops" [T] which corresponds to the average amount of computation used by the end of training.

5.6.2 Comparison of Memory Complexities

In the next sections, one of the metrics we will track is the overall memory consumption of our MNTDP model to compare it to different approaches from the literature. We can however already propose a theoretical analysis of the complexity of growth of each method. This comparison is presented in Table 5.4. It is important to make the distinction between the training time and the test-time complexity since the training environment will usually be a large cluster while the deployment environment for a given task can be orders of magnitude smaller.

First, we observe that while most methods require more space during training, all methods but DEN and RCL are similarly light to deploy and therefore fast to evaluate at test time. This is because modular architectures like MNTDP use only a sparse subset of modules for any given task: they select a single "path", reducing to the independent network case. Another property of MNTDP is that, benefiting from the task-driven prior, the training complexity doesn't depend on the number of tasks. It is a very desirable property for models aimed at learning on very long streams and this is what will allow us to train it on the S^{long} stream.

5.6.3 Results on Existing Benchmarks

In Figure 5.8 and Figure 5.9 we compare MNTDP against several baselines on two standard streams with 10 tasks, Permuted MNIST, and Split CIFAR100. We observe that all models do fairly well, with EWC falling a bit behind the others in terms of average accuracy and finetuning methods which exhibit strong forgetting. PNNs has good average accuracy but requires more memory and compute. Compared to MNTDP, both RCL and HAT have lower average accuracy and require more compute. MNTDP-D yields the best average accuracy, but requires more computation than "independent models"; notice however that its wall-clock training time is actually the same as "independent models" since all candidate paths (seven in our case) can be trained in parallel on modern GPU devices. In fact, it turns out that on these standard streams MNTDP trivially reduces to "independent models" without any module sharing, since each task is fairly distinct and has a relatively large amount of data. It is therefore not possible to assess how well models can transfer knowledge across tasks, nor it is possible to assess how well models scale. Fortunately, we can leverage the CTrL benchmark to better study these properties, as described next.

Table 5.5 and Table 5.6 report performance across all axes of evaluation on the standard Permuted MNIST and Split CIFAR 100 streams.

Model	$< \mathcal{A} >$	$< \mathcal{F} >$	Mem.	FLOPs	LCA@5
Independent	0.98	0.00	71.8	9.0	0.43
Finetune	0.49	-0.49	7.2	12.0	0.13
New-head freeze	0.89	0.00	7.5	19.0	0.14
New-head finetune	0.55	-0.43	7.5	15.0	0.14
New-leg freeze	0.98	0.00	35.4	10.0	0.43
New-leg finetune	0.89	-0.09	35.4	14.0	0.19
EWC †	0.94	-0.03	143.7	18.0	0.13
Online EWC †	0.95	-0.01	21.6	14.0	0.16
ER (Reservoir) †	0.69	-0.29	15.0	12.0	0.29
ER	0.90	-0.08	15.0	15.0	0.29
PNN	0.98	0.00	253.8	123.0	0.15
MNTDP-S	0.97	0.00	71.8	21.0	0.22
MNTDP-S (k=all)	0.98	0.00	71.8	24.0	0.19
MNTDP-D	0.98	0.00	71.8	56.0	0.37
HAT†	0.95	0.00	7.6	25.0	0.11
HAT (Wide)†	0.97	0.00	78.5	209.0	0.13
DEN †	0.95	0.00	8.1	-	-
RCL †	0.96	0.00	8.5	148.2	-

Table 5.5. – Results on the standard permuted-MNIST stream. In this stream, each of the 10 tasks corresponds to a random permutation of the input pixels of MNIST digits. For DEN and RCL, since we are using the authors' implementations, we do not have access to the LCA measure. † corresponds to models using stream-level cross-validation (see Section 5.6.1).

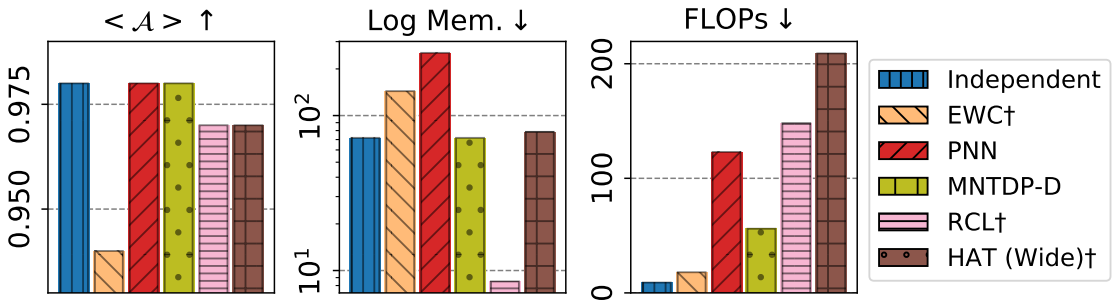


Figure 5.8. – Results on Permuted-MNIST. † correspond to models cross-validated at the stream-level, a setting that favors them over the other methods which are cross-validated at the task-level.

Model	$< \mathcal{A} >$	$< \mathcal{F} >$	Mem.	FLOPs	LCA@5
Independent	0.82	0.00	24.4	1327.0	0.11
Finetune	0.18	-0.56	2.4	886.0	0.15
New-head freeze	0.57	0.00	2.5	740.0	0.21
New-head finetune	0.21	-0.52	2.5	687.0	0.15
New-leg freeze	0.50	0.00	2.5	1759.0	0.12
New-leg finetune	0.18	-0.42	2.5	1115.0	0.11
EWC †	0.55	0.01	51.0	1151.0	0.11
Online EWC †	0.54	-0.02	7.3	1053.0	0.12
ER (Reservoir) †	0.32	-0.50	20.9	1742.0	0.17
ER	0.66	-0.15	20.9	1524.0	0.17
PNN	0.78	0.00	133.8	9889.0	0.15
MNTDP-S	0.75	0.00	24.4	1295.0	0.11
MNTDP-S (k=all)	0.75	0.00	24.4	1323.0	0.11
MNTDP-D	0.83	0.00	24.3	6168.0	0.12
MNTDP-D*	0.81	0.00	260.9	488.0	0.17
HAT*†	0.74	-0.01	27.0	175.0	0.11
HAT (Wide)*†	0.79	0.00	269.3	1830.0	0.11

Table 5.6. – Results on the standard Split Cifar 100 stream. Each task is composed of 10 new classes.* corresponds to models using an Alexnet backbone, † to models using stream-level cross-validation (see Section 5.6.1).

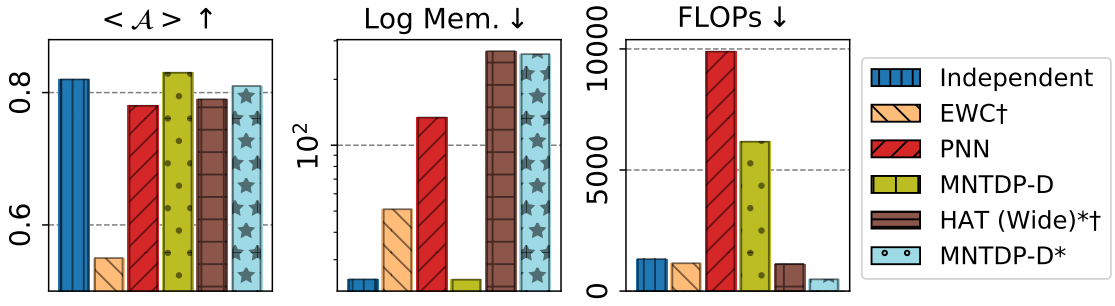


Figure 5.9. – Results on Split Cifar-100. * denotes an Alexnet Backbone. † correspond to models cross-validated at the stream-level.

5.6.4 Results on CTrL

We first evaluate models in terms of their ability to transfer by evaluating them on the streams $\mathcal{S}^{-}, \mathcal{S}^{+}, \mathcal{S}^{\text{in}}, \mathcal{S}^{\text{out}}$ and $\mathcal{S}^{\text{pl}}$ introduced in Section 5.3.2. Table 5.7 shows that "independent models" is again a strong baseline, because even on the first four streams, all tasks except the last one are unrelated and therefore instantiating an independent model is optimal. However, MNTDP yields the best average accuracy overall. MNTDP-D achieves the best transfer on streams $\mathcal{S}^{-}, \mathcal{S}^{+}, \mathcal{S}^{\text{out}}$ and $\mathcal{S}^{\text{pl}}$, and indeed it discovers the correct path in each of these cases (e.g., it discovers to reuse the path of the first task when learning on $\mathcal{S}^{-}$ and to just swap the top modules when learning the last task on $\mathcal{S}^{+}$). Examples of the discovered paths are presented at the end of this chapter (Section 5.6.6). MNTDP underperforms on $\mathcal{S}^{\text{in}}$ because its prior does not match the data distribution, since in this case, it is the input distribution that has changed but swapping the first module is out of MNTDP search space. This highlights the importance of the choice of prior for this algorithm. In general, MNTDP offers a clear trade-off between accuracy, i.e. how broad the prior is which determines how many paths can be evaluated, and memory/compute budget. Computationally MNTDP-D is the most demanding, but in practice its wall clock time is comparable to "independent" because GPU devices can store in memory all the paths (in our case, seven) and efficiently train them all in parallel. We observe also that MNTDP-S has a clear advantage in terms of computing at the expense of a lower overall average accuracy, as sharing modules across paths during training can lead to sub-optimal convergence.

Overall, MNTDP has a much higher average accuracy than methods with a fixed capacity. It also beats PNNs, which seems to struggle with interference when learning on $\mathcal{S}^{\text{pl}}$, as all new modules connect to all old modules which are irrelevant for the last task. Moreover, PNNs use much more memory. "New-leg" and "new-head" models perform very well only when the tasks in the stream match their assumptions, showing the advantage of the adaptive prior of MNTDP. Finally, EWC shows great transfer on $\mathcal{S}^{-}, \mathcal{S}^{\text{in}}, \mathcal{S}^{\text{out}}$, which probe the ability of the model to retain information. However, it fails at $\mathcal{S}^{+}, \mathcal{S}^{\text{pl}}$ that require additional capacity allocation to learn a new task. Figure 5.1 gives a holistic view by reporting the normalized performance across all these dimensions.

Figures 5.1 and 5.10 provide radar plots of the models evaluated on the CTrL benchmark. The companion tables of these plots are Tables 5.9, 5.10, 5.11, 5.12 and 5.13.

	$< \mathcal{A} >$	$< \mathcal{F} >$	Mem.	FLOPs	$\mathcal{T}(\mathcal{S}^{-})$	$\mathcal{T}(\mathcal{S}^{+})$	$\mathcal{T}(\mathcal{S}^{\text{in}})$	$\mathcal{T}(\mathcal{S}^{\text{out}})$	$\mathcal{T}(\mathcal{S}^{\text{pl}})$
Independent	0.58	**0.0**	14.1	308	0.0	**0.0**	0.0	0.0	**0.0**
Finetune	0.19	-0.3	**2.4**	**284**	0.0	-0.1	-0.0	-0.0	-0.1
New-head	0.48	**0.0**	2.5	307	**0.4**	-0.3	-0.2	**0.3**	-0.4
New-leg	0.41	**0.0**	2.5	366	0.3	-0.3	**0.4**	-0.1	-0.4
Online EWC †	0.43	-0.1	7.3	310	0.3	-0.3	0.3	**0.3**	-0.4
ER	0.44	-0.1	13.1	604	0.0	-0.2	0.0	0.1	-0.2
PNN	0.57	**0.0**	48.2	1459	0.3	-0.2	0.1	0.2	-0.1
MNTDP-S	0.59	**0.0**	11.7	363	**0.4**	-0.1	0.0	**0.3**	-0.1
MNTDP-D	**0.64**	**0.0**	11.6	1512	**0.4**	**0.0**	0.0	**0.3**	**-0.0**
MNTDP-D*	0.62	0.0	140.7	115	0.3	-0.1	0.1	0.3	-0.1
HAT*†	0.58	-0.0	26.6	45	0.1	-0.2	0.0	0.1	-0.2
HAT (Wide)*†	0.61	0.0	163.9	274	0.2	-0.1	0.1	0.1	-0.1

Table 5.7. – Aggregated results on the transfer streams over multiple relevant baselines (complete tables with more baselines are provided the next 5 sections. * corresponds to models using an Alexnet backbone, † to models using stream-level cross-validation (see Section 5.6.1).

	$< \mathcal{A} >$	$< \mathcal{F} >$	Mem.	PFLOPs
Independent	0.57	**0.0**	243	**4**
Finetune	0.20	-0.4	**2**	5
New-head	0.43	**0.0**	3	6
On. EWC†	0.27	-0.3	7	**4**
MNTDP-S	0.68	**0.0**	159	5
MNTDP-D	**0.75**	**0.0**	102	26
MNTDP-D*	**0.75**	**0.0**	1782	3
HAT*†	0.24	-0.1	**32**	**≈0**
HAT*† (Wide)	0.32	**0.0**	285	1

Table 5.8. – Results on the long evaluation stream. * corresponds to models using an Alexnet backbone, † to models using stream-level cross-validation (see Section 5.6.1). See Table 5.14 for more baselines and error bars.

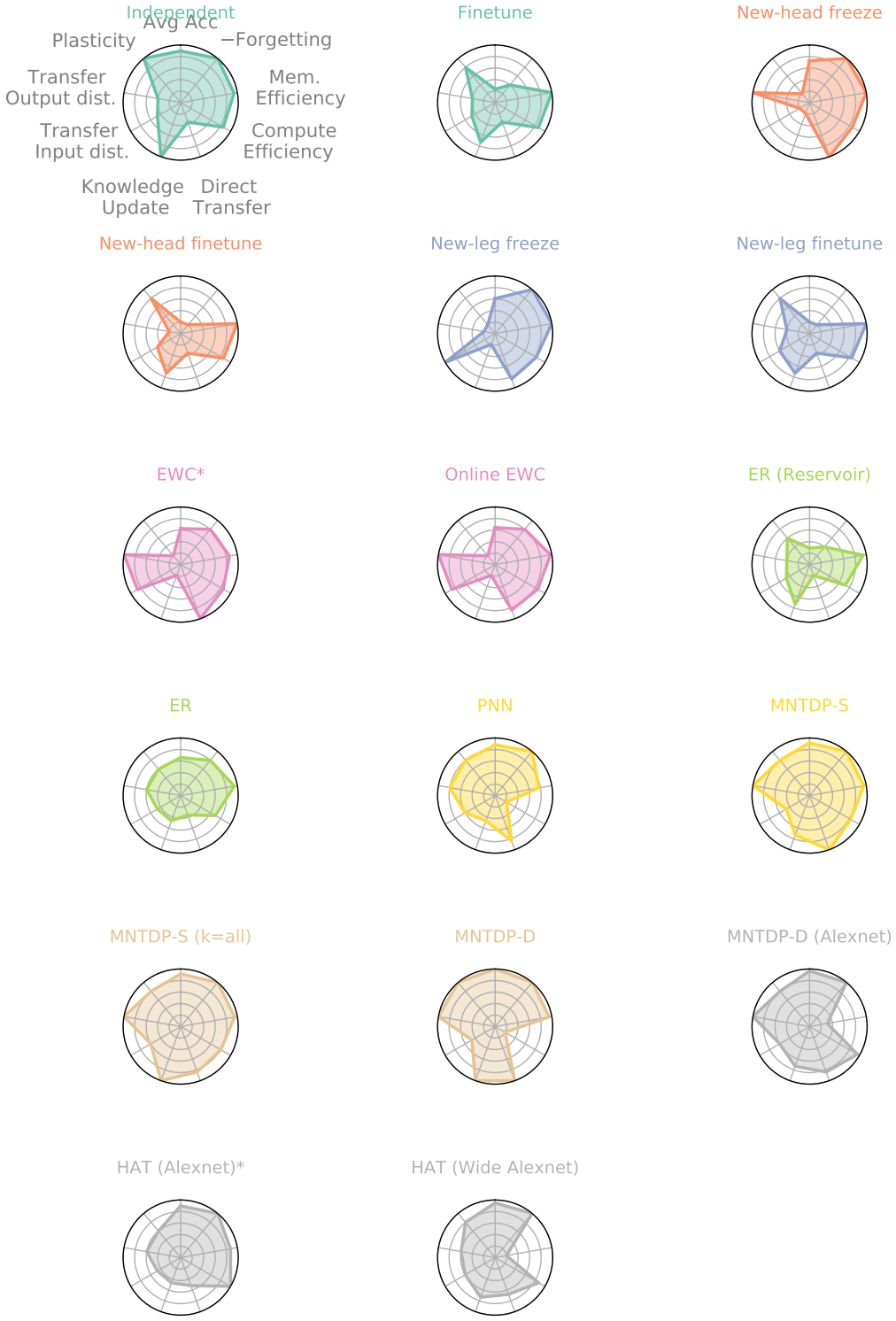


Figure 5.10. – Comparison of the global performance of all baselines on the CTrL Benchmark. MNTDP-D is the most efficient method on multiple of the dimensions, but it requires more computation than MNTDP-S.

Stream $\mathcal{S}^{-}$

Model	Acc T_1	Acc T_1'	$\Delta_{T_1,T_1'}$	$\mathcal{T}(\mathcal{S}^{-})$	$<\mathcal{A}>$	$<\mathcal{F}>$	Mem.	FLOPs	LCA@5
Independent	0.72	0.36	-0.35	0.00	0.56	0.00	14.6	292.0	0.10
Finetune	0.72	0.37	-0.34	0.01	0.18	-0.33	2.4	308.0	0.10
New-head freeze	0.72	0.71	-0.01	0.35	0.54	0.00	2.5	262.0	0.19
New-head finetune	0.72	0.35	-0.36	-0.01	0.15	-0.39	2.5	296.0	0.10
New-leg freeze	0.72	0.65	-0.07	0.29	0.47	0.00	2.5	362.0	0.11
New-leg finetune	0.72	0.33	-0.38	-0.03	0.13	-0.41	2.5	333.0	0.10
EWC †	0.72	0.71	-0.01	0.35	0.52	-0.02	31.5	344.0	0.13
Online EWC †	0.72	0.69	-0.03	0.33	0.54	-0.01	7.3	309.0	0.11
ER (Reservoir)†	0.72	0.30	-0.41	-0.06	0.20	-0.32	13.5	646.0	0.12
ER	0.72	0.36	-0.36	0.00	0.41	-0.13	13.5	551.0	0.11
PNN	0.72	0.65	-0.06	0.29	0.62	0.00	51.1	1099.0	0.13
MNTDP-S	0.72	0.71	0.00	0.35	0.63	0.00	11.0	310.0	0.10
MNTDP-S (k=all)	0.72	0.63	-0.09	0.27	0.61	0.00	10.7	341.0	0.10
MNTDP-D	0.72	0.72	0.00	0.36	0.67	0.00	9.2	1876.0	0.16
MNTDP-D*	0.64	0.64	0.00	0.28	0.63	0.00	130.2	101.0	0.21
HAT*†	0.61	0.42	-0.19	0.06	0.57	-0.01	26.6	54.0	0.12
HAT*† (Wide)	0.67	0.54	-0.13	0.18	0.60	0.00	164.0	257.0	0.14

Table 5.9. – Results in the $\mathcal{T}(\mathcal{S}^{-})$ evaluation stream. In this stream, the last task is the same as the first with an order of magnitude less data. * corresponds to models using an Alexnet backbone, † to models using stream-level cross-validation (see Section 5.6.1).

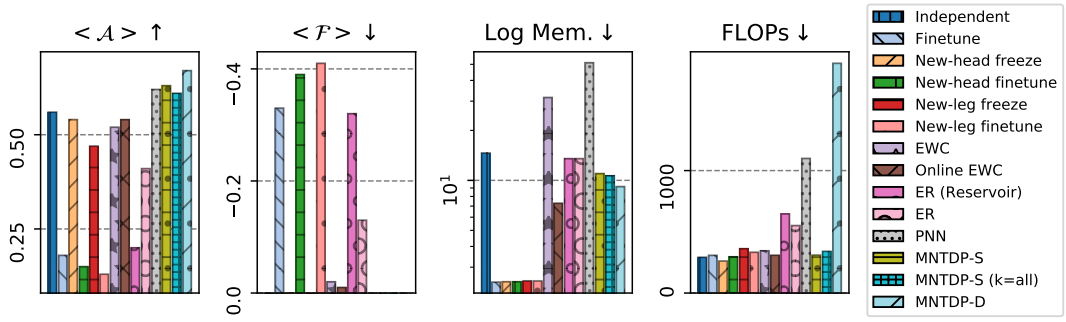


Figure 5.11. – Comparison of all baselines on the $\mathcal{S}^{-}$ stream.

Stream $\mathcal{S}^+$

Model	Acc T_1	Acc T_1'	$\Delta_{T_1,T_1'}$	$\mathcal{T}(\mathcal{S}^+)$	$<\mathcal{A}>$	$<\mathcal{F}>$	Mem.	FLOPs	LCA@5
Independent	0.37	0.71	0.35	0.00	0.57	0.00	14.6	404.0	0.10
Finetune	0.37	0.58	0.21	-0.13	0.24	-0.23	2.4	262.0	0.12
New-head freeze	0.37	0.43	0.06	-0.28	0.41	0.00	2.5	361.0	0.17
New-head finetune	0.37	0.57	0.20	-0.14	0.19	-0.36	2.5	327.0	0.11
New-leg freeze	0.37	0.37	0.01	-0.34	0.34	0.00	2.5	425.0	0.11
New-leg finetune	0.37	0.56	0.19	-0.15	0.17	-0.29	2.5	357.0	0.10
EWC †	0.37	0.42	0.05	-0.29	0.39	-0.02	31.5	337.0	0.11
Online EWC †	0.37	0.40	0.03	-0.31	0.37	-0.04	7.3	271.0	0.11
ER (Reservoir) †	0.37	0.56	0.20	-0.15	0.20	-0.33	13.5	493.0	0.11
ER	0.37	0.47	0.10	-0.24	0.43	-0.07	13.5	512.0	0.12
PNN	0.37	0.54	0.17	-0.17	0.52	0.00	51.1	1757.0	0.12
MNTDP-S	0.37	0.62	0.25	-0.09	0.56	0.00	14.0	417.0	0.10
MNTDP-S (k=all)	0.37	0.66	0.29	-0.05	0.56	0.00	14.0	595.0	0.10
MNTDP-D	0.37	0.72	0.35	0.01	0.61	0.00	14.0	1659.0	0.11
MNTDP-D*	0.40	0.64	0.24	-0.07	0.61	0.00	156.3	136.0	0.17
HAT*†	0.41	0.52	0.12	-0.19	0.57	0.00	26.6	39.0	0.13
HAT*† (Wide)	0.41	0.61	0.19	-0.10	0.59	0.00	164.0	324.0	0.14

Table 5.10. – Results in the $\mathcal{S}^+$ evaluation stream. In this stream, the 5th task is the same as the first with an order of magnitude more data. Tasks 2, 3, and 4 are distractors. * corresponds to models using an Alexnet backbone, † to models using stream-level cross-validation (see Section 5.6.1).

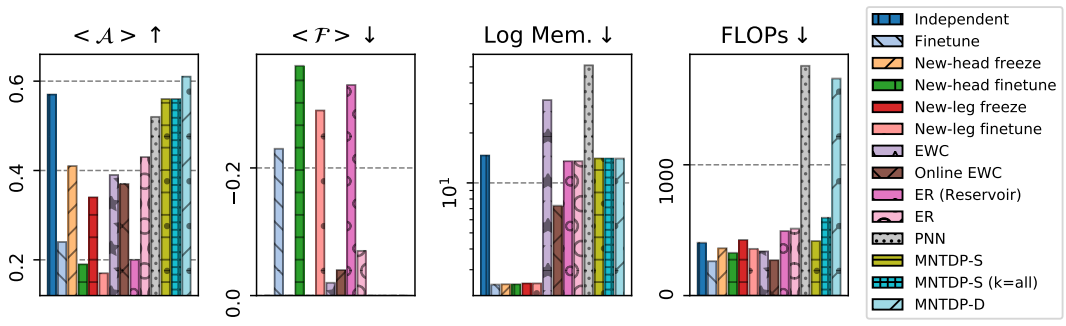


Figure 5.12. – Comparison of all baselines on the $\mathcal{S}^+$ stream

Stream $\mathcal{S}^{in}$

Model	Acc T_1	Acc T_1'	$\Delta_{T_1,T_1'}$	$\mathcal{T}(\mathcal{S}^{\text{in}})$	$<\mathcal{A}>$	$<\mathcal{F}>$	Mem.	FLOPs	LCA@5
Independent	0.98	0.60	-0.39	0.00	0.57	0.00	14.6	265.0	0.10
Finetune	0.98	0.57	-0.41	-0.03	0.18	-0.31	2.4	244.0	0.11
New-head freeze	0.98	0.39	-0.59	-0.21	0.45	0.00	2.5	246.0	0.13
New-head finetune	0.98	0.62	-0.36	0.02	0.19	-0.33	2.5	188.0	0.11
New-leg freeze	0.98	0.95	-0.04	0.35	0.48	0.00	2.5	282.0	0.10
New-leg finetune	0.98	0.70	-0.28	0.10	0.21	-0.34	2.5	238.0	0.10
EWC †	0.98	0.87	-0.12	0.27	0.43	-0.14	31.5	269.5	0.11
Online EWC †	0.98	0.87	-0.12	0.27	0.43	-0.12	7.3	287.0	0.10
ER (Reservoir) †	0.98	0.60	-0.38	0.00	0.30	-0.26	13.5	570.0	0.12
ER	0.98	0.60	-0.38	0.00	0.38	-0.17	13.5	604.0	0.12
PNN	0.98	0.70	-0.29	0.10	0.57	0.00	51.1	899.0	0.10
MNTDP-S	0.98	0.64	-0.34	0.04	0.57	0.00	12.2	333.0	0.10
MNTDP-S (k=all)	0.98	0.68	-0.30	0.08	0.57	0.00	13.4	327.0	0.10
MNTDP-D	0.98	0.62	-0.36	0.02	0.60	0.00	11.6	1225.0	0.11
MNTDP-D*	0.98	0.67	-0.31	0.07	0.59	0.00	156.6	115.0	0.12
HAT*†	0.98	0.61	-0.37	0.01	0.58	-0.01	26.6	41.0	0.12
HAT*† (Wide)	0.97	0.67	-0.30	0.07	0.62	0.00	164.0	186.0	0.11

Table 5.11. – Results in the transfer evaluation stream with input perturbation. In this stream, the last task is the same as the first one with a modification applied to the input space and with an order of magnitude less data. Tasks 2, 3, 4 and 5 are distractors. * corresponds to models using an Alexnet backbone, † to models using stream-level cross-validation (see Section 5.6.1).

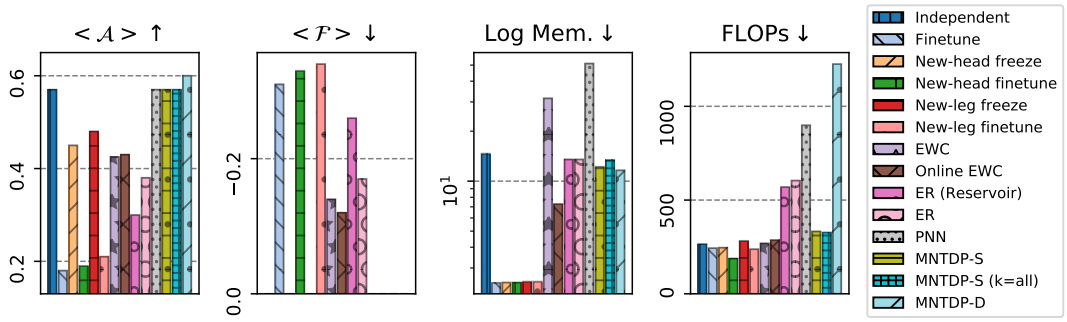


Figure 5.13. – Comparison of all baselines on the $\mathcal{S}^{\text{in}}$ stream

Stream $\mathcal{S}^{out}$

Model	Acc T_1	Acc T_1'	$\Delta_{T_1,T_1'}$	$\mathcal{T}(\mathcal{S}^{\text{out}})$	$<\mathcal{A}>$	$<\mathcal{F}>$	Mem.	FLOPs	LCA@5
Independent	0.70	0.37	-0.32	0.00	0.61	0.00	14.6	349.0	0.10
Finetune	0.70	0.36	-0.33	-0.01	0.15	-0.37	2.4	331.0	0.11
New-head freeze	0.70	0.70	0.01	0.33	0.54	0.00	2.5	374.0	0.19
New-head finetune	0.70	0.31	-0.39	-0.06	0.14	-0.41	2.5	379.0	0.10
New-leg freeze	0.70	0.25	-0.45	-0.12	0.40	0.00	2.5	369.0	0.10
New-leg finetune	0.70	0.33	-0.36	-0.04	0.14	-0.40	2.5	340.0	0.10
EWC †	0.70	0.68	-0.01	0.31	0.52	-0.03	31.5	387.0	0.11
Online EWC †	0.70	0.66	-0.04	0.29	0.51	-0.03	7.3	399.0	0.10
ER (Reservoir) †	0.70	0.39	-0.31	0.02	0.22	-0.35	13.5	732.0	0.11
ER	0.70	0.50	-0.19	0.13	0.54	-0.07	13.5	758.0	0.11
PNN	0.70	0.62	-0.07	0.25	0.62	0.00	51.1	1799.0	0.10
MNTDP-S	0.70	0.64	-0.05	0.27	0.64	0.00	10.1	406.0	0.11
MNTDP-S (k=all)	0.70	0.63	-0.06	0.26	0.63	0.00	11.6	411.0	0.10
MNTDP-D	0.70	0.70	0.01	0.33	0.68	0.00	11.6	1299.0	0.15
MNTDP-D*	0.64	0.64	0.00	0.27	0.65	0.00	130.2	99.0	0.22
HAT*†	0.62	0.44	-0.18	0.07	0.60	0.00	26.6	42.0	0.13
HAT*†	0.68	0.51	-0.17	0.14	0.64	0.00	164.0	293.0	0.12

Table 5.12. – Results in the transfer evaluation stream with output perturbation. In this stream, the last task uses the same classes as the first task but in a different order and with an order of magnitude less data. Tasks 2, 3, 4 and 5 are distractors. * corresponds to models using an Alexnet backbone, † to models using stream-level cross-validation (see Section 5.6.1).

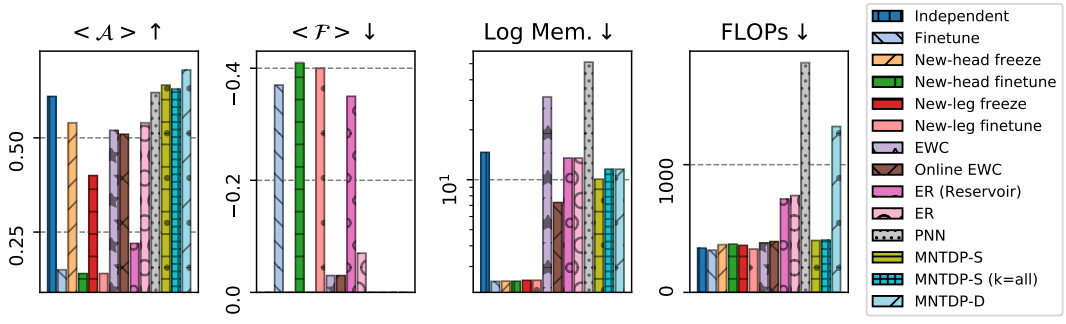


Figure 5.14. – Comparison of all baselines on the $\mathcal{S}^{\text{out}}$ stream

Stream $\mathcal{S}^{pl}$

Model	Acc T_5	Δ_{T_5',T_5}	$< \mathcal{A} >$	$< \mathcal{F} >$	Mem.	FLOPs	LCA@5
Independent	0.71	0.00	0.59	0.00	12.2	232.0	0.10
Finetune	0.57	-0.14	0.21	-0.30	2.4	274.0	0.12
New-head freeze	0.29	-0.42	0.45	0.00	2.4	294.0	0.13
New-head finetune	0.56	-0.15	0.20	-0.35	2.4	336.0	0.11
New-leg freeze	0.27	-0.44	0.37	0.00	2.5	390.0	0.11
New-leg finetune	0.58	-0.13	0.19	-0.34	2.5	375.0	0.11
EWC †	0.28	-0.43	0.27	-0.19	26.7	239.0	0.11
Online EWC †	0.28	-0.43	0.30	-0.17	7.3	282.0	0.12
ER (Reservoir)†	0.48	-0.23	0.20	-0.27	11.7	383.0	0.10
ER	0.51	-0.20	0.45	-0.08	11.7	597.0	0.10
PNN	0.56	-0.15	0.54	0.00	36.5	1742.0	0.13
MNTDP-S	0.58	-0.13	0.55	0.00	11.0	351.0	0.10
MNTDP-S (k=all)	0.60	-0.11	0.56	0.00	11.0	340.0	0.11
MNTDP-D	0.70	-0.01	0.62	0.00	11.6	1503.0	0.10
MNTDP-D*	0.65	-0.06	0.64	0.00	130.2	124.0	0.17
HAT*†	0.50	-0.21	0.58	0.00	26.5	50.0	0.11
HAT*† (Wide)	0.61	-0.10	0.61	0.00	163.7	312.0	0.12

Table 5.13. – Results in the plasticity evaluation stream. In this stream, we compare the performance on the probe task when it is the first problem encountered by the model and when it as already seen 4 distractor tasks. * corresponds to models using an Alexnet backbone, † to models using stream-level cross-validation (see Section 5.6.1).

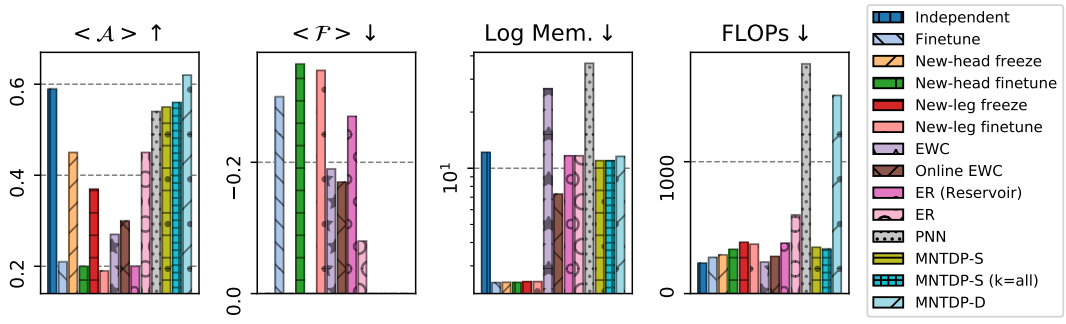


Figure 5.15. – Comparison of all baselines on the $\mathcal{S}^{pl}$ stream

Stream $\mathcal{S}^{long}$

Model	$< \mathcal{A} >$	$< \mathcal{F} >$	Mem.	FLOPs	LCA@5
Independent	0.57 ± 0.01	0.0 ± 0.0	243.7 ± 0.0	3542.33 ± 139.16	0.2
Finetune	0.2 ± 0.0	-0.35 ± 0.0	2.4 ± 0.0	4961.33 ± 112.16	0.23
New-head freeze	0.43 ± 0.01	0.0 ± 0.0	2.6 ± 0.0	5574.33 ± 249.65	0.27
Online EWC †	0.27 ± 0.01	-0.25 ± 0.01	7.4 ± 0.0	3882.67 ± 159.15	0.21
MNTDP-S	0.68 ± 0.0	0.0 ± 0.0	158.63 ± 2.58	5437.67 ± 110.77	0.21
MNTDP-D	0.75 ± 0.0	0.0 ± 0.0	102.03 ± 0.8	26066.67 ± 662.74	0.34
MNTDP-D*	0.75 ± 0.0	0.0 ± 0.0	1803.47 ± 16.45	2598.67 ± 70.48	0.46
HAT*†	0.24 ± 0.01	-0.1 ± 0.03	31.9 ± 0.0	147.0 ± 27.39	0.21
HAT (Wide) *†	0.32 ± 0.0	0.0 ± 0.0	285.0 ± 0.0	1056.33 ± 137.29	0.21

Table 5.14. – Results on the long evaluation stream. We report the mean and standard error using 3 different instances of the stream, all generated following the procedure described in Section 5.4. Standard errors are negligible for LCA.

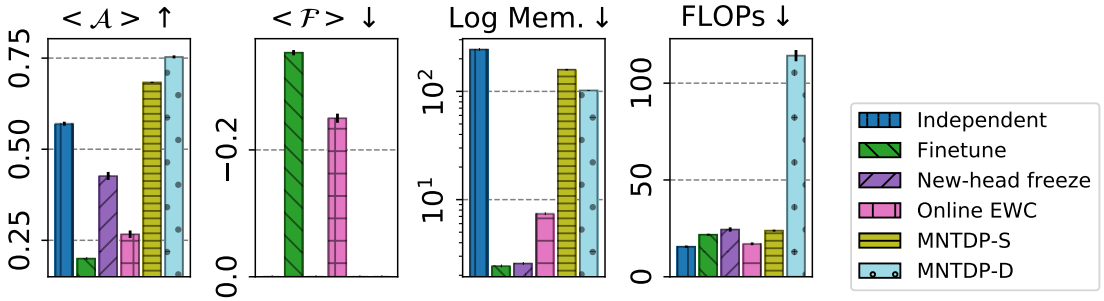


Figure 5.16. – Comparison of all baselines on the $\mathcal{S}^{\text{long}}$ stream

We conclude by reporting results on $\mathcal{S}^{\text{long}}$ composed of 100 tasks. Table 5.8 reports the results of all the approaches we could train without running into out-of-memory. MNTDP-D yields an absolute 18% improvement over the baseline *independent model* while using less than half of its memory, thanks to the discovery of many paths with shared modules. Its actual runtime is close to *independent model* because of GPU parallel computing. To match the capacity of MNTDP, we scale HAT's backbone to the maximal size that can fit in a Titan X GPU Memory (6.5x, wide version). The wider architecture greatly increases inference time in later tasks (see also discussion on memory complexity at test time in Section 5.6.2), while our modular approach uses the same backbone for every task and yet it achieves better performance. Figure 5.17 shows the average accuracy

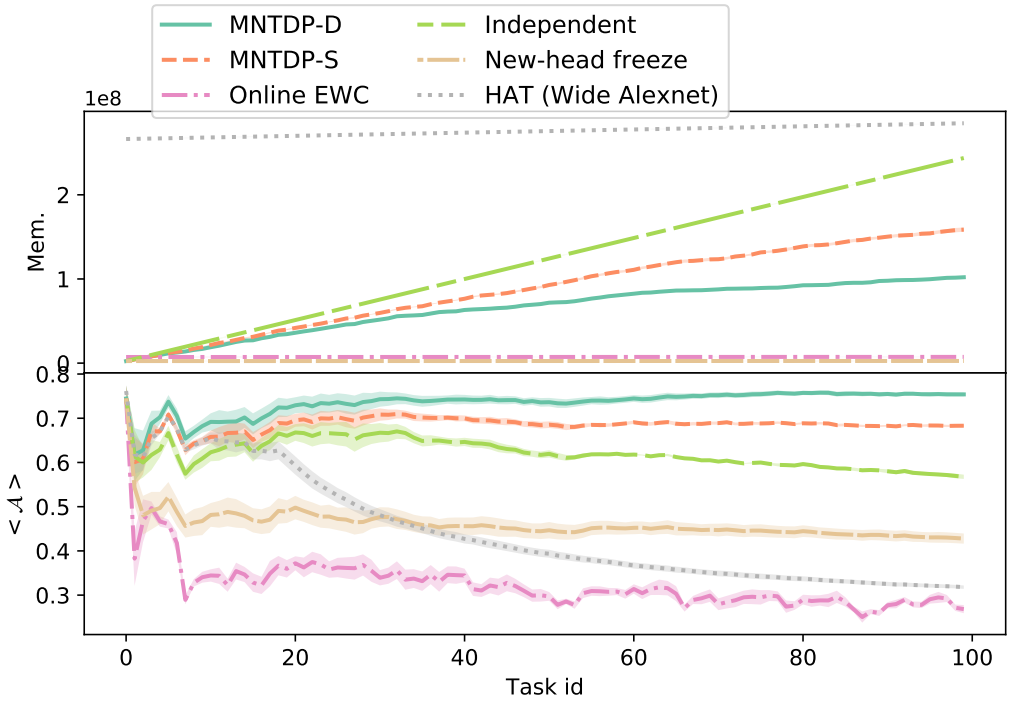


Figure 5.17. – Evolution of $< \mathcal{A} >$ and Mem. on $\mathcal{S}^{\text{long}}$.

up to the current task over time. MNTDP-D attains the best performance while growing sublinearly in terms of memory usage. Methods that do not evolve their architecture, like EWC, greatly suffer in terms of average accuracy.

5.6.5 Ablation:

We first study the importance of the prior. Instead of selecting the nearest neighbor path, we pick one path corresponding to one of the previous tasks at random. In this case, $\mathcal{T}(\mathcal{S}^{-})$ decreases from 0 to -0.2 and $\mathcal{T}(\mathcal{S}^{out})$ goes from 0 to -0.3. With a random path, MNTDP learns not to share any module, demonstrating that it is indeed important to form a good prior over the search space. Detailed results of Section 5.6.4 demonstrate how on small streams MNTDP is robust to the choice of k in the prior since we attain similar performance using $k = 1$ and $k = all$, although only $k = 1$ let us scale to $\mathcal{S}^{\text{long}}$. Finally, we explore the robustness to the number of modules by splitting each module in two, yielding a total of 10 modules per path, and by merging adjacent modules yielding a total of 3 modules for the same overall number of parameters. We find that $\mathcal{T}(\mathcal{S}^{\text{out}})$ decreases from 0 to -0.1, with a 9% decrease on t_1^-, when the number of modules decreases but stays the same when the number of modules increases, suggesting that the algorithm has to have a sufficient number of modules to flexibly grow.

5.6.6 Discovered paths

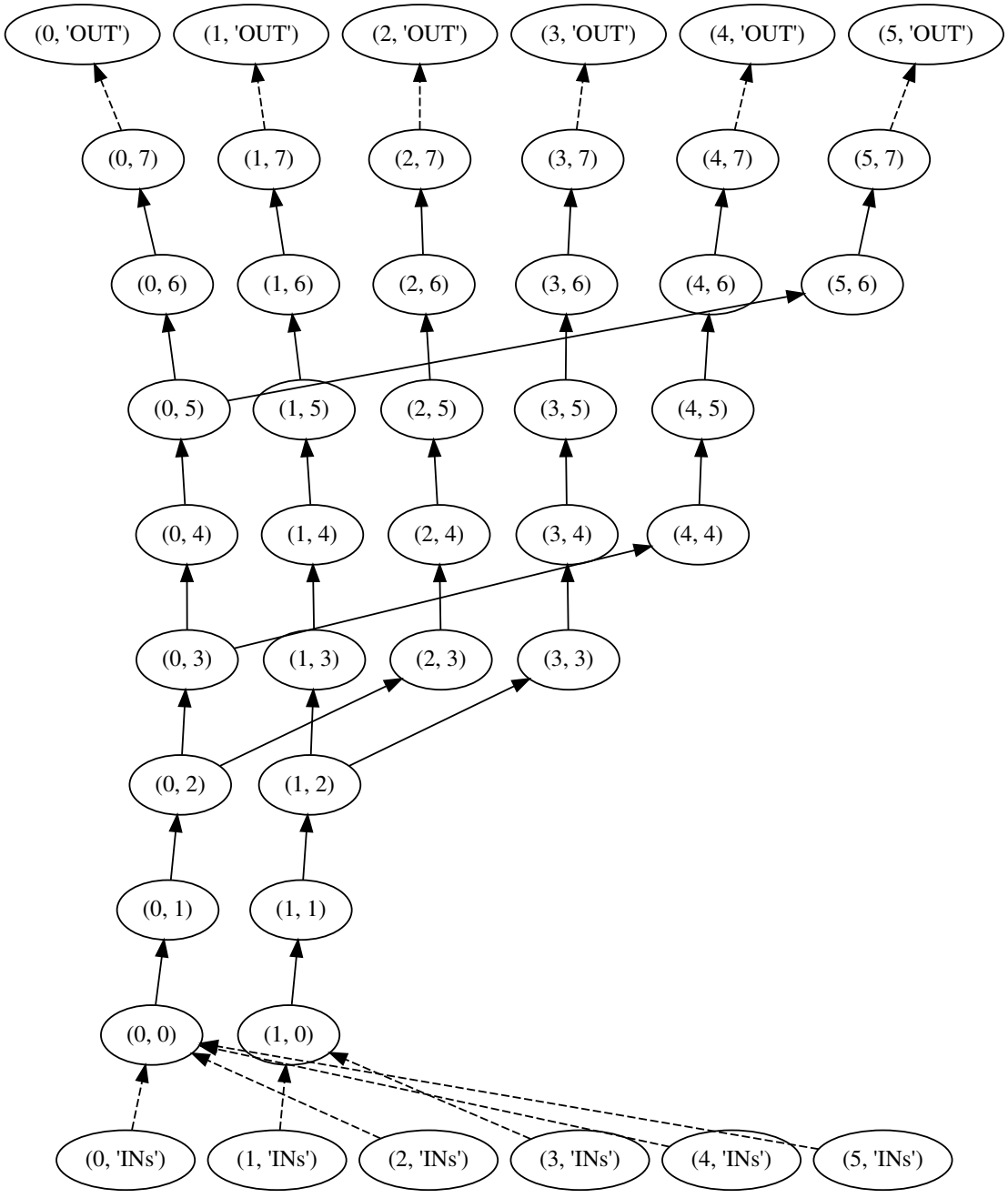


Figure 5.18. – Global graph of paths discovered by MNDTP-S on the $\mathcal{T}(\mathcal{S}^{\text{out}})$ Stream. When facing the last task, the model correctly identified that modules from the first task should be reused, ultimately introducing 2 new modules to solve it.

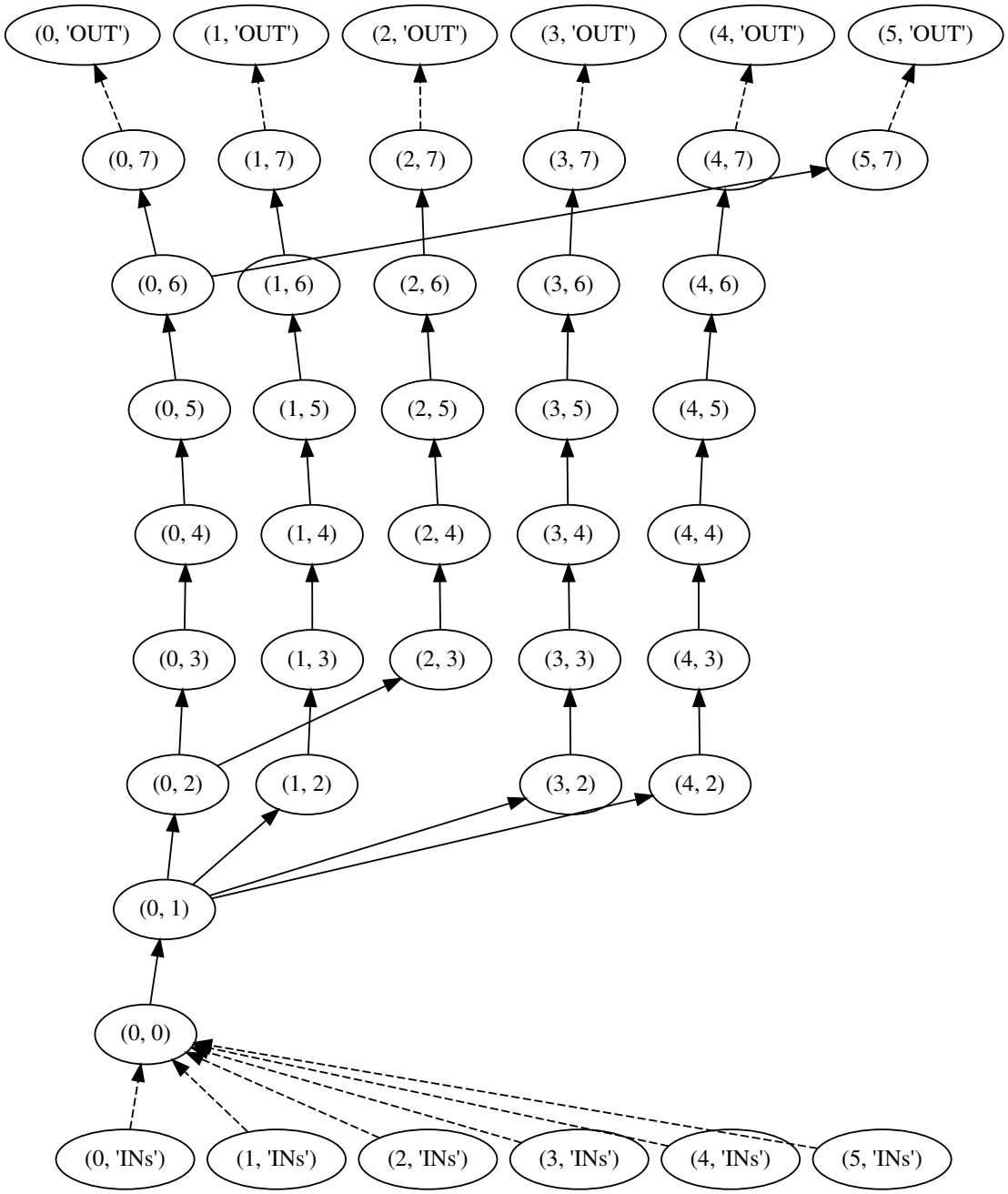


Figure 5.19. – Global graph of paths discovered by MNDTP-D on the $\mathcal{T}(\mathcal{S}^{\text{out}})$ Stream. "INs" (resp. "OUT") nodes are the input (resp. output) of the path for each task. Solid edges correspond to parameterized modules while dashed edges are only used to show which block is selected for each task and don't apply any transformation. We observe that a significant amount of new parameters are introduced for tasks 2, 3, 4, and 5, which are very different from the first task. The model is however able to correctly identify that the last task is very similar to the first one, resulting in a very large reuse of past modules and only introducing a new classification layer to adapt to the new task.

5.7 Conclusions

We introduced a new benchmark to enable a more comprehensive evaluation of CL algorithms, not only in terms of average accuracy and forgetting but also knowledge transfer and scaling. We have also proposed a modular network that can gracefully scale thanks to an adaptive prior over the search space of possible ways to connect modules. Our experiments show that our approach yields a very desirable trade-off between accuracy and compute/memory usage, suggesting that modularization in restricted search spaces is a promising avenue of investigation for continual learning and knowledge transfer.

CHAPTER 6

CONCLUSION

Contents

In this thesis, we tackled the Neural Architecture Search (NAS) problem. To sidestep the compute-intensive approaches that compose most of the literature, we approached it through different angles with a focus on budget constraints. This strategy allowed us to develop new models solving real-world problems and creating tools for the community to use.

6.1 Summary of Contributions

6.1.1 Budgeted Super Networks

We started with the challenging problem of Convolutional Neural Network (ConvNet) architecture search for Computer Vision. With the recent rise in popularity of Deep Learning comes an exponentially growing number of publications proposing new types of layers and new architectures. We proposed Budgeted Super Networks (BSN), a method for automatically discovering such architectures able to outperform reference models from the literature. This search algorithm allows its user to impose any kind of prior on the search procedure via the definition of a black box cost function and a maximum allowed budget. This cost function doesn't need to be differentiable and is directly optimized to learn the best possible architecture combining high predictive performance with strict respect of the budget constraint.

We demonstrated the effectiveness of our approach in a wide range of settings on the classification and semantic segmentation tasks. We validated its ability to explore different kinds of search spaces with two different backbones. Furthermore, we showed that the black-box approach accepting generic cost functions works flawlessly, using widely different kinds of cost and observing that the model can morph the network in a shape respecting them. Depending on the users need, BSN is able to learn efficient architecture minimizing the *memory* footprint of the inference model, its *latency* and the much more complicated *distributed computation* cost which is a high-level measure of the overall complexity of the topology of an architecture.

6.1.2 Stochastic Adaptive Neural Architecture Search

Then, we proposed to investigate the usage of budget-constrained NAS to computer audition. Thanks to the temporal component inherent to audio signals and the information an audio frame carries about the next one, we were able to propose a highly time-efficient model for the keyword spotting problem. Due to its omnipresence in connected devices, this problem is an excellent candidate for budgeted approaches.

We proposed Stochastic Adaptive Architecture Search (SANAS), a model able to adaptively decide which Deep Neural Network (DNN) architecture should be used at any point in a sequence, using shallow architectures on simple frames and more powerful ones when the signal is harder to process. The system can be trained without any prior knowledge about the structure of the problem at hand: it automatically learns how to evaluate the difficulty of a frame as well as which architecture needs to be used for any level of difficulty at test time.

SANAS is trained end-to-end to minimize a trade-off between the average computation cost per frame and the prediction performance, resulting in faster models than the state of the art for a similar performance or to better performance for a given computation time.

6.1.3 The Continual Transfer Learning Benchmark

As a third contribution, we proposed to explore other settings in which NAS models can be used. We presented a scenario in which a learner has to continually adapt itself to solve a stream of diverse tasks. While similar to Continual Learning (CL), this scenario put the emphasis on maximizing transfer instead of minimizing forgetting as it is commonly done in CL.

We identified a set of transfer-related properties that the learner should possess in order to perform well in this setting. We then introduced the Continual Transfer Learning (CTrL) benchmark, a set of streams created specifically to probe a learner on each of the desired properties, allowing a fine-grained to see which properties a given approach has and to identify its failure modes. We also proposed a long evaluation stream containing tasks of different sizes and having various degrees of similarity. This challenging setting allows testing the transfer ability of a learner at scale, as some tasks can't be solved without transferring knowledge from previously encountered tasks.

6.1.4 Modular Networks with Task-Driven Priors

Finally, we proposed a new modular architecture called Modular Networks with Task Driven Prior (MNTDP) relying on NAS to efficiently find the best combination of pre-trained modules and blank ones to solve each task encountered. To improve the scalability of our approach, we introduce a task-driven prior. It allows to dramatically reduce the search space by pruning unlikely connections, yielding constant training time w.r.t. the number of tasks.

Our experiments show that MNTDP is able to outperform existing approaches on the proposed benchmark while being on par with these methods on existing evaluation settings. It is the only approach able to solve the long evaluation stream and reaches a very desirable trade-off between accuracy and compute/memory usage, confirming that our method stays competitive at scale.

6.2 Perspectives for Future Work

Improving the search procedure The contributions presented in Chapter 3 and Chapter 4 sample each layer of an architectures using an independent Bernoulli distribution. The search procedure efficiency could be improved by keeping track of what has already been selected up to the current point and use this knowledge to take the next decision. For example, the conditional sampling procedure used by the stochastic version of MNTDP (5) could be extended and used in both the static and dynamic architecture search settings.

Extending the reach of adaptive models In Chapter 4, we showed that the SANAS adaptive model performs well on the keyword-spotting problem and we believe that the proposed approach could be extended to other sequential problems. We applied it to Reinforcement Learning (RL) tasks and obtained pre-

liminary results showing that besides selecting the size of the architecture, active learning behavior can automatically emerge without being prompted: the model not only selects which layer to use but also which features to look at.

For example in the Lunar-Lander environment (Brockman et al., 2016), the task is to control the motors of a ship to reach the ground at a reasonable speed and angle to avoid crashing. On this problem, the learned behavior of the adaptive model only looks at whether the legs of the ship are touching the ground when the ship is at low altitude and completely ignores them – by choosing to not connect them to the architecture input and therefore not processing them at all – to focus on speed and tilt features when the ship if farther from the ground. This behavior not only saves a small amount of computation but more importantly can save a lot of budget in applications where the feature acquisition is expensive, allowing to acquire only the features that will be processed.

Another interesting direction is to extend the control of the model to the time dynamics, allowing it to decide the duration of each time step. This could result in even more budget savings when the model can predict that nothing will happen nor should be done in the next x seconds.

More powerful architecture priors In Chapter 5, the MNTDP model uses a simple heuristic to decide which tasks are related to the current one and prune less promising parts of the search space based on that information. A direct improvement would be to use more elaborated kinds of priors to construct the search space.

One promising direction is to use task descriptors carrying more knowledge about the relationship between tasks. These descriptors could be computed from the data, for example by learning task-embeddings based on a few samples from each task. We could then use a distance metric in this task-space as a similarity metric. Another possible kind of task descriptors could be given as additional information coming with each task. For example, a textual description of the objective would allow zero shot-learning by recognizing already solved problems without looking at the data or by composing already acquired skills if the module library already contains the required knowledge.

Longer-term research perspectives In Chapter 5, we focused on the CL problem as the first step toward a lifelong learning system able to efficiently solve new tasks by leveraging knowledge acquired throughout his life.

A possible next step toward this objective could be to step aside from the strict CL setting as such lifelong learners don't mind forgetting, what they really want is to learn and remember quickly. Therefore by relaxing the anytime evaluation constraint, the problem could be shifted from *not forgetting* to *quickly remembering*.

Since all intelligent beings are using never-ending streams of information coming from multiple senses to evolve in their environment, another desirable step would be to train such systems on even larger streams containing different modalities and multi-modal tasks. Including the large amount of data currently used to reach state-of-the-art performance in each modality is a challenging problem since it can take weeks to fully extract from scratch the information contained in each of these gigantic datasets. This is where a true lifelong learner that never starts *tabula rasa* once released but always builds on its current knowledge would shine and appears as a necessary step towards the next generation of artificially intelligent systems.

BIBLIOGRAPHY

Adel, Tameem, Han Zhao, and Richard E. Turner (2020). "Continual learning with adaptive weights". In: *International Conference on Learning Representations*.

Alet, Ferran, Tomás Lozano-Pérez, and Leslie Pack Kaelbling (2018). "Modular meta-learning". In: *2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings*. Vol. 87. Proceedings of Machine Learning Research. PMLR, pp. 856–868. URL: http://proceedings.mlr.press/v87/alet18a.html.

Amodei, Dario et al. (2016). "Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 173–182. URL: http://proceedings.mlr.press/v48/amodei16.html.

Andrychowicz, Marcin et al. (2016). "Learning to learn by gradient descent by gradient descent". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al., pp. 3981–3989. URL: https://proceedings.neurips.cc/paper/2016/hash/fb87582825f9d28a8d42c5e5e5e8b23d-Abstract.html.

Arik, Sercan Ömer et al. (2017). "Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting". In: *ICASSP 2018* abs/1703.05390.

Baevski, Alexei et al. (2020). "wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. URL: https://proceedings.neurips.cc/paper/2020/hash/92d1e1eb1cd6f9fba3227870bb6d7Abstract.html.

Baker, Bowen et al. (2017). "Designing Neural Network Architectures using Reinforcement Learning". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=S1c2cvqee.

Bengio, Emmanuel et al. (2015). "Conditional Computation in Neural Networks for faster models". In: *CoRR* abs/1511.06297. URL: http://arxiv.org/abs/1511.06297.

Bolukbasi, Tolga et al. (2017). "Adaptive Neural Networks for Fast Test-Time Prediction". In: *CoRR* abs/1702.07811. arXiv: 1702.07811. URL: http://arxiv.org/abs/1702.07811.

Brock, Andrew et al. (2018). "SMASH: One-Shot Model Architecture Search through HyperNetworks". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=rydeCEhs-.

Brockman, Greg et al. (2016). *OpenAI Gym*. eprint: arXiv:1606.01540.

Brown, Tom B. et al. (2020). *Language Models are Few-Shot Learners*. arXiv: 2005.14165 [cs.CL].

Cai, Han, Ligeng Zhu, and Song Han (2019). "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=HylVB3AqYm.

Chaudhry, Arslan et al. (2019a). "Continual Learning with Tiny Episodic Memories". In: *CoRR* abs/1902.10486. arXiv: 1902.10486. URL: http://arxiv.org/abs/1902.10486.

Chaudhry, Arslan et al. (2019b). "Efficient Lifelong Learning with A-GEM". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=Hkf2%5C_sC5FX.

Cho, KyungHyun et al. (2014). "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: *CoRR* abs/1409.1259. arXiv: 1409.1259. URL: http://arxiv.org/abs/1409.1259.

Chollet, François (2017). "Xception: Deep Learning with Depthwise Separable Convolutions". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, pp. 1800–1807. DOI: 10.1109/CVPR.2017.195. URL: https://doi.org/10.1109/CVPR.2017.195.

Cimpoi, M. et al. (2014). "Describing Textures in the Wild". In: *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David (2015). "BinaryConnect: Training Deep Neural Networks with binary weights during propagations". In: *CoRR* abs/1511.00363. arXiv: 1511.00363. URL: http://arxiv.org/abs/1511.00363.

Deng, Jia et al. (2009). "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848. URL: https://doi.org/10.1109/CVPR.2009.5206848.

Denoyer, Ludovic and Patrick Gallinari (2014). "Deep Sequential Neural Network". In: *CoRR* abs/1410.0510. URL: http://arxiv.org/abs/1410.0510.

Devlin, Jacob (2017). *Sharp Models on Dull Hardware: Fast and Accurate Neural Machine Translation Decoding on the CPU*. URL: http://arxiv.org/abs/1705.01991.

Dong, Xuanyi et al. (2017). "More is Less: A More Complicated Network with Less Inference Complexity". In: *CoRR* abs/1703.08651. arXiv: 1703.08651. URL: http://arxiv.org/abs/1703.08651.

Duan, Yan et al. (2016). "RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning". In: *CoRR* abs/1611.02779. arXiv: 1611.02779. URL: http://arxiv.org/abs/1611.02779.

Fernando, Chrisantha et al. (2017). "PathNet: Evolution Channels Gradient Descent in Super Neural Networks". In: *CoRR* abs/1701.08734. URL: http://arxiv.org/abs/1701.08734.

Finn, Chelsea, Pieter Abbeel, and Sergey Levine (2017a). "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 1126–1135. URL: http://proceedings.mlr.press/v70/finn17a.html.

— (2017b). "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 1126–1135. URL: http://proceedings.mlr.press/v70/finn17a.html.

Finn, Chelsea et al. (2019). "Online meta-learning". In: *International Conference on Machine Learning*.

Goodfellow, I. J. et al. (2013). "An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks". In: *arXiv*.

Gordon, Ariel et al. (2018). "MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, pp. 1586–1595. DOI: 10.1109/CVPR.2018.00171. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Gordon%5C_MorphNet%5C_Fast%5C_%5C_CVPR%5C_2018%5C_paper.html.

Ha, David, Andrew M. Dai, and Quoc V. Le (2017). "HyperNetworks". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=rkpACe1lx.

Han, Song, Huizi Mao, and William J Dally (2015). "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huff-

man Coding". In: *International Conference on Learning Representations (ICLR'16 best paper award)*.

Hassibi, Babak and David G. Stork (1992). "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon". In: *Advances in Neural Information Processing Systems 5, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992]*. Ed. by Stephen Jose Hanson, Jack D. Cowan, and C. Lee Giles. Morgan Kaufmann, pp. 164–171. URL: http://papers.nips.cc/paper/647-second-order-derivatives-for-network-pruning-optimal-brain-surgeon.

He, Kaiming et al. (2016). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, pp. 770–778. DOI: 10.1109/CVPR.2016.90. URL: https://doi.org/10.1109/CVPR.2016.90.

Hinton, Geoffrey E., Oriol Vinyals, and Jeffrey Dean (2015). "Distilling the Knowledge in a Neural Network". In: *CoRR* abs/1503.02531. arXiv: 1503.02531. URL: http://arxiv.org/abs/1503.02531.

Hochreiter, Sepp, A. Steven Younger, and Peter R. Conwell (2001). "Learning to Learn Using Gradient Descent". In: *Artificial Neural Networks - ICANN 2001, International Conference Vienna, Austria, August 21-25, 2001 Proceedings*. Ed. by Georg Dorffner, Horst Bischof, and Kurt Hornik. Vol. 2130. Lecture Notes in Computer Science. Springer, pp. 87–94. DOI: 10.1007/3-540-44668-0_13. URL: https://doi.org/10.1007/3-540-44668-0%5C_13.

Howard, Andrew G. et al. (2017). "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861. arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

Huang, Gao et al. (2017). "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243. URL: https://doi.org/10.1109/CVPR.2017.243.

Huang, Gao et al. (2018). "Multi-Scale Dense Networks for Resource Efficient Image Classification". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=Hk2aImxAb.

Huang, Zehao and Naiyan Wang (2018). "Data-Driven Sparse Structure Selection for Deep Neural Networks". In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XVI*. Ed. by Vittorio Ferrari et al. Vol. 11220. Lecture Notes in Computer Science. Springer, pp. 317–334. DOI: 10.1007/978-3-030-01270-0_19. URL: https://doi.org/10.1007/978-3-030-01270-0%5C_19.

Hung, Steven C. Y. et al. (2019). "Compacting, Picking and Growing for Unforgetting Continual Learning". In: *Advances in Neural Information Processing Systems*

32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada. Ed. by Hanna M. Wallach et al., pp. 13647–13657. URL: http://papers.nips.cc/paper/9518-compacting-picking-and-growing-for-unforgetting-continual-learning.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html.

Jaderberg, Max et al. (2017). "Population Based Training of Neural Networks". In: *CoRR* abs/1711.09846. arXiv: 1711.09846. URL: http://arxiv.org/abs/1711.09846.

Jamieson, Kevin G. and Ameet Talwalkar (2016). "Non-stochastic Best Arm Identification and Hyperparameter Optimization". In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*. Ed. by Arthur Gretton and Christian C. Robert. Vol. 51. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 240–248. URL: http://proceedings.mlr.press/v51/jamieson16.html.

Kae, Andrew et al. (2013). "Augmenting CRFs with Boltzmann Machine Shape Priors for Image Labeling". In: *CVPR*.

Kessler, Samuel et al. (2019). "Indian Buffet Neural Networks for Continual Learning". In: *CoRR* abs/1912.02290. arXiv: 1912.02290.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1412.6980.

Kirkpatrick, James et al. (2016). "Overcoming catastrophic forgetting in neural networks". In: *CoRR* abs/1612.00796. arXiv: 1612.00796. URL: http://arxiv.org/abs/1612.00796.

Krizhevsky, Alex (2009a). "Learning Multiple Layers of Features from Tiny Images". In: *University of Toronto, technical report*.

— (2009b). *Learning multiple layers of features from tiny images*. Tech. rep.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by Peter L. Bartlett et al., pp. 1106–1114. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

LeCun, Y. et al. (1998). "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

LeCun, Yann, John S. Denker, and Sara A. Solla (1989). "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*. Ed. by David S. Touretzky. Morgan Kaufmann, pp. 598–605. URL: http://papers.nips.cc/paper/250-optimal-brain-damage.

Li, Liam et al. (2020). "A System for Massively Parallel Hyperparameter Tuning". In: *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. Ed. by Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze. mlsys.org. URL: https://proceedings.mlsys.org/book/303.pdf.

Li, Lisha et al. (2017a). "Hyperband: Bandit-Based Configuration Evaluation for Hyperparameter Optimization". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=ry18Ww5ee.

Li, Xilai et al. (2019). "Learn to Grow: A Continual Structure Learning Framework for Overcoming Catastrophic Forgetting". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, pp. 3925–3934. URL: http://proceedings.mlr.press/v97/li19m.html.

Li, Zhenguo et al. (2017b). "Meta-SGD: Learning to Learn Quickly for Few Shot Learning". In: *CoRR* abs/1707.09835. arXiv: 1707.09835. URL: http://arxiv.org/abs/1707.09835.

Liu, Chenxi et al. (2018a). "Progressive Neural Architecture Search". In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part I*. Ed. by Vittorio Ferrari et al. Vol. 11205. Lecture Notes in Computer Science. Springer, pp. 19–35. DOI: 10.1007/978-3-030-01246-5_2. URL: https://doi.org/10.1007/978-3-030-01246-5%5C_2.

Liu, Hanxiao, Karen Simonyan, and Yiming Yang (2019). "DARTS: Differentiable Architecture Search". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=S1eYHoC5FX.

Liu, Hanxiao et al. (2018b). "Hierarchical Representations for Efficient Architecture Search". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=BJQRKzbA-.

Lomonaco, Vincenzo and Davide Maltoni (2017). "Core50: a new dataset and benchmark for continuous object recognition". In: *arXiv:1705.03550*.

Lopez-Paz, David and Marc'Aurelio Ranzato (2017). "Gradient Episodic Memory for Continual Learning". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al., pp. 6467–6476. URL: http://papers.nips.cc/paper/7225-gradient-episodic-memory-for-continual-learning.

Low, Daniel, Kate Bentley, and Satrajit Ghosh (Jan. 2020). "Automated assessment of psychiatric disorders using speech: A systematic review". In: *Laryngoscope Investigative Otolaryngology* 5. DOI: 10.1002/lio2.354.

Lu, Liang (2017). "Toward Computation and Memory Efficient Neural Network Acoustic Models with Binary Weights and Activations". In: *CoRR* abs/1706.09453. arXiv: 1706.09453. URL: http://arxiv.org/abs/1706.09453.

McClelland, James L, Bruce L McNaughton, and Randall C O'reilly (1995). "Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory." In: *Psychological review*.

McCloskey, Michael and Neal J Cohen (1989). "Catastrophic interference in connectionist networks: The sequential learning problem". In: *Psychology of learning and motivation*.

McGill, Mason and Pietro Perona (2017). "Deciding How to Decide: Dynamic Routing in Artificial Neural Networks". In: *CoRR* abs/1703.06217. URL: http://arxiv.org/abs/1703.06217.

Mesaros, Annamaria, Toni Heittola, and Tuomas Virtanen (Nov. 2018). "A multi-device dataset for urban acoustic scene classification". In: *Proceedings of the Detection and Classification of Acoustic Scenes and Events 2018 Workshop (DCASE2018)*, pp. 9–13. URL: https://arxiv.org/abs/1807.09840.

Micikevicius, Paulius et al. (2017). "Mixed Precision Training". In: *CoRR* abs/1710.03740. arXiv: 1710.03740. URL: http://arxiv.org/abs/1710.03740.

Nan, F. and V. Saligrama (May 2017). "Adaptive Classification for Prediction Under a Budget". In: *ArXiv e-prints*. arXiv: 1705.10194 [stat.ML].

Netzer, Yuval et al. (2011). "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

Nichol, Alex, Joshua Achiam, and John Schulman (2018a). "On First-Order Meta-Learning Algorithms". In: *CoRR* abs/1803.02999. arXiv: 1803.02999. URL: http://arxiv.org/abs/1803.02999.

— (2018b). "On First-Order Meta-Learning Algorithms". In: *CoRR* abs/1803.02999. arXiv: 1803.02999. URL: http://arxiv.org/abs/1803.02999.

Pham, Hieu et al. (2018). "Efficient Neural Architecture Search via Parameter Sharing". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by

Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 4092–4101. URL: http://proceedings.mlr.press/v80/pham18a.html.

Rajasegaran, Jathushan et al. (2019). "Random Path Selection for Incremental Learning". In: *CoRR* abs/1906.01120. arXiv: 1906.01120. URL: http://arxiv.org/abs/1906.01120.

Real, Esteban et al. (2017). "Large-Scale Evolution of Image Classifiers". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 2902–2911. URL: http://proceedings.mlr.press/v70/real17a.html.

Real, Esteban et al. (2019). "Regularized Evolution for Image Classifier Architecture Search". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, pp. 4780–4789. DOI: 10.1609/aaai.v33i01.33014780. URL: https://doi.org/10.1609/aaai.v33i01.33014780.

Rebuffi, Sylvestre-Alvise, Hakan Bilen, and Andrea Vedaldi (2017). "Learning multiple visual domains with residual adapters". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al., pp. 506–516. URL: http://papers.nips.cc/paper/6654-learning-multiple-visual-domains-with-residual-adapters.

Ring, Mark B. (1994). "Continual Learning in Reinforcement Environments". PhD thesis. Austin, Texas 78712: University of Texas at Austin.

Rolnick, David et al. (2019). "Experience Replay for Continual Learning". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach et al., pp. 348–358. URL: http://papers.nips.cc/paper/8327-experience-replay-for-continual-learning.

Romero, Adriana et al. (2014). "FitNets: Hints for Thin Deep Nets". In: *CoRR* abs/1412.6550. arXiv: 1412.6550. URL: http://arxiv.org/abs/1412.6550.

Ruder, Sebastian et al. (2019). "Latent Multi-Task Architecture Learning". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, pp. 4822–4829. DOI: 10.1609/aaai.v33i01.33014822. URL: https://doi.org/10.1609/aaai.v33i01.33014822.

Rusu, Andrei A. et al. (2016). "Progressive Neural Networks". In: *CoRR* abs/1606.04671. arXiv: 1606.04671. URL: http://arxiv.org/abs/1606.04671.

Rybakov, Oleg et al. (Oct. 2020). "Streaming Keyword Spotting on Mobile Devices". In: *Interspeech 2020*. DOI: 10.21437/interspeech.2020-1003. URL: http://dx.doi.org/10.21437/Interspeech.2020-1003.

Sainath, Tara N. and Carolina Parada (2015). "Convolutional neural networks for small-footprint keyword spotting". In: *INTERSPEECH*. ISCA, pp. 1478–1482.

Saxena, Shreyas and Jakob Verbeek (2016). "Convolutional Neural Fabrics". In: *CoRR* abs/1606.02492. URL: http://arxiv.org/abs/1606.02492.

Schwarz, Jonathan et al. (2018). "Progress & Compress: A scalable framework for continual learning". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 4535–4544. URL: http://proceedings.mlr.press/v80/schwarz18a.html.

Serizel, Romain et al. (2020). "Sound event detection in synthetic domestic environments". In: *ICASSP 2020 - 45th International Conference on Acoustics, Speech, and Signal Processing*. Barcelona, Spain. URL: https://hal.inria.fr/hal-02355573.

Serrà, Joan et al. (2018). "Overcoming catastrophic forgetting with hard attention to the task". In: *CoRR* abs/1801.01423. arXiv: 1801.01423. URL: http://arxiv.org/abs/1801.01423.

Shin, Hanul et al. (2017). "Continual Learning with Deep Generative Replay". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al., pp. 2990–2999. URL: http://papers.nips.cc/paper/6892-continual-learning-with-deep-generative-replay.

Silver, David et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nat.* 529.7587, pp. 484–489. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

Silver, David et al. (2017). "Mastering the game of Go without human knowledge". In: *Nat.* 550.7676, pp. 354–359. DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.

Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams (2012). "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by Peter L. Bartlett et al., pp. 2960–2968. URL: https://proceedings.neurips.cc/paper/2012/hash/05311655a15b75fab86956663e1819cd-Abstract.html.

Srivastava, Rupesh Kumar, Klaus Greff, and Jürgen Schmidhuber (2015). "Training Very Deep Networks". In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. Ed. by Corinna Cortes et al., pp. 2377–2385. URL: https://proceedings.neurips.cc/paper/2015/hash/215a71a12769b056c3c32e7299f1Abstract.html.

Sutton, Richard S. and Andrew G. Barto (1998). *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press. ISBN: 978-0-262-19398-6. URL: https://www.worldcat.org/oclc/37293240.

Sutton, Richard S. et al. (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*. Ed. by Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller. The MIT Press, pp. 1057–1063. URL: http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.

Tang, Raphael and Jimmy Lin (2018). "Deep Residual Learning for Small-Footprint Keyword Spotting". In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2018, Calgary, AB, Canada, April 15-20, 2018*.

Tang, Raphael et al. (2018). "An Experimental Analysis of the Power Consumption of Convolutional Neural Networks for Keyword Spotting". In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2018, Calgary, AB, Canada, April 15-20, 2018*.

Thrun, S. (1994). "A lifelong learning perspective for mobile robot control". In: *Proceedings of the IEEE/RSJ/GI Conference on Intelligent Robots and Systems*.

Thrun, Sebastian (1998). "Lifelong learning algorithms". In: *Learning to learn*. Springer.

Thrun, Sebastian and Lorien Y. Pratt, eds. (1998). *Learning to Learn*. Springer. ISBN: 978-1-4613-7527-2. DOI: 10.1007/978-1-4615-5529-2. URL: https://doi.org/10.1007/978-1-4615-5529-2.

Turpault, Nicolas et al. (Oct. 2019). "Sound event detection in domestic environments with weakly labeled data and soundscape synthesis". In: *Workshop on Detection and Classification of Acoustic Scenes and Events*. New York City, United States. URL: https://hal.inria.fr/hal-02160855.

Vanhoucke, Vincent, Andrew Senior, and Mark Z. Mao (2011). "Improving the speed of neural networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*.

Ven, Gido M. van de and Andreas S. Tolias (2019). "Three scenarios for continual learning". In: *CoRR* abs/1904.07734. arXiv: 1904.07734. URL: http://arxiv.org/abs/1904.07734.

Veniat, Tom and Ludovic Denoyer (2018). "Learning Time/Memory-Efficient Deep Architectures With Budgeted Super Networks". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, pp. 3492–3500. DOI: 10.1109/CVPR.2018.00368. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Veniat%5C_Learning%5C_TimeMemory-Efficient%5C_Deep%5C_CVPR%5C_2018%5C_paper.html.

Veniat, Tom, Ludovic Denoyer, and Marc'Aurelio Ranzato (2021). "Efficient Continual Learning with Modular Networks and Task-Driven Priors". In: *9th International Conference on Learning Representations, ICLR 2021* abs/2012.12631. arXiv: 2012.12631. URL: https://arxiv.org/abs/2012.12631.

Véniat, Tom, Olivier Schwander, and Ludovic Denoyer (2019). "Stochastic Adaptive Neural Architecture Search for Keyword Spotting". In: *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2019, Brighton, United Kingdom, May 12-17, 2019*. IEEE, pp. 2842–2846. DOI: 10.1109/ICASSP.2019.8683305. URL: https://doi.org/10.1109/ICASSP.2019.8683305.

Wah, C. et al. (2011). "CaLtech-UCSD birds-200-2011 dataset". In: *Tech Report: CNS-TR-2011-001*.

Warden, Pete (2018). "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition". In: *CoRR* abs/1804.03209. arXiv: 1804.03209. URL: http://arxiv.org/abs/1804.03209.

Wierstra, Daan et al. (2007). "Solving Deep Memory POMDPs with Recurrent Policy Gradients". In: *Artificial Neural Networks - ICANN 2007, 17th International Conference, Porto, Portugal, September 9-13, 2007, Proceedings, Part I*.

Williams, Ronald J. (1992). "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Machine Learning* 8, pp. 229–256. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

Wortsman, Mitchell et al. (2020). "Supermasks in Superposition". In: *CoRR* abs/2006.14769. arXiv: 2006.14769. URL: https://arxiv.org/abs/2006.14769.

Wu, Bichen et al. (2019). "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, pp. 10734–10742. DOI: 10.1109/CVPR.2019.01099. URL: http://openaccess.thecvf.com/content%5C_CVPR%5C_2019/html/Wu%5C_FBNet%5C_Hardware-Aware%5C_Efficient%5C_ConvNet%5C_Design%5C_via%5C_Differentiable%5C_Neural%5C_Architecture%5C_Search%5C_CVPR%5C_2019%5C_paper.html.

Xiao, Han, Kashif Rasul, and Roland Vollgraf (Aug. 28, 2017). *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. arXiv: cs.LG/1708.07747 [cs.LG].

Xu, Ju and Zhanxing Zhu (2018). "Reinforced Continual Learning". In: *CoRR* abs/1805.12369. arXiv: 1805.12369. URL: http://arxiv.org/abs/1805.12369.

Yoon, Jaehong et al. (2018). "Lifelong Learning with Dynamically Expandable Networks". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=Sk7KsfW0-.

Zenke, Friedemann, Ben Poole, and Surya Ganguli (2017). "Continual Learning Through Synaptic Intelligence". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 3987–3995. URL: http://proceedings.mlr.press/v70/zenke17a.html.

Zhang, Daniel et al. (2021). *The AI Index 2021 Annual Report*. arXiv: 2103.06312 [cs.AI].

Zhang, Mengmi et al. (2019). "Prototype Reminding for Continual Learning". In: *CoRR* abs/1905.09447. arXiv: 1905.09447. URL: http://arxiv.org/abs/1905.09447.

Zhang, Xiangyu et al. (2018). "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, pp. 6848–6856. DOI: 10.1109/CVPR.2018.00716. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Zhang%5C_ShuffleNet%5C_An%5C_Extremely%5C_CVPR%5C_2018%5C_paper.html.

Zhang, Yundong et al. (2017). "Hello Edge: Keyword Spotting on Microcontrollers". In: *CoRR* abs/1711.07128. arXiv: 1711.07128. URL: http://arxiv.org/abs/1711.07128.

Zhao, Hanbin et al. (2020). "What and Where: Learn to Plug Adapters via NAS for Multi-Domain Learning". In: *CoRR* abs/2007.12415. arXiv: 2007.12415. URL: https://arxiv.org/abs/2007.12415.

Zhong, Zhao et al. (2018). "Practical Block-Wise Neural Network Architecture Generation". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, pp. 2423–2432. DOI: 10.1109/CVPR.2018.00257. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Zhong%5C_Practical%5C_Block-Wise%5C_Neural%5C_CVPR%5C_2018%5C_paper.html.

Zhu, H. et al. (May 2017). "Sequential Dynamic Decision Making with Deep Neural Nets on a Test-Time Budget". In: *ArXiv e-prints*. arXiv: 1705.10924 [stat.ML].

Zoph, Barret and Quoc V. Le (2017). "Neural Architecture Search with Reinforcement Learning". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=r1Ue8Hcxg.

Zoph, Barret et al. (2018). "Learning Transferable Architectures for Scalable Image Recognition". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, pp. 8697–8710. DOI: 10.1109/CVPR.2018.00907. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Zoph%5C_Learning%5C_Transferable%5C_Architectures%5C_CVPR%5C_2018%5C_paper.html.

APPENDIX

Contents

A.1 Details of $\mathcal{S}^{long}$

Task id	Dataset	Classes	# Train	# Val	# Test
1	mnist	[6, 3, 7, 5, 0]	25	15	4830
2	svhn	[2, 1, 9, 0, 7]	25	15	5000
3	svhn	[2, 0, 6, 1, 5]	5000	2500	5000
4	svhn	[1, 5, 0, 7, 4]	25	15	5000
5	fashion-mnist	[T-shirt/top, Pullover, Trouser, Sandal, Sneaker]	25	15	5000
6	fashion-mnist	[Shirt, Ankle boot, Sandal, Pullover, T-shirt/top]	5000	2500	5000
7	svhn	[3, 1, 7, 6, 9]	25	15	5000
8	cifar100	[spider, maple tree, tulip, leopard, lizard]	25	15	500
9	cifar10	[frog, automobile, airplane, cat, horse]	25	15	5000
10	fashion-mnist	[Ankle boot, Bag, T-shirt/top, Shirt, Pullover]	25	15	5000
11	mnist	[4, 8, 7, 6, 3]	5000	2500	4914
12	cifar10	[automobile, truck, dog, horse, deer]	5000	2500	5000

Table A.1. – Details of the tasks used in $\mathcal{S}^{\text{long}}$, part 1/5.

Task id	Dataset	Classes	# Train	# Val	# Test
13	cifar100	[sea, forest, bear, chimpanzee, dinosaur]	25	15	500
14	mnist	[3, 2, 9, 1, 7]	25	15	5000
15	fashion-mnist	[Bag, Ankle boot, Trouser, Shirt, Dress]	25	15	5000
16	cifar10	[frog, cat, horse, airplane, deer]	25	15	5000
17	cifar10	[bird, frog, ship, truck, automobile]	5000	2500	5000
18	svhn	[0, 4, 7, 5, 6]	5000	2500	5000
19	mnist	[6, 5, 9, 4, 8]	5000	2500	4806
20	mnist	[8, 5, 6, 4, 9]	5000	2500	4806
21	cifar100	[sea, pear, house, spider, aquarium fish]	25	15	500
22	cifar100	[kangaroo, ray, tank, crocodile, table]	2250	250	500
23	cifar100	[trout, rose, pear, lizard, baby]	25	15	500
24	svhn	[3, 2, 8, 1, 5]	5000	2500	5000
25	cifar100	[skyscraper, bear, rocket, tank, spider]	25	15	500
26	cifar100	[telephone, porcupine, flatfish, plate, shrew]	2250	250	500
27	cifar100	[lawn mower, crocodile, tiger, bed, bear]	25	15	500
28	svhn	[3, 7, 1, 5, 6]	25	15	5000
29	fashion-mnist	[Ankle boot, Sneaker, T-shirt/top, Coat, Bag]	5000	2500	5000
30	mnist	[6, 9, 0, 3, 7]	5000	2500	4938
31	cifar10	[automobile, truck, deer, bird, dog]	25	15	5000
32	cifar10	[dog, airplane, frog, deer, automobile]	5000	2500	5000
33	svhn	[1, 9, 5, 3, 6]	5000	2500	5000
34	cifar100	[whale, orange, chimpanzee, poppy, sweet pepper]	25	15	500
35	cifar100	[worm, camel, bus, keyboard, spider]	25	15	500

Table A.2. – Details of the tasks used in $\mathcal{S}^{\text{long}}$, part 2/5.

Task id	Dataset	Classes	# Train	# Val	# Test
36	fashion-mnist	[T-shirt/top, Coat, Ankle boot, Shirt, Dress]	25	15	5000
37	cifar10	[dog, deer, ship, truck, cat]	25	15	5000
38	cifar10	[cat, dog, airplane, ship, deer]	5000	2500	5000
39	svhn	[7, 6, 4, 2, 9]	25	15	5000
40	mnist	[9, 7, 1, 3, 2]	25	15	5000
41	cifar100	[mushroom, butterfly, bed, boy, motorcycle]	25	15	500
42	fashion-mnist	[Shirt, Pullover, Bag, Sandal, T-shirt/top]	25	15	5000
43	cifar100	[rabbit, bear, aquarium fish, bee, bowl]	25	15	500
44	fashion-mnist	[Coat, T-shirt/top, Pullover, Shirt, Sandal]	25	15	5000
45	fashion-mnist	[Pullover, Dress, Coat, Shirt, Sandal]	25	15	5000
46	mnist	[3, 9, 7, 6, 4]	25	15	4940
47	cifar10	[deer, bird, dog, automobile, frog]	25	15	5000
48	svhn	[8, 7, 1, 0, 4]	25	15	5000
49	cifar100	[forest, skunk, poppy, bridge, sweet pepper]	2250	250	500
50	cifar100	[caterpillar, can, motorcycle, rabbit, wardrobe]	25	15	500
51	cifar100	[trout, mountain, kangaroo, pine tree, bee]	25	15	500
52	cifar100	[clock, fox, castle, bus, willow tree]	25	15	500
53	cifar10	[cat, airplane, dog, ship, truck]	25	15	5000
54	mnist	[9, 7, 8, 1, 5]	25	15	4866
55	fashion-mnist	[Bag, T-shirt/top, Sandal, Shirt, Dress]	25	15	5000
56	fashion-mnist	[Sneaker, Ankle boot, Coat, Sandal, Trouser]	5000	2500	5000
57	mnist	[1, 4, 3, 9, 7]	25	15	4982
58	cifar10	[truck, automobile, frog, ship, dog]	25	15	5000
59	mnist	[7, 2, 8, 5, 4]	25	15	4848

Table A.3. – Details of the task in $\mathcal{S}^{\text{long}}$, part 3/5.

Task id	Dataset	Classes	# Train	# Val	# Test
60	mnist	[2, 8, 9, 1, 7]	5000	2500	4974
61	svhn	[9, 5, 1, 8, 6]	25	15	5000
62	mnist	[1, 8, 7, 4, 5]	25	15	4848
63	cifar10	[truck, dog, bird, automobile, airplane]	25	15	5000
64	mnist	[8, 4, 3, 7, 6]	25	15	4914
65	svhn	[3, 5, 7, 2, 1]	25	15	5000
66	cifar100	[otter, camel, bee, road, poppy]	25	15	500
67	svhn	[4, 2, 1, 8, 7]	25	15	5000
68	mnist	[3, 7, 6, 8, 9]	25	15	4932
69	fashion-mnist	[Pullover, Sneaker, Trouser, Dress, Sandal]	25	15	5000
70	svhn	[5, 0, 7, 2, 3]	25	15	5000
71	svhn	[9, 6, 2, 4, 8]	25	15	5000
72	mnist	[7, 1, 2, 0, 6]	25	15	4938
73	cifar10	[dog, automobile, ship, airplane, cat]	25	15	5000
74	mnist	[0, 7, 6, 2, 4]	25	15	4920
75	cifar10	[bird, deer, airplane, dog, ship]	25	15	5000
76	cifar100	[mountain, bicycle, caterpillar, spider, possum]	25	15	500
77	svhn	[8, 3, 4, 0, 6]	25	15	5000
78	svhn	[1, 5, 9, 0, 8]	25	15	5000
79	cifar100	[can, dolphin, house, pickup truck, crab]	25	15	500
80	cifar100	[squirrel, possum, crocodile, mountain, hamster]	25	15	500

Table A.4. – Details of the task in $\mathcal{S}^{\text{long}}$, part 4/5.

Task id	Dataset	Classes	# Train	# Val	# Test
81	mnist	[7, 0, 1, 6, 2]	25	15	4938
82	fashion-mnist	[T-shirt/top, Dress, Trouser, Shirt, Sneaker]	25	15	5000
83	cifar10	[cat, frog, automobile, dog, airplane]	25	15	5000
84	cifar10	[automobile, cat, dog, ship, horse]	25	15	5000
85	cifar100	[cup, otter, orchid, kangaroo, rose]	25	15	500
86	mnist	[1, 5, 7, 2, 9]	25	15	4892
87	svhn	[6, 5, 3, 2, 7]	25	15	5000
88	cifar10	[dog, deer, cat, frog, bird]	25	15	5000
89	mnist	[6, 2, 5, 9, 4]	25	15	4832
90	cifar100	[pear, rocket, sea, road, orange]	25	15	500
91	svhn	[0, 8, 4, 6, 1]	25	15	5000
92	cifar10	[truck, horse, ship, deer, dog]	25	15	5000
93	mnist	[5, 8, 6, 4, 3]	25	15	4806
94	svhn	[2, 6, 3, 4, 1]	25	15	5000
95	fashion-mnist	[Bag, Trouser, Sneaker, Ankle boot, Sandal]	25	15	5000
96	svhn	[7, 9, 1, 5, 8]	25	15	5000
97	cifar100	[lamp, otter, skyscraper, sea, raccoon]	25	15	500
98	cifar100	[clock, flatfish, snake, can, man]	25	15	500
99	svhn	[6, 3, 0, 8, 7]	25	15	5000
100	fashion-mnist	[Shirt, Coat, Dress, Sandal, Pullover]	25	15	5000

Table A.5. – Details of the task in $\mathcal{S}^{\text{long}}$, part 5/5.