Master thesis : Sparse hypernetworks for multitasking

Auteur : Cubélier, François
Promoteur(s) : Geurts, Pierre
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2021-2022
URI/URL : http://hdl.handle.net/2268.2/14574

Sparse hypernetworks for multitasking

Author: **Francois Cubelier**
Supervisor: **Prof. Pierre Geurts**

Master's thesis completed in order to obtain the Degree Of Science (MSc) in Computer Science and Engineering

University of Liège - School of Engineering
Belgium
Academic year 2021-2022

Abstract

In this work, we study hypernetworks as meta-models, i.e. models producing models. We first review the literature on hypernetworks by addressing its terminology issues and exploring various applications. We also propose a typology of hypernetworks. We then present a new approach for building complete hypernetworks via sparse models that solves the scalability problem of hypernetworks. We analyze different hyperparameter combinations for sparse hypernetworks to show the effect of the connectivity types proposed. Their performances are compared with the current solution, chunked hypernetworks, on multitasking computer vision benchmarks and we show that they can match their performance, even though chunked hypernetworks are slightly ahead on more complex problems. Finally, we show that linear sparse hypernetwork generally outperforms non-linear sparse hypernetworks and chunked hypernetworks for inferring a new target model for a new task with a pretrained multitasking hypernetwork, especially for new tasks that are radically different from the pretraining tasks.

Acknowledgments

I would like to thank my supervisor Professor Pierre Geurts for allowing me to work on this interesting topic and providing me his guidance throughout this entire year through interesting discussions. I also thank him for his help and feedback that helped me continually improve my work and myself during this project.

Contents

Chapter 1

Introduction

1.1 Motivations

1.1.1 Context

There has been interest in building more dynamic deep learning models in recent years. The attention mechanism, which computes parameters dynamically based on the input, allows to reach state-of-the-art results in natural language understanding (Vaswani et al. [2017b]) and computer vision (Dosovitskiy et al. [2020]). Researchers are trying to build more multitasking capable models with the aim of producing less narrow artificial intelligence.

In deep learning, hypernetworks represent neural networks that output the parameters of another neural network. Classical deep learning generally aims to learn a function from x to y with a differentiable model $\mathcal{M}$ s.t. $\hat{y} = \mathcal{M}(x; \theta)$ parameterized by θ. Similarly, the parameters θ of this target model can be predicted conditioned on some input z with another neural network $\mathcal{H}$ st. $\theta = \mathcal{H}(z; \nu)$ parameterized by ν, which is called a hypernetwork. They are naturally fitted for building task- or context-dependent models. A symbolic illustration of a hypernetwork is given in Figure 1.1.

1.1.2 Problem statement

Hypernetworks provide a convenient way to produce dynamic or conditional neural network parameters. They are typically used at the level of a layer or for neural networks that have a small number of parameters. For example, a hypernetwork $\mathcal{H}$ could be taught to output the parameters θ_l of the layer l of another neural network $\mathcal{M}$. The input of such hypernetwork could be the vector embedding of a task, which would help to specialize $\mathcal{M}$ for the corresponding task, or the previous feature maps of the layer preceding l, which would dynamically adapt θ_l to help $\mathcal{M}$ make better predictions.

Instead of predicting only part of the model $\mathcal{M}$ parameters, one could directly predict all the parameters θ of the network $\mathcal{M}$, which is the focus of this

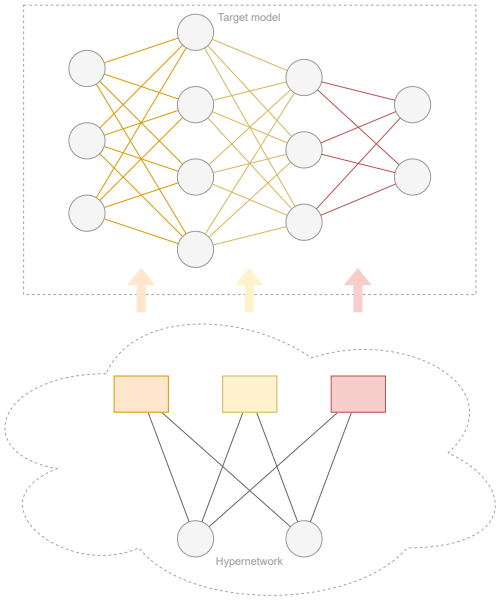


Figure 1.1: Symbolic representation of a hypernetwork producing the parameters of a target model.

work. We call such a hypernetwork that produces all the parameters of a target model, a **complete hypernetwork**. Modern deep neural networks, however, typically have a large number of parameters, and predicting this amount of parameters cannot be done trivially. Indeed, a hypernetwork could have a very large output space. Using a classical architecture, such as a fully connected MLP with hidden layers proportional to the size of the output space, would not be scalable in terms of number of parameters. This is the **scalability** issue of hypernetworks.

Current solutions involve reusing the same model multiple times and predicting the target parameters by parts. The first solution, chunked hypernetworks, slices the target model θ into equally sized chunks. A single model $\mathcal{H}_C$ is then reused multiple times to predict each part of the target model. Chunking produces an arbitrary slice of parameters; a single chunk may contain parameters that come from completely different layers. Therefore, it is possible that chunking may not be the most efficient solution for predicting entire models, although it obtains good results in practice. The second solution, layer-wise hypernetworks, divides the parameter vector θ into layer parameters. Layer-wise hypernetwork may seem more warranted. However, they are limited by the varying sizes of each layer of the target model, which may introduce some inefficiency. Indeed, layer-wise hypernetworks have to deal with layers of different sizes, contrary to chunking. There are 2 solutions. On the one hand, one could use a hypernetwork that is able to produce enough parameters for the layer that has

the largest number of parameters. Then, to predict the parameters of smaller layers, one can just ignore part of the actual output of the hypernetwork and keep only as many parameters as needed for the current layer. With this technique, some outputs are uselessly produced, which introduces some inefficiency. On the other hand, if the target network has a limited number of different sizes for its layers, then one can learn a hypernetwork for each size. The downside of this technique is that it reduces the sharing of information between the size-specific hypernets. In any case, they are still limited by the size of the largest layer, which may be a problem if this size is already so large that it poses a scalability issue. One cannot build layer-wise hypernetworks without knowing the sizes of the different layers and in particular the size of the biggest layer in terms of the number of parameters. They are also a very local version of hypernetworks, as they can be applied locally to a single layer independently of the rest of the target model.

1.2 Project statement and contributions

The current method of chunked hypernetworks allows one to build scalable hypernetworks without being limited by the maximal size of a particular layer. However, the arbitrary slicing of the parameters seems unsatisfactory. In this work, we propose a new global sparse architecture for hypernetworks that avoids sharing the same model for different parts of a target network and does not need to slice it arbitrarily. This architecture is essentially an extremely sparse expanding MLP. It has a scalable number of parameters and can predict very large target models with very large layers. We define this sparse architecture by specifying its sparse connectivity pattern, initialization, and training.

We evaluate different hyperparamter combinations for sparse hypernetwoks. We compare sparse hypernetworks with the chunked one and separate experts on multitasking computer vision benchmarks. We also compare its robustness by predicting target networks with different levels of sparsity. We show that, even though sparse hypernetworks can match chunked ones on some problems, chunked hypernetworks perform slightly better on more complex problems.

In addition, we evaluated the ability of a trained multitasking hypernetwork to infer new tasks and show that the linear sparse hypernetworks can outperform chunked hypernetworks in this domain.

Finally, as a preamble to the work, we propose a review of hypernetworks in the literature, as well as a typology of hypernetworks.

1.3 Organization of this document

Chapter 2 reviews the literature on hypernetworks. It tries to clarify its terminology and gives a brief overview of its history. Some applications of hypernetworks are reported.

In Chapter 3, a formal definition of hypernetworks is given and a hypernetwork typology is proposed.

Chapter 4 defines the sparse hypernetwork architecture. Some connectivity patterns are proposed and analyzed in terms of the number of parameters of the resulting hypernetworks. We describe the initialization and normalization used, as well as the training of multitasking hypernetworks.

Chapter 5 discusses the experiments and their results. Different combinations of hyperparameters of the proposed sparse hypernetwork are tested on a benchmark. Then the sparse hypernetwork is compared to the chunked one and individual experts over a few computer vision multitasking problems. There is a study of the robustness of hypernetworks with respect to less overparameterized target neural networks by predicting target models with varying levels of sparsity. We also analyze the generalization properties of multitasking hypernetworks. In this generalization study, the hypernetwork is first pretrained on a set of tasks before trying to solve new tasks with this hypernetwork. The inferred tasks are of different natures: they can be similar, completely different, or a composition of the initial tasks.

Chapter 2

Literature review

In this chapter, we review the literature on hypernetworks. We clarify hypernetworks terminology and give a brief history of hypernetworks and their origins. Then we review hypernetwork applications.

2.1 Hypernetworks

By Hypernetwork, we denote the concept of models producing models and its variations in the field of machine learning. A parametric model $\mathcal{H}$, the *hypernetwork*, is used to produce the parameters of another model $\mathcal{M}$, the *main* or *target* model. Hypernetworks are typically implemented by neural network models trained by gradient descent, because they are well suited for the task of producing another model, which is generally unsupervised. A formal definition is given in Chapter 3.

Hypernetworks are used in a wide variety of applications such as Bayesian neural networks, ensemble methods, transfer learning, etc. Applications are reviewed in 2.4. Although various applications have been tested, hypernetworks remain vastly understudied.

Hypernetworks have reached state-of-the-art results [Knyazev et al., 2021; von Oswald et al., 2020; Brock et al., 2018], yet big challenges remain. The scalability of the hypernetworks is a big issue, considering the potentially gigantic size of the output, i.e. the number of parameters of the target model.

2.2 Terminology

In 2016, Ha et al. introduced the concept of hypernetwork and defined it as a "network generating the weights of another network". In this work, we use this term to refer to a model generating another model because it is currently the unifying term in the literature used to denote this concept. It may be argued that it is not the best name to designate this concept, *metanetworks*, *hypermodels*, or naturally *metamodels* would have perhaps been more suitable. The

concept is actually suffering from the fact that there is no unified terminology to denote it. That is why we use the currently most widely used term in the literature: *hypernetworks* (or hypernet in short).

In the subsequent literature [Ha et al., 2016], some works use hypernetworks without directly referring to them. Other concepts have been used to refer to some kind of hypernetwork, such as fast weights [von der Malsburg, 1994; Srivastava et al., 2019; Hinton, 1987; Schmidhuber, 1992], dynamic networks [Jia et al., 2016; Klein et al., 2015; Zhang et al., 2019], metanetworks [Munkhdalai and Yu, 2017] and parameter prediction [Shakibi, 2014].

Dwaracherla et al. [2020] refer to hypernetworks as hypermodels and restricts the use of hypernetworks to only neural network implementations, arguing that single fully connected layers are not part of neural networks, but can be used as hypermodel.

It has to be noted that weaker versions of hypernetworks are actually quite common. Numerous modern deep learning architectures make use of auxiliary models that act on a primary model [Perez et al., 2017; Karras et al., 2019]. In general, modern architectures consist of high-level composition, which could be interpreted as a way of dynamically computing some parameters of the model. If many use hypernetwork-like architectures, few know about the concept.

2.3 History

Even though the term hypernetworks was coined in 2016 [Ha et al., 2016], it was not the first time that the idea of generating dynamically parameters was used. One of the first mentions of dynamically changing neural network connections are known as fast weights. Mixture of expert models are also related to hypernetworks as it dynamically selects models based on input. Some neuroevolution methods can be seen as hypernets using evolutionary algorithms.

2.3.1 Fast weight

von der Malsburg [1994] and Feldman [1982] were among the first to discuss the idea of dynamically changing connections in neural networks. von der Malsburg [1994] indicates the possibility of modeling a dynamic connection w_{ij} from brain cell i to cell j as a multiplication between a slow "classical" weight s_{ij} and a dynamical connection a_{ij} that decays to zero over time: $w_{ij} = s_{ij}a_{ij}$. Hinton [1987] studies an additive view: $w_{ij} = s_{ij} + a_{ij}$ were s_{ij} and a_{ij} have different learning rates and a_{ij} decays to zero by a factor h at each iteration. The idea is that these dynamical models could improve performance by using a short-term memory, the dynamical connection (fast weights), in addition to the long-term memory modeled by classical connection (slow weights).

Schmidhuber [1992] introduced a method where a model computes the weight changes of another main network in order to model a short term memory.

2.3.2 Mixture of expert models

Jacobs et al. [1991] and Jordan and Jacobs [1994] were one of the first to study mixture of expert models. In this type of architecture, expert models are dynamically selected based on input via a gating network. In the conventional setting, considering the output o_i of expert i, the final output of the network is computed as $\sum_i o_i p_i$ with p_i the weight of the contribution of expert i that is dynamically computed based on input (p_i are normalized s.t $\sum_i p_i = 1$). To some extent, mixture of expert models are linked to hypernetworks. First, when a static network would only compute one contribution vector $\mathbf{p}$, the Mixture of Experts computes it dynamically based on input. Second, a simpler way of dealing with the problem would be to train a large network to categorize the data and predict the corresponding output. With Mixture of Expert, as with hypernets, the work is factorized into smaller networks, one gating network for categorizing data and other smaller expert models networks for predicting the output. This is one of the key insights on why hypernets could be more efficient thanks to hyperconditioning [Galanti and Wolf, 2020].

2.3.3 Hypernetworks

Before hypernetworks was coined, other works used parameter generator subnetworks.

Ba et al. [2015] studied parameter prediction. They showed a correlation of parameters in a typical neural network. They produced the weight matrix W of a linear layer in MLP with a linear operation: $W = UW_\alpha$ based on a lower dimensional learnable matrix W_α. Bertinetto et al. [2016] propose to learn feed-forward one shot learners and use a factorized representation of the weight matrices in order to reduce the dimensionality of the output of the hypernet.

Jaderberg et al. [2015] proposed hypernetworks like architecture where the input image is used to compute the parameters of a transformation of that image. They called it *spatial transformers*.

Noh et al. [2015] introduced a hypernetwork without defining the concept for visual question answering (VQA). The textual question is fed into a hypernetwork which predicts some weights of the primary network. They refer to to this idea as "parameter prediction network" and used parameters hashing [Chen et al., 2015] to reduce the size of the parameter matrix to output.

Others also introduced hypernetworks without defining the concept and giving it a name: Ba et al. [2015] predict MLP and convolutional networks for image classification based on textual and image inputs. They use dimensionality reduction techniques to reduce the number of parameters to predict.

Klein et al. [2015] and Jia et al. [2016] make use of dynamic filters computed based on the input and applied later in the model. Riegler et al. [2015] also predict the parameters of a submodule based on blur kernel input via an MLP for image super-resolution.

Finally, in 2016, Ha et al. [2016] introduced the term hypernetworks. In this paper, hypernetworks are presented as a versatile tool that can be used

as a form of relaxed weight sharing for deep convolutional networks, as well as a way of dynamically changing the parameters of recurrent neural networks. In order to produce the parameters of a deep convolutional network with less learnable parameters, they make use of weight sharing in the hypernet which itself consists in a 2 layered linear network. For the RNN, they use an hypernet, that is an RNN too and evolves in parallel.

2.4 Applications

2.4.1 Vision

Jia et al. [2016] showcase an architectural type of hypernetwork used for filtering images. The input image is first fed into a hypernetwork predicting filters. These filters are then used to produce the output image. The authors suggested that their model could be used to pre-train networks for various other tasks, as it learns some filters in an unsupervised way.

2.4.2 Functional representation

Klocek et al. [2019] propose to represent natural images as a function $\mathcal{I} : \mathbb{R}^2 \to [0, 1]^3$ which associates an RGB triplet (r, g, b) to each point coordinate (x, y). Their method makes use of a hypernet $\mathcal{H}$ that takes as input an image and produces the parameters of the corresponding functions $\mathcal{I}$ implemented as an MLP. One advantage of such a technique is that this representation is independent of any number of pixels or sampling frequencies. This functional representation can be sampled with any density of pixels or even nonuniformly, which could be interesting if you only want to deal with some part of an image at a time. They also successfully manage to make interpolation of images by simple linear interpolation of the generated parameters of the corresponding images.

Littwin and Wolf [2019] approach 3D shape reconstruction by representing 3D objects as a function $\mathcal{I} : \mathbb{R}^3 \to [0, 1]$ which associates an score of presence in the object to each point coordinate (x, y, z). The hypernet takes as input a 2D image and generates the parameters of $\mathcal{I}$ that represent the corresponding 3D object. They also propose the interpolation of 3D objects via this method. Their method can be compared to another parallel work, which does not use hypernetworks [Mescheder et al., 2019], where all inputs (both image and position) are concatenated and fed into a single network. Lior Wolf [2020] argues that this method can be less efficient than hypernetworks, which may be due to the fact that hypernetwork conditioning is more suited than classical conditioning where the inputs and the condition are concatenated [Galanti and Wolf, 2020].

Spurek et al. [2020a,b, 2021]; Kostiuk et al. [2021] also study the functional representation of 3D objects. HyperCloud [Spurek et al., 2020a] proposes 2 methods for representing 3D objects: 1) as a transformation of a uniform ball to the actual object in order to generate point clouds, 2) as a transformation of a 3D sphere to the surface of the object in order to generate surface mesh.

In these methods, the hypernet takes as inputs a point cloud and produces the parameters of the corresponding MLP representing the object. HyperFlow [Spurek et al., 2020b] works in a similar way but uses CNF-based hypernet auto-encoder to produce the parameters of CNF representation of images (instead of MLP based), which yields better results. HyperPocket [Spurek et al., 2021] uses a similar method in order to reconstruct the full point cloud of an object from which we only have parts. Finally, HyperColor [Kostiuk et al., 2021] directly outputs colored 3D objects for video game applications.

2.4.3 Distribution on models

Hypernetworks produce the parameters of other neural networks. Therefore, they are naturally suited to represent implicit distribution over a model's parameters. This advantage can, for example, be used in a Bayesian setting or for building ensembles of models.

Bayesian neural networks

Krueger et al. [2018] use hypernetworks for bayesian inference. The main model parameters are produced via an invertible generator network from a Gaussian distribution. The hypernetwork is used as a generator of models and models a distribution over models. It is trained in a variational inference scheme. The main difficulty of these Bayesian techniques is to make the hypernetwork learn a good distribution on models that produces diverse and performant models. It should avoid collapsing into producing only the best model. In this work, the hypernet is limited to producing only scaling factors of the parameters. They experiment their method on regularization, active learning and anomaly/adversarial example detection. Similarly, Karaletsos et al. [2018] study probabilistic meta-representations of neural networks. Sheikh et al. [2018] use hypernet generators to sample competitive neural networks. Pawlowski et al. [2018] also implement implicit Bayesian neural networks via hypernets. The main advantage of hypernet for Bayesian neural networks is that they allow to represent a wide range of distributions. They typically are more scalable than other Bayesian methods for neural networks. Henning et al. [2018] and Ratzlaff and Fuxin [2020] learn generators of neural networks via adversarial training. Bachman et al. [2018]; Deutsch et al. [2019]; Karaletsos and Bui [2020]; Xu et al. [2021] also analyse modeling distribution of neural networks.

Ensembles

Ukai et al. [2019] suggest the use of hypernets to generate ensemble models. They study the difference between uniform and normal distribution on the latent space generating weights of the target network. They observe that uniform distribution yields better standalone models while normal yields more diverse models suitable for ensembling. Kobayashi et al. [2021] also go into that direction by noting that deep ensembles do not necessarily perform better with the

top-performing networks but rather with simpler, less accurate models. Hypernetworks seems appropriate for ensemble methods as we can sample as many networks as we want by training a single hypernet. Yet, Ukai et al. [2019] do not make the strong conclusion that hypernetwork-based ensembling clearly outperforms other ensembling methods or that hypernetworks generate models outperforming normal ones. They noted that hypernetwork-based methods can simply match other methods in terms of accuracy. In addition to ensemble methods, they also compare all-in-one and layer-wise hypernetworks as well as varying latent input distributions.

Dwaracherla et al. [2020] show that linear hypernetworks ensembles can be more computationally efficient than other ensembling methods.

Wenzel et al. [2021] investigate hyperparameters ensembles based on hypernetworks where parameters are conditioned on hyperparameters themselves.

von Oswald et al. [2021] learn late-phase weights via hypernetworks and apply model averaging to create ensembles.

2.4.4 Multitasking and Continual learning

Because they can produce other models, hypernetworks are naturally used in a multitasking setting. A single hypernetwork can produce a model for each task.

Multitasking

Tay et al. [2020] tackle the problem of multitasking for natural language understanding tasks. They proposed a new HyperGrid transformer that handles these tasks as a single model as opposed to multimodels which require a lot more parameters. The hypernet, a single layer feed-forward network, outputs scaling factors of the weights of the main net. The weight matrices of the main net are produced via the outer product of 2 vectors, which would typically be a global vector and an input-based vector. It can also be generated only on input-based vectors or with a task-dependent vector. The matrix is then expanded into a bigger matrix forming a grid. Their model performance matches other state-of-the-art methods like T5 [Raffel et al., 2020]. Multiple hypernet architectures are compared, notably weight gating vs output gating, for which they conclude that weight gating performs better. In weight gating, the weights of the linear transformations are directly gated, while in output gating, it is simply the outputs of the linear transformations which are gated (after ReLU activation). Mahabadi et al. [2021] also propose a parameter efficient technique where hypernetworks are shared between layers of a transformer. Notably, these hypernetworks are fed with layers id embeddings and position embeddings of the adapter in the layer additionally to the task embeddings.

Navon et al. [2021] and Lin et al. [2021], two concurrent works, applied hypernetworks to multi-objective optimization. For a fixed learning capacity, the model typically has to find a tradeoff between multiple objectives. Here, a hypernet is used to produce nets based on a preference vector for the objectives. Once it has been trained, one can just find the optimal model for a given

preference vector by using the hypernet.

Ruchte and Grabocka [2021], on the contrary, argue that a classical concatenation of input and preference vectors is more efficient.

Continual learning

Continual learning aims to reduce catastrophic forgetting, which occurs while neural networks learn tasks sequentially and forget how to perform on previously learned tasks.

von Oswald et al. [2020] propose task-conditioned hypernetwork for continual learning. A chunked hypernetwork architecture is used in order to reduce the number of parameters and output regularization is used to further decrease forgetting. Ehret et al. [2021] study this method in the context of sequential data with recurrent neural networks. They noted that this regularization method can outperform weight-importance method. Huang et al. [2021] applied task-conditioned hypernetworks in continual reinforcement learning. While their hypernetwork method matched the baselines, they noted that improvements were needed regarding the huge size of the hypernetwork. Henning et al. [2021] improve on von Oswald et al. [2020] with a probabilistic extension of task-conditioned hypernetworks (termed "posterior meta-replay") in which task-specific posteriors over target models are learned.

Few-shot learning, pre-training and transfer learning

Mahabadi et al. [2021] implement a multitasking transformer via hypernetworks instead of task-specific adapter modules, which typically do not share information. They tested their model on NLP few-shot domain transfer and it surpassed the adapter baseline in most cases.

Rusu et al. [2019] propose an interesting meta-learning scheme where datasets corresponding to a task are encoded into a latent vector z and then decoded into a model specific to the task. In this method, the meta-learning occurs in the lower dimensional space z instead of the high dimensional space of the parameters as in Finn et al. [2017]. Lamb et al. [2020] also encode a dataset and its metadata into a latent variable z that is decoded into a model corresponding to the data. The aim is to adapt the model to new features in the dataset. They show that they can produce good initialization parameters conditioned on a context consisting of observations with the new features and the associated metadata. Note that these methods, which encode a dataset, require an encoder that can handle a variable-length input.

2.4.5 Pruning and compression

Hypernetworks can also be used to produce pruned models or compress them. Liu et al. [2019b] propose a PruningNet to produce the weights of a pruned network. The PruningNet is trained to output weights from randomly sampled encoding vectors which correspond to different pruned target structures. Once

it is trained, architectures can be sampled and evaluated. In particular, an evolutionary algorithm is applied to find the encoding vectors of the best pruned architecture.

Li et al. [2020] introduced DHP, a differentiable meta-pruning technique. Each layer of the target architecture has an associated latent vector z representation from which an hypernet generates the corresponding weights. Thanks to the hypernet architecture, sparse latent vectors generate a sparse weight matrix. Therefore, they apply a L_1 regularization on the latent vector to sparsify the target architecture. They use SGD to optimize the hypernet but rely on proximal gradient method to find the best latent vectors.

2.5 Conclusion

In this chapter, we reviewed hypernetworks. We began by discussing the problem of terminology, pointing out that "hypernetworks" is the term most often used to refer to neural networks producing other neural networks.

We briefly introduced a history of hypernetworks and their predecessors starting from the idea of fast weight that adapts model parameters, going through related concepts like Mixture of experts which can dynamically select models based on the input and finally the actual implementation of neural networks producing the parameters of other neural networks learned via gradient descent.

We then reviewed a few applications of hypernetworks. They can be used as a way a representing a 3d object or an image as functions in a functional representation setting. They can also be used in a Bayesian framework to model distribution over models and to make ensemble models. They are very useful for multitasking applications, especially in a continual learning, few-shot learning, or transfer learning setting. They can also be used to predict neural networks with desired properties like sparsity.

Hypernetworks can be used in a variety of settings. Their can bring many interesting properties that we need in deep learning, like dynamic models, context-dependent models, and distribution over models. They are however not yet perfect meta-models. Research still needs to be done in that direction and there are certainly other innovative ways of using them that remain to be explored.

Chapter 3

Hypernetwork typology

The hypernetwork literature has developed rapidly in the previous years, leading to a variety of models and methods. In this section, we start by giving a formal definition of hypernetworks. Then, we propose a typology of hypernetwork models. First, we differentiate them according to the nature of their input. Then we analyze them with respect to their outputs: Do they produce the entire target network or only a part of it, and how to design a high-level architecture accordingly?

3.1 Formal Definition

In deep learning, hypernetworks denote neural networks that produce the parameters of another neural network. Let us first define a function $\mathcal{M}$ (a neural network), differentiable with respect to its parameters $\theta \in \mathbb{R}^n$, with input x and output y:

$$y = \mathcal{M}(x; \theta)$$

Then we can define a hypernetwork $\mathcal{H}$, differentiable model with input $z \in \mathbb{R}^d$, that produces the parameters θ of the model $\mathcal{M}$:

$$\theta = \mathcal{H}(z; \nu),$$

where ν are the parameters of the hypernetwork. We call $\mathcal{M}$ the target model or the main model of $\mathcal{H}$. An example is shown in Figure 3.1.

The target model $\mathcal{M}$ is equipped with a loss function $\mathcal{L}_{\mathcal{M}}(\theta)$. In order to train the hypernetwork $\mathcal{H}$ to produce parameters for $\mathcal{M}$, which is an unsupervised task, we can rely on this loss function. We can optimize the loss function $\mathcal{L}_{\mathcal{M}}(\theta)$ by gradient descent on its parameters ν for a given z, which is possible because both $\mathcal{M}$ and $\mathcal{H}$ are differentiable models. One can also learn z together with the hypernetwork, for example, in the case of a task-conditioned hypernetwork. This leads to the simplest loss function $\mathcal{L}_{\mathcal{H}}(z, \nu)$ that can be used to train a hypernetwork:

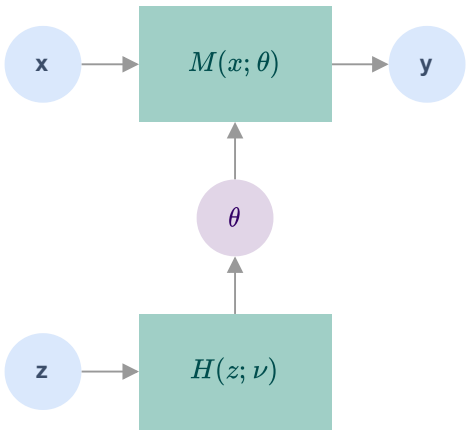


Figure 3.1: A hypernetwork $\mathcal{H}$ parametrized by ν producing the parameters θ of a target model $\mathcal{M}$ modeling $y = \mathcal{M}(x, \theta)$

$$\mathcal{L}_{\mathcal{H}}(z, \nu) = \mathcal{L}_{\mathcal{M}}(\theta = \mathcal{H}(z, \nu)). \tag{3.1}$$

Note that when a hypernetwork is trained to produce the parameters θ of a target network, these parameters θ are no longer trainable parameters, but simple computed values.

3.2 Input wise

Hypernetworks can be differentiated by the nature of their input z. There are many types of input for hypernetworks. Here, we focus on 3 categories: input hypernetwork, task-based hypernetwork, and noise-based hypernetwork. An overview is given in Figure 3.2. These types of input are presented separately, but can be combined together by mixing them before feeding them to the hypernetwork.

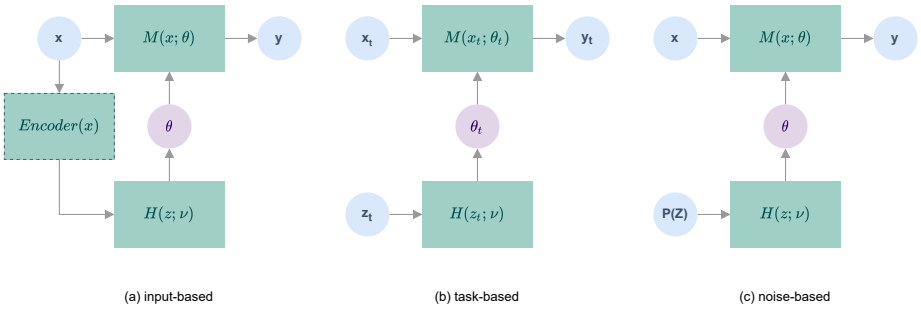


Figure 3.2: Different types of inputs for hypernetworks

3.2.1 Input of interest

A hypernetwork can simply use the input x of the main network as its own input. This input x may need to be encoded before using it as input for the hypernetwork. Sometimes, the input of a neural network naturally decomposes into a main part and a conditioning part. In this case, it might be useful to use an input-based hypernet. At the level of a single layer, the input-based hypernet corresponds to computing some of the layer parameters based on the current incoming feature maps.

Input-based hypernets allow to dynamically compute parameters specific for the current sample. This level of dynamism can greatly improve the performance of the model. However, they come with a cost. Indeed, a set of parameters has to be computed for every single example with a priori no hope of sharing or reusing these parameters with other examples. This is particularly costly at training time, where each element in a batch would need its corresponding parameters, although the cost depends on the actual number of predicted parameters.

3.2.2 Task

Another natural input for the hypernetwork is a *task* or a *context* embedding $z \in \mathbb{R}^d$. In this case, hypernetworks are trained on a set of tasks. The hypernetwork parameters are shared for each task, which only differ by the input z. These task-conditioned hypernetworks are less costly than the input-based one during training, since one can only predict one set of target parameters per batch if the batch contains only elements of the corresponding task. These tasks typically correspond to predefined tasks that are given at test time.

A task-conditioned hypernetwork with a set of tasks $\{0, 1, 2, ..., T-1\}$, a distribution over tasks $p(\tau)$, and the corresponding loss functions $\mathcal{L}^t_{\mathcal{M}}$, can be trained with the following loss function by taking the expected loss function over the tasks.

$$\mathcal{L}_{\mathcal{H}}(\nu, z_{0,...,T-1}) = E_{t\sim p(\tau)} \left\{ \mathcal{L}^t_{\mathcal{M}}(\theta = \mathcal{H}(z_t, \nu)) \right\} \tag{3.2}$$

3.2.3 Distribution

Instead of using an input with a predefined meaning, one could sample z from a controlled distribution $p(Z)$. This kind of input can be used to create an ensemble model. Once the hypernet has been trained, an arbitrary number of models can be sampled from it. This kind of model produces an implicit and nontrivial distribution over the main model parameters. This can be used to quantify the uncertainty of the main model.

A noise-based hypernetwork can be trained with the following loss function.

$$\mathcal{L}_{\mathcal{H}}(\nu) = E_{z\sim p(Z)} \left\{ \mathcal{L}_{\mathcal{M}}(\theta = \mathcal{H}(z, \nu)) \right\} \tag{3.3}$$

3.3 Output wise

Hypernetworks produce parameters of deep neural networks, which may have a large number of parameters. Hypernetworks may be ***complete***, producing the entire target model, or ***partial***, producing only some parts of the target model. The choice of producing all parameters or only a part of them is up to the designer of the model. It may be inefficient or unwise to produce all the parameters of a neural network, depending on the application and its goal. Hypernetworks typically adds a computational overhead because they generally need to produce the parameters before applying them in the target neural network.

3.3.1 Layer only hypernetwork

Many hypernetworks are very local and produce only dynamic parameters for a single layer. In this type of hypernetwork, the hypernetwork itself generally acts in a compressive regime, having fewer parameters or the same number of parameters as the target layer. Examples of this are rescaling hypernetwork that simply learns a scaling vector for a target matrix. This vector rescales every column/row of the target matrix [Ha et al., 2016; Tay et al., 2020]. Widely used attention layers [Vaswani et al., 2017a] can be seen as hypernetworks that dynamically produce the present layer parameters based on the input. Dynamic convolutional layers [Klein et al., 2015] are also an example in which some filters of a given layer are produced based on the current feature maps with a hyperconvolutional module.

3.3.2 Complete hypernetwork

Another way to use hypernetworks is to produce the entire target model. This may be used if the target model needs to be entirely dependent on some variable z or if there is no a priori shareable part in the target model. Producing the entire model makes the z variable more interpretable as a model indexer and the hypernetwork as a *meta model*. In this section, we review four strategies to build complete hypernetworks: a global strategy, two shared model strategies, and a local strategy, which are summarized in Figure 3.3. The global strategy is based on a single latent vector z, while the shared model strategies generally use some vector embedding of the place where a shared model should predict parameters.

There are different methods that one could use to produce θ, which are explained below. The simplest is to use a single model with a single forward pass to predict θ from z, for example, with an MLP. This is the global strategy. However, using a single neural network to produce all the parameters of another neural network at once is not necessarily trivial because of the size of the output space required. The next two strategies, chunking and shared layer-wise, involve splitting the parameter space and reusing the same model multiple times. The

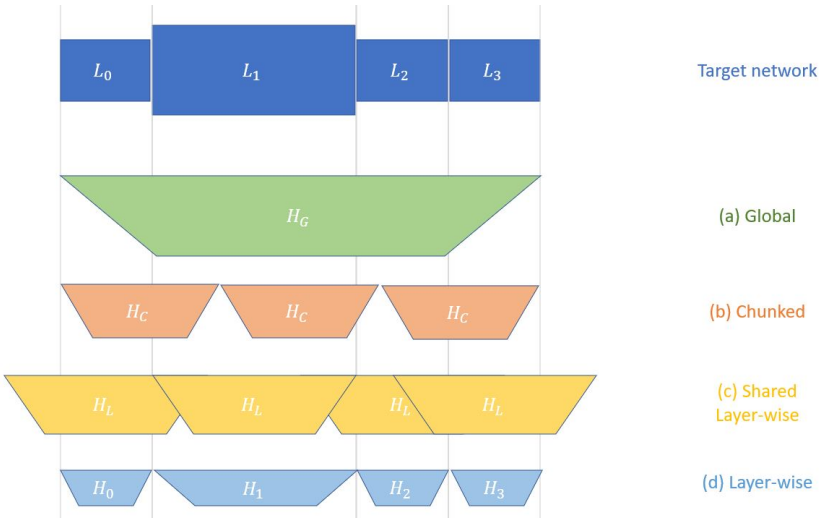


Figure 3.3: High level architectures for complete hypernetworks

last, a layer-wise strategy, builds different small hypernets for each of the layers. We describe below these four strategies in details.

Global

The first strategy uses a fully connected MLP as the core of a hypernet. This kind of model would be limited to small targets, as the number of parameters of such hypernet would typically not be scalable. Indeed, an MLP with hidden size proportional to the size of the output layer n would scale quadratically with n, which is not feasible in practice. We call this strategy of producing all parameters with a single nonshared model a **global** complete hypernetwork. This strategy produces the entire target based on a single vector z and, a priori, every part of the target model depends on every part of the vector z (or most of it). We call this property a high *global connectivity* between the z vector and the output parameters. This kind of model typically produces one large vector that contains all target parameters. This work focuses on complete global hypernetworks and proposes a sparse MLP as a solution to the scalability issue. Other sparse solutions, such as deep deconvolutional networks, have not been studied in this work and have not really been studied so far in the literature. A priori, a deconvolutional network only seems less appropriate if there are no optimal target parameters with the required spatial structure. However, this remains to be examined further, as deconvolutional layers have very few parameters compared to the fully connected equivalent, which is a nice feature for hypernetworks.

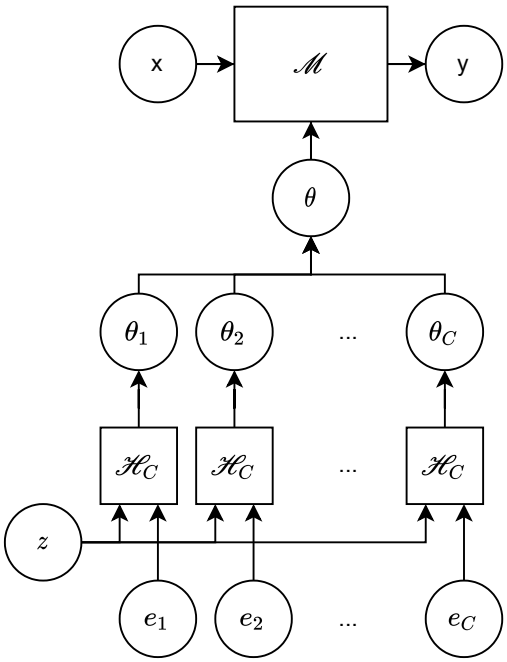


Figure 3.4: Illustration of a chunked hypernetwork $\mathcal{H}_C$ producing the parameters θ of the target model $\mathcal{M}$

.

Chunking

Because the size of the output space is too large, why not directly reduce this size by splitting the target parameters into chunks ? This is the idea of **chunking** [Pawlowski et al., 2018; von Oswald et al., 2020; Ehret et al., 2021]. A chunked hypernetwork splits the target parameters vector into C chunks of at most $\frac{n_\theta}{C}$ parameters, where n_θ is the number of parameters of the target network. Then a shared hypernetwork model $\mathcal{H}_c$, which could be implemented by an MLP, is used to produce each of the chunks based on a chunk embedding e_c:

$$\begin{aligned} \theta_c &= \mathcal{H}_C(z, e_c) \qquad \forall c \in \{1, 2, ..., C\}, \\ \theta &= \langle \theta_1, \theta_2, ..., \theta_C \rangle, \end{aligned}$$

where θ_c is the predicted chunk of parameters, z is a global embedding, and e_c are chunk embedding vectors. $\langle \theta_1, \theta_2, ..., \theta_C \rangle$ denotes the concatenation of the θ_c vectors into a single vector. An illustration is shown in Figure 3.4.

The great advantage of chunked hypernetwork is that the output size can be reduced arbitrarily low by increasing the number of chunks C. This removes the scalability issue of hypernetworks. It can also be used in a compressive regime where the number of parameters of the hypernet $|\mathcal{H}_c| + C \times |e_c|$ is lower than the number of predicted parameters n_θ. This means that the chunked hypernet can

be compared to other non-hypernetwork models while keeping the same number of trainable parameters.

However, a possible downside of chunked hypernetwork is that the target model is fragmented arbitrarily. Therefore, some chunks may produce only part of a layer, while others produce parameters for different layers (in the same chunk). Reusing the same model to produce such a variety of chunks may require a powerful hypernetwork with expressive enough chunk embeddings. Therefore, a more general and arbitrary target model architecture could make the work of a chunked hypernetwork more difficult. In practice, however, chunked hypernetworks have proven to be very effective.

Shared layer-wise.

Instead of predicting arbitrary chunks as in the previous case, a **shared layer-wise** involves a shared hypernetwork $\mathcal{H}_L$ that is reused to produce the parameters of layers $\theta_l \in \mathbb{R}^{n_l}$ (th n_l parameters of layer l in a network of L layers):

$$\begin{aligned} \theta_l &= s(\mathcal{H}_L(z, e_l), n_l) \quad \forall l, \\ \theta &= \langle \theta_1, \theta_2, ..., \theta_L \rangle \end{aligned}$$

where $e_l \in \mathbb{R}^p$ is a trainable layer embedding, z is a global embedding vector and $s(v, n)$ is a function that selects the n first elements of the vector v ($s(v, n) = (v_i)_{1 \leq i \leq n}$).

Since each layer of the target model may have a different number of parameters, the hypernetwork should be able to produce enough parameters for the largest target layer. If the hypernetwork produces too many parameters for a layer, the predicted parameter vector is sliced accordingly.

The advantage of this type of hypernetwork is that it allows to reuse a hypernetwork for different layers instead of arbitrary chunks. In the particular case of a hypergraph neural network ([Zhang et al., 2018; Knyazev et al., 2021]), a shared node network can be used to produce each of the layer parameters. In addition, it can be applied to any architecture, even at test time. Note that in this case, the positional embedding e_l of the parameters position in the target model are not required, the hypergraph network has information about the position through the structure of the graph itself. This leads to an interesting search for neural architectures, in which the parameters of a new architecture can be predicted.

However, this type of hypernet is still limited by the size of the largest layer, which may even be infeasible for fully connected models. The varying size of the layers may require to use a more complex architecture where multiple shared hypernetworks are used, one for each layer size or layer type.

Layer-wise

Instead of reusing a model to predict multiple layers, the layer-wise strategy actually learns a small hypernetwork for each layer. This kind of hypernetwork is very local, as each layer is adapted independently if nothing is shared. Examples

of this are rescaling hypernetworks, where the parameters of a target network are learned with rescaling factors. Typically, the parameters of a neural network are matrices or other tensors. One can learn rescaling vectors that rescale the column/row of the target parameters. The rescaling vectors are much smaller than the target parameters and therefore only add a small overhead in terms of the number of parameters. Any other technique used for predicting a single layer can typically be applied to all layers to produce a 'complete' hypernetwork, such as attention or dynamic convolution.

3.4 Size

Hypernetworks may have different sizes (number of parameters) for a given target model. In particular, a hypernet can be in a **compressive** regime if it has fewer parameters than the target model. However, it should be noted that even though a hypernetwork has more parameters than its target model, it may still compress several target models into a single hypernetwork and the corresponding embeddings.

3.5 Summary and conclusion

In this chapter, we give a formal definition of hypernetworks. We explain three types of input for hypernetworks for dynamism, task-conditioning, and randomization. These inputs of hypernetworks can be combined depending on the application.

We differentiate hypernetworks based on their output. Hypernetworks can produce only some parts of the parameters of a neural network or all of them for complete hypernetworks. Four high-level architectures were reviewed that can deal with the potentially large output space of complete hypernetworks. We reviewed them from the more global to the more local methods: the global hypernetworks, the chunked and shared layer-wise methods which reuse a model at different locations, and the non-shared layer-wise method which uses multiple small hypernetworks for each layer of the target model.

Chapter 4

Sparse hypernetwork

4.1 Introduction

Hypernetworks are models that produce the parameters of another target model. Typical deep learning models may easily have millions of parameters or more. Hypernetworks need to be scalable in terms of the number of parameters and computing time, so that it is feasible to produce a huge amount of parameters for a deep learning model. Indeed, naive solutions to this problem may not be scalable at all. For a simple linear layer with input dimension d and output dimension n, the number of parameters of the target network would require $d \times n$ parameters. This may seem reasonable, but we often need d to be sufficiently large, as it represents the maximum intrinsic dimension of the space where the target networks vary with respect to the latent space for a given hypernetwork.

A current solution for building scalable hypernetworks that does not require knowing the maximum number of parameters of the target layers is known as chunking. Chunked hypernetworks divide the output space into chunks. The predicted parameters are merged into an n dimensional vector that is split into C chunks of the same size $\lceil \frac{n}{C} \rceil$[1]. The hypernetwork consists of a single model $\mathcal{H}$ that produces the $\lceil \frac{n}{C} \rceil$ parameters for a given chunk. The model is applied C times to C chunk embeddings of dimension d_c to produce the n parameters of the target model. Chunked hypernetworks have been proven to be sufficiently expressive to learn any target network (von Oswald et al. [2020]).

In this chapter, we propose a new architecture for building scalable hypernetworks. This architecture takes the form of a sparse neural network with hidden layers of exponentially growing size that rapidly reach the possibly large size of output space required by a hypernetwork.

[1] Note that the last chunk may be truncated, the last values being ignored

4.2 Sparse hypernetworks

In this chapter, we first describe the general idea of sparse neural networks for hypernetworks. We then describe different ways of choosing connections and analyze them in terms of the resulting number of parameters to show that they are scalable. We then describe how to initialize these hypernetworks and how to normalize their inputs and rescale their output to stabilize training. Finally, we explain how we train task-conditioned sparse hypernetworks.

4.2.1 General idea

As noted before, classical fully connected layers from latent dimension d to parameter space of size n do not scale at all. Sparse hypernetworks try to make these architectures scalable in terms of number of parameters by making them extremely sparse.

The sparse hypernetwork is essentially a sparsely connected MLP. We use exponentially increasing hidden layer size in order to go from latent dimension d to dimension n. This strategy allows to reach the possibly huge dimension n quickly enough from a far smaller dimension d, avoiding too deep hypernetworks. An illustration of the considered architecture is shown in Figure 4.1.

We have the following hidden layer sizes (with h_0 the input size and h_L the output size).

$$\begin{aligned} h_0 &= d \\ h_l &= \left\lfloor \frac{n}{b^{L-l}} \right\rfloor \forall l \in [1, 2, ..., L] \\ h_L &= n, \text{ with } L = \lfloor \log_b(n/d) \rfloor \end{aligned}$$

In what follows, we will consider an expansion factor b of 2. However, this number can be adapted depending on the case. With $b = 2$, the number of neurons in the sparse hypernetworks will be (excluding inputs and neglecting round-off errors): $\sum_{l=1}^{L} h_l \approx \sum_{l=1}^{L} \frac{n}{2^{L-l}} = \frac{n(2^{L+1}-2)}{2^L} \approx 2n$. For each intermediary vector h_l, we can define a connectivity factor c_l. The factor c_l is such that on average a neuron in h_l is connected to c_l neurons from the previous layer.

If c_l is known, the total number of parameters in the hypernetwork can be estimated as follows.

$$n_{hnet} \approx \sum_{l=1}^{L} c_l \times h_l \tag{4.1}$$

If this architecture was fully connected, then the number of parameters would be in the order of n^2 (Equation 4.7), which is not scalable to large target networks in practice.

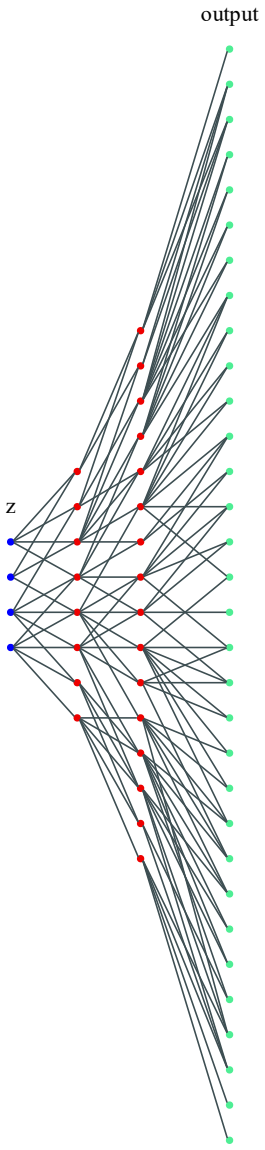


Figure 4.1: Illustration of a sparse hypernetwork. Three sparse layers transform the 4-dimensional input z (blue) into 32 parameters (green). The hidden sizes increase exponentially by a factor 2.

$$n_{hnet} \approx \sum_{l=1}^{L} c_l \times h_l \tag{4.2}$$

$$= \sum_{l=1}^{L} h_{l-1} \times h_l \tag{4.3}$$

$$\leq \sum_{l=1}^{L} \frac{n}{2^{L-l+1}} \times \frac{n}{2^{L-l}} \tag{4.4}$$

$$= \frac{n^2}{2 \times 2^{2L}} \sum_{l=1}^{L} 4^l \tag{4.5}$$

$$= \frac{2n^2}{2 \times 4^L} \frac{1 - 4^{L+1}}{1 - 4} \tag{4.6}$$

$$\leq \frac{2n^2}{3} \tag{4.7}$$

4.2.2 Connectivity

As seen in Equation 4.1, the number of parameters of the hypernetwork depends on c_l. Below, three connectivity patterns are proposed with an approximation of the number of parameters. The first one is the simplest, with a constant number of connections per output neuron. Then we propose two improvements that reduce the connectivity in higher dimensional layers and increase it in lower dimensional ones : linearly and exponentially decreasing connectivity. We compute below the number of parameters of these three connectivity patterns. How the connections are distributed will be discussed in Section 4.2.3. Visual examples of the connectivity patterns are shown in Figures 4.2 and 4.3.

With or without bias term

Sparse hypernetwork can be used with or without bias terms. For a sparse hypernetwork, the number of parameters added by using bias terms is non-negligible. It will add $\sum_{l=1}^{L} h_l \approx 2n$ parameters for a target network of n parameters.

Constant

A constant $c_l = c$ is used. Equation 4.1 then becomes:

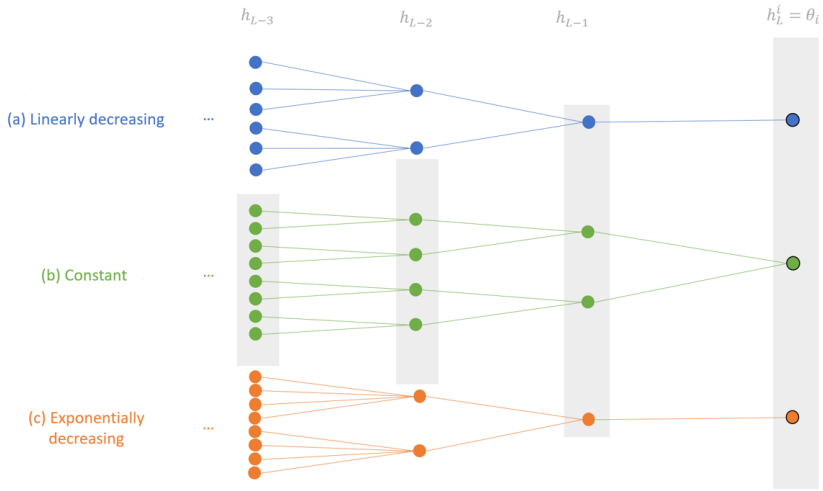


Figure 4.2: Illustration of the hidden neurons that are connected to a particular output neuron in the last layers of a sparse hypernetworks. In **(a) linearly decreasing**: the output neurons is connected to **1** previous neuron, which itself is connected to **2** previous neurons, which are themselves connected to **3** previous neurons and so on. In **(b) constant**: each output neurons is connected to the same number of input neurons: **2**. **(c) exponentially decreasing**: Starting from the output neuron, there are first **1** connection, then **2** connections, then **4** connections an so on.

$$n_{hnet} = \sum_{l=1}^{L} c \times h_l \tag{4.8}$$

$$\leq \sum_{l=1}^{L} c \times \frac{n}{2^{L-l}} \tag{4.9}$$

$$= \frac{cn}{2^L} \sum_{l=1}^{L} 2^l = \frac{cn(2^{L+1} - 2)}{2^L} \leq 2cn \tag{4.10}$$

Linearly decreasing

In order to increase the connectivity in the lower-dimensional space and lower it in the high-dimensional space, we can use linearly decreasing connectivity. Here, we choose $c_l = L - l + 1$. Equation 4.1 then becomes:

$$n_{hnet} = \sum_{l=1}^{L} (L - l + 1) \times h_l \tag{4.11}$$

$$\leq \sum_{l=1}^{L} (L - l + 1) \times \frac{n}{2^{L-l}} \tag{4.12}$$

$$= \frac{n}{2^L} \left[(L+1) \sum_{l=1}^{L} 2^l - \sum_{l=1}^{L} l 2^l \right] \leq 4n \tag{4.13}$$

The proof of 4.13 is given in Appendix A.1.

Exponentially decreasing

An alternative to linearly decreasing connectivity is an exponentially decreasing one. Here, we choose $c_l = min(2^{L-l}, h_{l-1})$, and the equation 4.1 becomes:

$$n_{hnet} = \sum_{l=1}^{L} min(2^{L-l}, h_{l-1}) \times h_l \tag{4.14}$$

$$n_{hnet} \leq \sum_{l=1}^{L} 2^{L-l} \times h_l \tag{4.15}$$

$$\leq \sum_{l=1}^{L} 2^{L-l} \times \frac{n}{2^{L-l}} \tag{4.16}$$

$$= Ln = \lfloor \log_2(n/d) \rfloor n \leq n \log_2(n) \tag{4.17}$$

Connectivity	c_l	#params w/o bias	#params w/ bias
Fully connected	h_{l-1}	$\frac{8n^2}{3}$	$\frac{8n^2}{3} + 2n$
Constant	c	$2cn$	$2(c+1)n$
Linearly decreasing	$L - l + 1$	$4n$	$6n$
Exponentially decreasing	$min(2^{L-l}, h_{l-1})$	$n \log_2(n)$	$n(\log_2(n) + 2)$

Table 4.1: Different types of connectivity and upperbounds on the number of parameters. The upper bounds on the number of parameters do not depent on d the size of z. For comparison a single linear layer hypernetwork would require $d \times n$ parameters without bias and $(d+1) \times n$ with bias, which prevent from using large d.

Summary

A summary of the upper bounds on the number of parameters can be found in Table 4.1 and a visual example of these sparse hypernetworks is given in Figure 4.3.

4.2.3 Distribution of connections

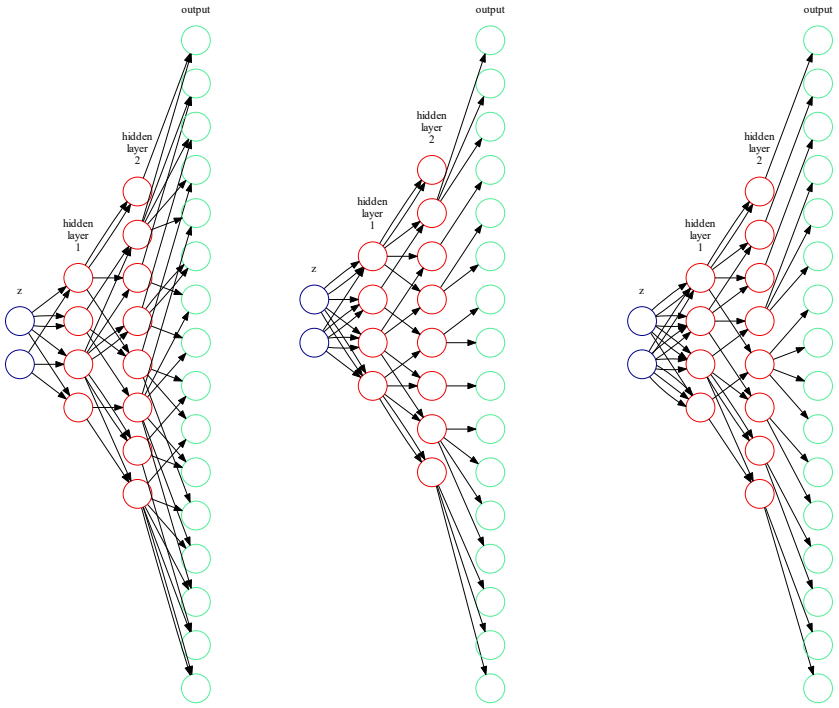
The extremely sparse networks presented above have only a few connections per neuron. The question of how to connect them arises naturally. Many sparsification methods (Blalock et al. [2020]) start from fully connected layers and try to remove connections. However, it is not really possible to represent an extremely sparse expanding network in a dense manner. Indeed, it would require $O(n^2)$ parameters (Equation 4.7), which is often prohibitive. Here, we focus on randomization in order to choose the connections. The connections will be generated according to a random process.

An interesting property of extremely sparse expanding networks is its global connectivity. By global connectivity, we mean how many input values are connected to each output neuron on average. If the global connectivity is maximal, every output neuron will depend on every input value. In a low global connectivity, each output neuron will depend on a few input values, but will be independent of the others. Figure 4.4 shows how the mean connectivity progressively increases within sparse hypernetworks with different connectivity patterns.

We start by presenting the simplest distribution of connections, the uniform one. Then we propose a variation with random permutation. Then we present two more local connectivity distributions that allow to control the connectivity: the Gaussian distribution and the mixed one.

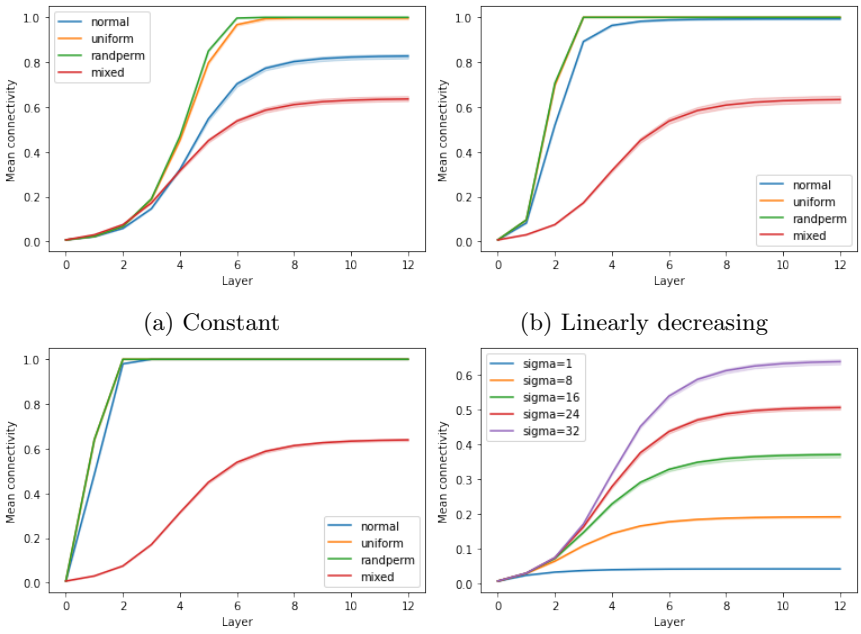
Uniform

The simplest way of choosing connections is to choose them uniformly at random. For each output neuron o_l^i, we draw c_l input neurons (with replacement) to connect it to. Note that by sampling the connections in this way, there will



(a) Constant (b) Linearly decreasing (c) Exponentially decreasing

Figure 4.3: Example of actual sparse hypernetworks with constant (2 connections), linearly decreasing and exponentially decreasing connectivity. The input z has dimension 2 and the output has a dimension 16.



(a) Constant

(b) Linearly decreasing

(c) Exponentially decreasing

(d) Constant and normal distribution

Figure 4.4: Mean connectivity of the input neurons for each hidden layer of the sparse hypernetwork with different configurations. The hypernetwork goes from a latent size of 2^7 to an output of size 2^{20} ($\approx$ 1M) in 13 sparse linear layers. The graph shows the percentage of hidden neurons an input neurons is connected to in a given layer. The mixed pattern is repeated in a, b and c for comparison.

be approximately $\frac{1}{e^{bc_l}} h_{l-1}$ neurons that are not chosen (A.2), which is approximately 13% for a base factor $b = 2$ and $c_l = 1$. This will reduce the effective size of the hidden layer h_l. Furthermore, it will reduce the number of parameters as if there were only $c_l(1 - e^{-bc_l})$ connections per neuron o_l^i.

This method intuitively provides high connectivity between the input and outputs. However, it does not allow to control the global connectivity. The only way to reduce it would be to have a shorter network (higher b or lower d).

Random permutations

The uniform distribution may leave a proportion of input neurons without any connections. Although each output neuron has the same number of connections, each input neuron may have a varying number of connections. Random permutations aim to avoid this behavior by being more deterministic. This connectivity pattern is equivalent to the uniform one without replacement. Suppose that, at the hypernetwork layer l, we have a random permutation σ of the indices in $\{0, 1, ..., c_l \times h_{l-1}\}$. Then each output neuron o_l^i has c_l connections to

$\{\sigma(i+k) \mod h_l | k \in \{0, 1, ..., c_l\}\}$. With this type of connectivity, each input neuron o_l^i is connected to (almost) exactly $c_l \frac{h_{l+1}}{h_l}$ connections, which is slightly higher than for the uniform one. Therefore, the connections are more evenly distributed with respect to the inputs.

Gaussian

Contrary to the previous method, the Gaussian connection provides a parametric way to choose connections. Intuitively, this method draws connections by choosing input neurons according to a discretized Gaussian distribution centered on the considered output neuron. More formally, we proceed as follows.

For a given output neuron o_l^i, expansion factor b and number of connections c_l, we draw a value $\hat{j}$ according to a Gaussian distribution $\mathcal{N}(\frac{i}{b}, \sigma^2)$. Then we add a connection between o_l^i and o_l^j with $j = max(min(round(\hat{j}), h_{l-1}), 0)$. An illustration of the difference with the uniform distribution is shown in Figure 4.5.

Note that there can be collisions as for the previous method. These collisions will reduce the number of parameters as they only count for one connection from i to j.

This method is parameterized by σ. We always assume that σ is smaller than $d = h_0$ (typically $\sigma \leq \frac{h_0}{4}$) and that it is therefore much smaller than other h_l. A higher σ would increase the connectivity, while a smaller one would increase locality, decrease the connectivity, and reduce the number of parameters.

In most of our experiments, we chose $\sigma = \frac{h_0}{4}$ for all layers except the last one for which we use $\sigma = 0$ in case $c_L = 1$ in order to avoid missing too many input neurons.

Decreasing σ Depending on the connectivity c_l chosen, there may be fewer connections per neuron in the latest layers. This means that there is a higher chance that neurons are left without connections and therefore that they are wasted. Furthermore, as the activation passes through the network, reducing the spread of the connections will transform the layers from being global to being more local and specialized into their region. We also advise that the last layer is more deterministic as there can be only one connection per neuron, that is, setting the variance to 0. One strategy to reduce the spread of the distribution for deeper layers would be to decay the standard deviation linearly to 0 as follows in Equation 4.18.

$$\sigma_l = \frac{d}{4} - \frac{dl}{4L}, l \in [1, 2, ..., L] \tag{4.18}$$

Through most of our experiments, we kept σ as:

$$\sigma_l = \begin{cases} \frac{d}{4}, & \text{if } l < L \\ 0, & \text{if } l = L \end{cases} \tag{4.19}$$

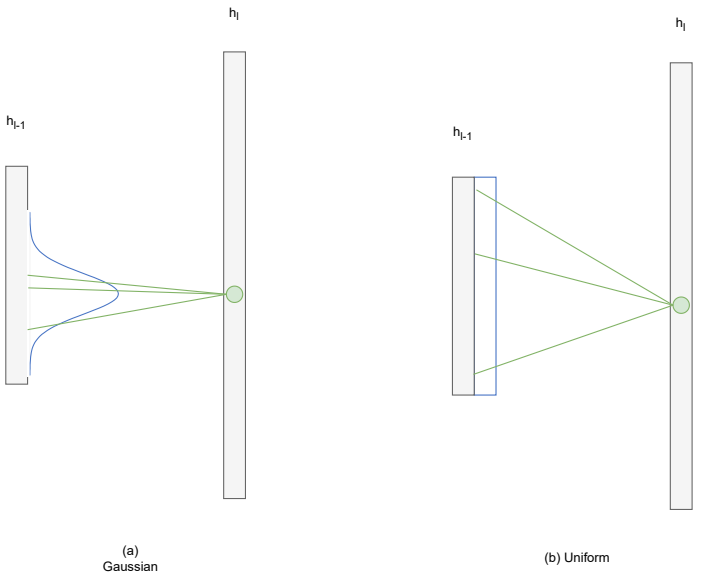


Figure 4.5: Illustration of Gaussian distribution vs uniform one in a sparse layer.

Mixed

A mixed distribution has a specific connectivity. In this case, we use $c_L = 1$ and for the other layers $c_l = 4$. With this pattern, the number of parameters is upper bounded by $5n$ without bias term and $7n$ with. It has n parameters in the last layer and $4n$ for the previous layers which can be derived from a constant connectivity with a target of $\frac{n}{2}$ ($4n = 2 \times 4 \times \frac{n}{2}$), hence the $5n = n + 4n$ upper-bound on the number of parameters.

The mixed distribution mixes a deterministic pattern and a random one. This distribution is used with a specific number of connections c_l. The idea is to use a very local deterministic pattern and increase the global connectivity by adding a few random connections. For the last layer, each output o_L^i is deterministically connected to $o_{L-1}^{\lfloor i/b \rfloor}$. For the other layers l, o_l^i are deterministically connected to $o_{l-1}^{\lfloor i/b \rfloor - 1}$, $o_{l-1}^{\lfloor i/b \rfloor}$ and $o_{l-1}^{\lfloor i/b \rfloor + 1}$. This connectivity is similar to a convolution with a window of size 3 (except that there is no shared filter). In addition, we add one connection for each o_l^i that is drawn according to a Gaussian distribution as previously described. This connection will increase the global connectivity. In total, each of these o_l^i has 4 connections.

4.2.4 Initialization and Normalization

Hypernetworks are different from other neural networks in the sense that their output values are themselves parameters of other networks. It is well known that the initialization of deep neural networks is crucial for proper convergence.

It follows that hypernetworks should also be properly initialized so that they produce proper parameters for the target network at initialization. Typical initialization of deep learning model parameters is done via a variance analysis. Chang et al. [2019] explains how to initialize the hypernetwork parameters so that the predicted parameters follow more closely the usual initialization pattern.

The usual method for initializing hypernetworks is to change the initialization of the last linear layer in order to have proper output values based on the variance analysis.

However, this makes the assumption that you already know which layer will be predicted by your network, which is not always the case. Indeed in the case of a chunked hypernetwork or a graph neural network, the same model may be reused in order to produce different layers, which may require largely different scales. This brings us to the second assumption that is made about the input of the hypernetwork. These methods require that the variance of these inputs is known and can be controlled. In the case of a chunked hypernetwork, because the same model is reused for different layers, the scale of the output layer could be controlled by changing the scale of the input vector. However, while the chunk embedding vectors are different for each chunk, one cannot change a task vector embedding for each embedding without making it a task-chunk embedding vector. This would make the separation less clear between what is specific to the task and what is specific to a chunk/layer.

Furthermore, in a more general case, the inputs of the hypernetwork are not necessarily trainable parameters. They may be the results of more complex computations, over which we have no control. In that case, these inputs may have a too large or too small variance that would break the assumption made by the initialization scheme.

To shield the core of the hypernetwork from any dependence on inputs and outputs scale, we propose to 1) normalize the inputs and 2) add a rescaling layer after the hypernetwork (similarly as the operation-dependent normalizations from Knyazev et al. [2021]). An illustration is shown in Figure 4.6.

The core of the hypernetwork is set to simply preserve the forward variance. With the normalization of the input and the variance preservation, we can assume that the output of the hypernetwork will have a variance of approximately 1. Then we make each target layer predicted parameters go through a specific rescaling. This rescaling layer simply multiplies all values by a constant factor in order to transform the variance of 1 to the desired layer-specific variance (Table 4.2). The fully connected layers are rescaled according to a Xavier initialization scheme (Glorot and Bengio [2010]) and the convolution according to a fan-in scheme (He et al. [2015a]). The bias term are also scaled down even though it is not required. There is no particular justification for the difference in the initialization scheme for fully connected layers and convolutional layers.

Target layer	Rescaling factor
Fully connected	$\sqrt{\frac{6}{3c_{in}c_{out}}}$
Convolutional	$\sqrt{\frac{1}{3c_{in}k_0k_1}}$
Bias	$\sqrt{\frac{1}{3b_{size}}}$

Table 4.2: Rescaling factor used for each target layer parameter type, where c_{in} and c_{out} are respectively the number of input and output channels, k_o and k_1 are the dimension of the kernels and b_{size} is the dimension of the bias term.

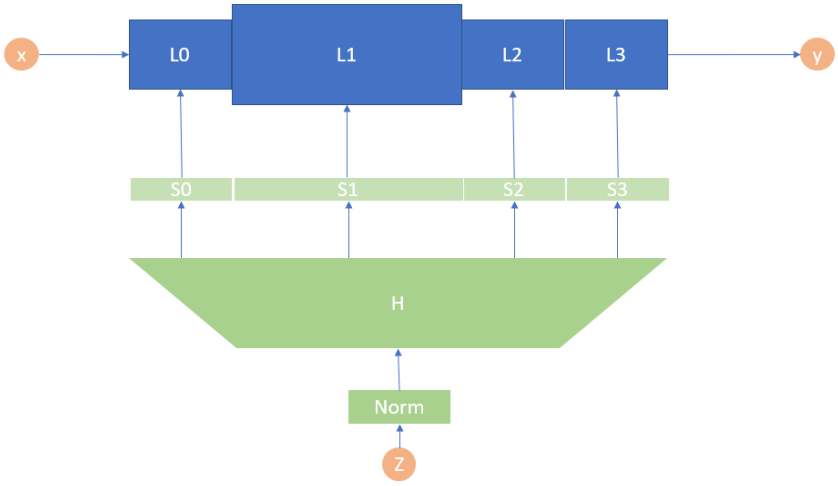


Figure 4.6: Illustration of input normalization and output rescaling for hypernetworks. The input z is normalized. The ouputs are rescaled with different scaling factor (S_0, S_1, S_2, S_3) adapted for each target layer (L_0, L_1, L_2, L_3).

4.2.5 Training in multitasking setting

The hypernetworks are trained using the classical procedure: by gradient descent on the target network loss function. More specifically, task-conditioned hypernetworks can be trained according to the following loss function based on 3.2:

$$L(\nu, z_{0,\ldots,T-1}) = E_{t\sim P(\tau)}\left\{E_{x,y\sim P(X,Y|t)}\left\{l(y, \mathcal{M}(x, \theta = \mathcal{H}(z_t; \nu))\right\}\right\} \quad (4.20)$$

The loss function is estimated by sampling some tasks and the corresponding data from the training set. In most of our experiments, we sample only 3 tasks corresponding to 3 batches. Each batch corresponds to a single task, allowing one to produce only one task-specific model per batch. The gradients of these 3 batches are then accumulated. We also always use gradient clipping to prevent the effects of unwanted big gradients.

Chapter 5

Experiments

In this chapter, we perform a set of experiments with sparse hypernetworks on computer vision multitasking problems.

We start by giving a brief overview of the datasets and protocol used in the experiments in Section 5.1.

In Section 5.2, we test different combinations of sparse hypernetwork hyperparameters to understand their effect.

Next, in Section 5.3 further compare the sparse hypernetwork with the current alternative of chunked hypernetworks and with the individual experts trained on each task.

Section 5.4 further continues this comparison but with sparse target networks. The aim is to see if the different types of hypernetworks behave differently as the target networks become sparser.

Finally, in Section 5.5, we test the capacity of hypernetworks to generalize to new tasks. We continue to compare different types of hypernetworks. In particular, we compare the generalization and transfer learning power of linear and non-linear sparse hypernetworks as well as chunked hypernetworks. We try to see how the different reprenstation capacities of these hypernetworks affect their generalization to new tasks.

All the experiments presented concern multitasking classification problems. Often, however, a single accuracy computed on a test set is presented; it is the average accuracy for all tasks.

5.1 Datasets and protocol

All the experiments are performed on the following datasets. They are multitasking computer vision problems. They are all based on classification problems from which the classes are split into multiple classification tasks. These multitasking problems are adapted from well-known computer vision benchmarks and allow to easily test and compare different models.

Split MNIST 10/5. This dataset is based on MNIST Deng [2012], which contains 10 classes. In Split MNIST 10/5, this 10-ary classification problem is split into five binary tasks. The first binary task consists in differentiating zeros and ones, the second twos and threes, and so on. Therefore, all tasks have disjoint domains. The 10 classes are separated as follows: each image with label $l \in \{0, 1, 2, ..., 9\}$ belongs to the task $\lfloor \frac{l}{2} \rfloor$.

Split CIFAR 10/5. The split CIFAR 10/5 problem splits the CIFAR10 dataset (Krizhevsky [2009]) into five binary classification tasks. The 10 classes are separated into 5 pairs of class and each task aims to predict the new binary label of an image. The 10 classes are separated similarly as in split MNIST/5.

Split CIFAR 100/10. Similarly to split CIFAR10/5, split CIFAR100/10 separates the 100-ary classification task into 10 tasks of 10 classes. It is much more difficult than split CIFAR10/5 as there are more tasks, more classes per task, and fewer examples of each class. The 100 classes are separated as follows: each image with label $l \in \{0, 1, 2, ..., 99\}$ belongs to the task $\lfloor \frac{l}{10} \rfloor$.

Protocol. The multitasking models are trained via gradient descent in a multitasking setting. We sample batches containing only examples of a single task, so that the hypernetworks only have to generate a single target model per batch. Additionally, we average the gradients over 3 batches. Often, we average the results over 5 trials (with the same train/validation split). More details about training and model hyperparameters are given in Appendix B.

5.2 Analysis of sparse hypernetworks

The present experiment aims to analyze different combinations of hyperparameters for sparse hypernetworks. More precisely, we compare different ways of selecting connections in the sparse hypernet as described in Section 4.2, comparing linearly decreasing, exponentially decreasing, and constant connectivity. We compare different distributions of connections: normal, uniform, rand-perm, and mixed distributions. More generally, we compare purely linear hypernets (without bias term) and MLP ones with PReLU activations (He et al. [2015b]). For each combination, we averaged the results over 5 trials (with the same split of dataset).

In this experiment, as in all other experiments, the task embeddings are normalized for the sake of demonstration, this is however not required in this case as these inputs are learned together with the hypernet parameters.

The target model is a ResNet-32 without batch normalization of 426K parameters from He et al. [2015c].

We tested two types of non-linearity, MLP with bias term and PReLU activations ($\alpha_{init} = 0.25$) and a purely linear hypernetwork without bias term, which is the simplest model and has the lowest number of parameters. The

Non linearity	Accuracy
MLP	78.68±1.21
Linear	77.96±1.30

Table 5.1: Accuracy of sparse hypernetworks for different non linearity on split CIFAR100/10

Distribution	Accuracy
mixed	78.88±1.07
normal	78.74±1.50
randperm	78.08±1.12
uniform	77.58±1.09

Table 5.2: Accuracy of sparse hypernetworks for different sparsity distribution on split CIFAR100/10

average and standard deviation of the results are shown in Table 5.1 (averaged over distribution and connectivity types). The MLP version did obtain a slightly higher accuracy; however, the linear version still achieves comparable performance, only 0.72 percent below.

With respect to the distributions (Table 5.2), all distributions performed comparably. However, the more local distributions (mixed and normal) performed slightly better than the nonlocal ones (randomperm and uniform). This may indicate that parameters that are closer to each other in the target network may need to share more information between them than parameters that are further away from each other. However, the gap between local and non-local distributions is not that large. These results are averaged over non-linearity and connectivity types.

For the different types of connectivity (Table 5.3), their performances are close. However, the constant connectivity type seems to have slightly lower accuracy. This indicates that the increase of connectivity at the lower-dimensional layers of the sparse hypernetworks has a positive effect on performance.

All aggregated results are shown in Table 5.4 for each combination of hyperparameters. For linear models, the best model is the one with normally distributed connectivity and exponentially decreasing with an accuracy of 80.12%.

Connectivity type	Accuracy
constant	77.67±0.92
exponential-decrease	78.59±1.39
linear-decrease	78.14±1.47
mixed	78.88±1.07

Table 5.3: Accuracy of sparse hypernetworks for different connectivity types on split CIFAR100/10

Linearity	Distrib.	Connectivity type	Accuracy	Params	Con.
MLP	mixed	mixed	79.18±0.98	2.95M	68.97%
	normal	constant	77.69±0.62	2.53M	84.72%
		exponential-decrease	79.07±1.11	2.69M	99.99%
		linear-decrease	80.10±0.57	2.53M	100%
	randperm	constant	77.33±1.08	3.41M	100%
		exponential-decrease	79.18±0.56	2.76M	100%
		linear-decrease	78.48±1.77	2.56M	100%
	uniform	constant	77.95±0.98	3.41M	98.75%
		exponential-decrease	78.61±1.41	2.76M	100%
		linear-decrease	78.18±0.51	2.56M	100%
Linear	mixed	mixed	78.58±1.09	2.1M	68.97%
	normal	constant	77.41±1.48	1.68M	84.72%
		exponential-decrease	80.12±0.71	1.83M	99.99%
		linear-decrease	78.04±1.63	1.67M	100%
	randperm	constant	78.20±0.54	2.56M	100%
		exponential-decrease	77.79±1.04	1.9M	100%
		linear-decrease	77.50±0.39	1.7M	100%
	uniform	constant	77.43±0.65	2.56M	98.75%
		exponential-decrease	76.77±0.58	1.9M	100%
		linear-decrease	76.52±0.62	1.7M	100%

Table 5.4: Accuracy of sparse hypernetworks for different hyperparameters on split CIFAR100/10

This model is well above all other sparse linear hypermodels. This model also has a relatively low number of parameters of $1.87M$ compared to the lowest number of parameters of $1.67M$. For the non-linear models, the normally distributed ones with linearly decreasing connectivity obtain the best results. This mean accuracy obtained of 80.10% is actually slightly smaller than the one of the best linear hypernetwork, although very close. Interestingly, this model is also the non-linear model with the least number of parameters ($2.53M$).

5.3 Comparison of hypernetworks architecture

5.3.1 Split MNIST

The split MNIST problem separates the 10 classes of MNIST into 5 binary classification tasks (0vs1, 2vs3, ..., 8vs9). This is a multitasking problem, where the tasks are similar (they are equivalent to the task: "Is that number even or odd?") but their domains are different (distinct here). The model is given an image and has to answer the binary class of the image. Furthermore, the model knows which task it is working on.

For this problem, the target network is a tiny neural network of only 190 parameters consisting of a few convolutions followed by a final linear layer with

```
ConvMNIST(
  (convs): Sequential(
    (conv0): Conv2d(kernel_size=(5, 5), stride=(1, 1)), PReLU
    (conv1): Conv2d(kernel_size=(5, 5), stride=(1, 1)), PReLU
    (conv2): Conv2d(kernel_size=(5, 5), stride=(1, 1)), PReLU
    (conv3): Conv2d(kernel_size=(5, 5), stride=(1, 1)), PReLU
    (conv4): Conv2d(kernel_size=(5, 5), stride=(1, 1)), PReLU
    (conv5): Conv2d(kernel_size=(5, 5), stride=(1, 1)), PReLU
  )
  (out): Linear(in_features=16, out_features=2)
)
```

Figure 5.1: Target model used in the split MNIST experiment.

Model	Accuracy	Parameters	Iterations
Chunked	99.02 ± 0.19	1003	28±6k
Experts	99.32 ± 0.12	950	17±4k
Sparse linear	94.07 ± 7.00	1019.5	25±7k
Sparse nonlinear	99.10 ± 0.15	1024.6	27±5k

Table 5.5: Average accuracy (n=10) for the split MNIST experiment comparing chunked and sparse hypernetworks to experts.

PReLU activations. The convolution operates on a single channel without bias term. The target model architecture is shown in Figure 5.1.

In this experiment, the task-conditioned sparse hypernetwork is compared with the task-conditioned chunked hypernetwork and task-specific experts. The models are trained with a batch size of 50 (iterations accumulate 3 batches), learning rate of 1e-3, RAdam optimizer (Liu et al. [2019a]), early stopping after 10 non-improving epochs. The latent size of z is 8. The chunked hypernets have 10 chunks of size 19. Note that the sparse linear hypernetwork uses an expansion factor of 1.6 instead of 2, which makes it deeper, so that it has a number of parameters comparable to the other models.

Table 5.5 displays the results of this experiment. The chunked hypernet, the non-linear sparse one, and the experts obtain similar accuracy on this simple task. The linear hypernet did not reach the same level of accuracy during this experiment.

Looking at the results per task in Table 5.6, the linear hypernet actually obtains similar results as the others for tasks 1 and 3, but not for tasks 0, 2, and 4.

Although the average accuracy of the linear hypernet is lower for certain tasks on average, this does not mean that each model has such an average accuracy. On the contrary, these models generally achieve similarly good performance in certain tasks, while being completely wrong in other tasks. Figure

Model	Task 0	Task 1	Task 2	Task 3	Task 4
Chunked	99.90 ± 0.11	98.22 ± 0.28	99.48 ± 0.27	99.43 ± 0.32	98.07 ± 0.64
Experts	99.93 ± 0.08	98.79 ± 0.34	99.81 ± 0.18	99.61 ± 0.16	98.44 ± 0.52
Sparse linear	89.95 ± 21.06	98.26 ± 0.30	94.62 ± 15.68	99.49 ± 0.34	88.03 ± 20.05
Sparse nonlinear	99.91 ± 0.09	98.65 ± 0.32	99.51 ± 0.25	99.47 ± 0.26	97.97 ± 0.52

Table 5.6: Average test accuracies (n=10) in percentage for each task on split MNIST. The linear hypernet obtained lower results on task 0, 2 and 4.

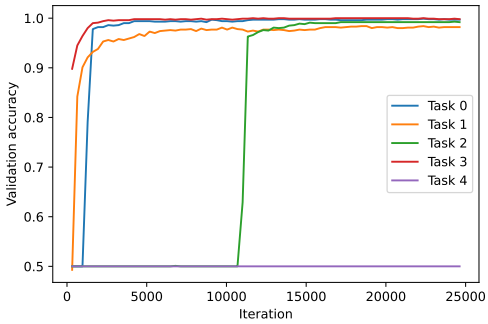


Figure 5.2: Validation accuracy for each task of split MNIST during training of a linear sparse hypernetwork

5.2 shows the different validation accuracies of each task for a sparse linear model. The model struggled to learn all the tasks in parallel and learned them sequentially. It first learned tasks 1 and 3, then 0, 2 later and never managed to learn binary task 4 on which it kept a 50% accuracy. We hypothesize that these poor performances are due to either a too small latent size for the linear hypernetwork (the latent vector is normalized), a too small learning rate, or simply due to the linearity of the model. The non-linear sparse hypernetwork also similarly learned tasks in a sequential fashion, as can be seen in Figure 5.3. The chunked hypernetworks did not show that kind of sequential learning (Figure 5.4).

5.3.2 Split CIFAR 10

We tested the models on split CIFAR10/5 with a Resnet-32 target network. The results are shown in Table 5.7. We can see that the non-linear sparse hypernetwork obtains the best mean accuracy of all models, closely followed by the chunked hypernetwork however. Additionally, all multitasking hypetnetworks exceeded experts on this problem.

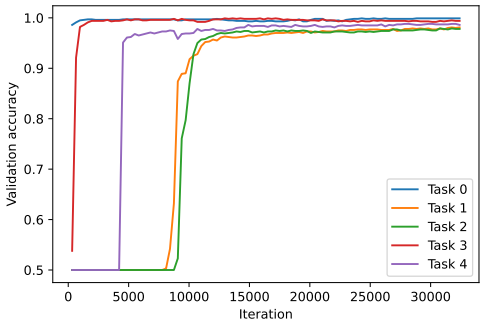


Figure 5.3: Validation accuracy for each task of split MNIST during training of a non linear sparse hypernetwork

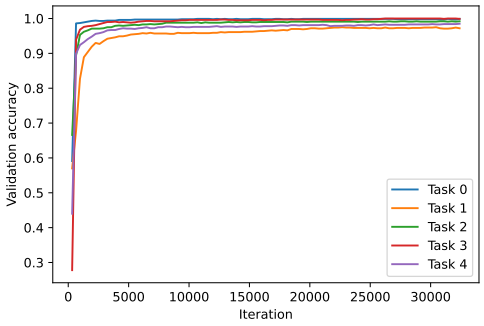


Figure 5.4: Validation accuracy for each task of split MNIST during training of a chunked hypernetwork

Model	Accuracy
Chunked	95.99±0.46
Experts	95.48±0.07
Sparse linear	95.70±0.27
Sparse MLPbias	**96.05±0.19**

Table 5.7: Mean accuracy of the multitasking hypernetworks and experts on split CIFAR10/5.

Model	Accuracy	Parameters	Iterations
Chunked	81.47±0.68	2.54M	13±1k
Experts	78.82±0.65	4.26M	13±1k
Sparse linear	79.64±1.69	1.67M	19±3k
Sparse nonlinear	78.49±0.78	2.53M	18±3k

Table 5.8: Split CIFAR100 results with ResNet-32 target model. The multitasking hypernetworks obtain similar performances lower than the experts.

Model	Accuracy	Parameters
Chunked	86.18	67.3M
Experts	85.53	118M
Sparse linear	82.38	44.6M
Sparse nonlinear	85.37	66.9M

Table 5.9: Split CIFAR100 results with ResNet18 target model. The multitasking hypernetworks obtain similar performances lower than the experts.

5.3.3 Split CIFAR 100

Resnet32 target model

For this experiment, we compared a chunked hypernet, a sparse one with a task-specific experts model. A batch size of 50 was used (gradients accumulated on 3 batches) with Adam optimizer, a starting learning rate of 1e-3, a learning rate reduction after 5 non improving epochs, early stopping after 10 non improving epochs. The latent size is 64 and there were 22 chunks for the chunk hypernet (chunk size of 19383). The target model is a ResNet32 (n=5) as described in He et al. [2015c] Section 4 without batch normalizations of 426426 parameters.

The chunked one has the best accuracy while converging in fewer (Table 5.8). The sparse linear hypernetwork supersedes the non-linear one while having fewer parameters. The non-linear sparse hypernetwork was the only model below the expert models in terms of accuracy.

Resnet18 target model

It is the same experiment as the previous one, except that we use a ResNet18 as target network, which is much larger than the previous one with 11M paramaters. In this experiment, a larger latent size of 512 is used to deal with the larger target model and chunked hypernetworks have 176 chunks.

The results are shown in Table 5.9. The chunked hypernetwork outperforms all other models in this multitasking problem. It is also slightly better than the independent experts. The non-linear sparse model performs equivalently as the experts. The linear model, however, falls behind all other models, contrary to the previous experiment where it outperformed the non-linear one and the experts.

5.4 Sparse target network

In the experiments in Section 5.3, we used hypernetworks to produce the parameters of ResNet models, which are typically large overparameterized models. The fact that the target network is overparameterized may make the task easier for hypernet. Some parameters may be redundant or useless.

In this section, we try to evaluate the performance of the hypernetwork in the case when the target network is not overparameterized. We aim to see whether hypernetworks have similar performance or if they degrade more quickly.

To do so, we produce the parameters of a sparse ResNet for increasing levels of sparsity. The sparsity distribution is a uniform distribution. A random binary mask defined before training with the desired sparsity probability is multiplied with the vector of parameters produced by the hypernetworks. Note that the size of the output of the hypernetwork is fixed and does not diminish with the increased target sparsity, which introduces some parameter inefficiency for the sparse hypernetworks.

Different levels of sparsity were tested, from no sparsity to 95% target network sparsity. The target is a ResNet-32. A latent size of 64 was used for z. The chunked hypernetworks have 23 chunks.

The results are shown in Figure 5.5. The chunked hypernetwork performed better than every other method except for the 95% sparsity level for which the non-linear sparse hypernetwork performed slightly better. The experts models generally had the lowest across the different sparsities. Both sparse hypernetworks performed in between experts and chunked hypernetworks. The non-linear one performed better than the linear one for the no sparsity setting and for the highest level of sparsity. It was the contrary for mid-level of sparsity. One of the key insights of this experiment is that the chunked hypernetwork did not suffer more from a higher level of target parameter sparsity than the sparse hypernetworks.

5.5 Experiments in the z space

Complete task-dependent hypernetworks can be interpreted as meta-models. For a given hypernetwork, z represents a model index. When a hypernetwork is trained on a set of tasks, the corresponding set of z-vectors (or distributions) is learned together. In this section, we will explore the z space and go outside of those learned vectors z.

We will try to answer a few questions. Can we infer a new z for a completely new task? for tasks similar to the ones on which the hypernetwork has been trained? What is the performance of the models obtained by linear interpolation of the z values? Can we compose z-values to get a composition of tasks?

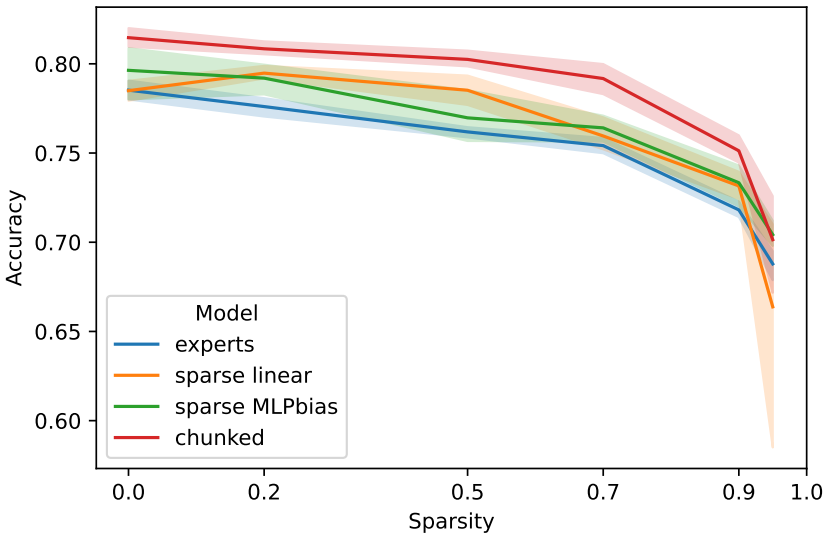


Figure 5.5: Accuracy for increasing level of sparsity of target networks (from 0% to 95% sparsity)

5.5.1 Models and training used

These experiments are done with tasks of split CIFAR100/10 or split CIFAR10/5. The target network is a ResNet-32. The training is similar to the previous experiments (see Appendix B for default settings).

5.5.2 Task inference

In this section, we try to use hypernetwork in a transfer learning fashion. We first pretrain the hypernetwork over a set of tasks. Then we freeze the hypernetwork weights and try to infer a z for a new task. By doing so, we hope that a pretrained hypernetwork over certain tasks can generalize to other tasks. With this experiment, we also want to test the performance of the hypernetwork as a meta-model, which has the capacity to produce new models.

The first experiment involves inferring a new z for completely different tasks than the ones on which the multitasking hypernetwork has been trained on. The second set of experiments concerns inferring new tasks that are similar to the pretraining task. In this case, we expect the inferred models to have relatively good performance.

Model	0	1	2	3	4	5	6	7	8	9
chunked	81.72	77.04	80.42	78.18	80.90	81.16	81.16	77.76	82.58	13.18 ± 3.22
sparse MLPbias	80.94	76.12	80.14	78.40	80.74	79.50	79.86	77.62	81.36	9.30 ± 2.63
sparse linear	80.24	73.14	79.26	77.00	79.00	79.26	79.64	76.94	80.38	**28.90 ± 3.85**

Table 5.10: Hypernetwork accuracy over the first 9 tasks and the accuracy on the last task with an inferred z on the pretained hypernetwork.

Completely different tasks

In the first experiment, we pretrain the hypernetwork on the first 9 tasks of split CIFAR100/10, then we freeze the hypernet and train a new z on the last task by gradient descent. This new task does not share any images with the previous 9 tasks and is therefore completely new and different. The new z is initialized at $z_9 = \frac{1}{9}\sum_{t\in 0,..,8} z_t$.

The results are shown in Table 5.10. The sparse linear hypermodel actually largely outperforms the other models by reaching 29% accuracy on the new task despite being less good on the training tasks. The chunked hypernetwork remains at 13%, which is slightly above 10% (for a task of 10 classes). The non-linear hypernet remains clueless with 9.3% accuracy. The results obtained by the linear hypernetwork are promising. They show that there might be some meta-generalization property of hypernetworks even though the best results obtained are far below the ones obtained on the 9 first tasks and the average accuracy over split CIFAR100/10 (Table 5.8). The better results obtained by the linear hypernetwork could be due to the fact that it has limited representation capabilities compared to the non-linear alternatives. This could help the hypermodels avoid meta-overfitting on the pretraining tasks and improve generalization to new tasks.

Similar tasks

In the previous experiment, we tested the performance of the hypernetworks for learning completely new tasks. In this experiment, we evaluate the transfer learning performance of the hypernetwork to learn similar tasks to the ones on which it has been pretrained.

If a model has been pretrained on two tasks of split MNIST for differentiating 0vs1 and 2vs3. We may expect that the learned hypernetwork can represent models for the similar tasks 0vs2 and 1vs3. We would like to see whether or not the hypernetwork can learn some reusable knowledge.

For this experiment, we used the split CIFAR10/5 problem (5 binary tasks). We first trained a hypernetwork (producing ResNet32 model) on the 2 first tasks 0:"airplane vs automobile" and 1:"bird vs cat". Then, after freezing the hypernetwork, we try to learn 4 new z for 4 similar tasks: 2:"airplane vs bird", 3:"automobile vs cat", 4:"airplane vs cat" and 5:"bird vs automobile". The tested tasks are summarized in Table 5.11.

The 3 models obtain very similar performances when trained on the first

Id	Binary Task	class id vs class id
0	airplane vs automobile	0vs1
1	bird vs cat	2vs3
2	airplane vs bird	0vs2
3	automobile vs cat	1vs3
4	airplane vs cat	0vs3
5	bird vs automobile	2vs1

Table 5.11: Tasks id and corresponding tasks for task inference experiment with similar tasks

Model	0vs1	2vs3	0vs2	1vs3	0vs3	2vs1
Chunked	98.34	**89.99**	77.07	80.89	**88.74**	**97.09**
Sparse MLPbias	98.42	89.13	63.66	74.95	88.37	96.49
Sparse linear	**98.51**	89.74	**79.99**	**84.80**	86.88	90.59

Table 5.12: Accuracy of hypernetwork over 2 pretrained task 0 and 1. Accuracy for infered z on 4 new tasks (2, 3, 4, 5).

2 tasks (Table 5.12). However, the results differ for the inference of z for new tasks. The sparse linear model obtains the best accuracy on the 2 following tasks (0vs2 and 1vs3) while the chunked hypernetwork obtains the best accuracy on the 2 last ones (0vs3 and 2vs1). The last two accuracies are quite good, 88.7% on the 0vs3 task and 97% on 2vs1.

The two last inferred tasks 0vs3 and 2vs1 have their classes in the same order as in the pretraining tasks. For task 0vs3, 0 is in the first position as in pretraining task 0vs1 and 3 is in the second position as in task pretraining 2vs3. For the task 2vs1, 2 is in the first position as in 2vs3 and 1 is in the second position as in 0vs1. It is not the case for the first two inferred tasks 0vs2 and 1vs3 which only have 1 class that is in the same order as in one of the pretraining tasks. Looking at the results Table 5.12, the linear model actually performs better than the others on tasks that have only one class that is in the same order and performs worse than the other when the task has all its classes in the same order as in the pretraining tasks. The nonlinear models seem thus to be more sensitive to a permutation of the class ID of their pretraining tasks. This may be a hint that linear hypernetworks better meta-generalize to new tasks compared to nonlinear ones that would meta-overfit their pretraining tasks.

Model	0vs1	2vs3	0vs2	1vs3	0vs3	2vs1
Chunked	**98.56**	**91.09**	**95.13**	**98.91**	**97.20**	99.18
Sparse MLPbias	98.43	90.30	94.50	98.79	96.81	**99.20**
Sparse linear	98.50	90.51	94.24	98.67	96.66	99.14

Table 5.13: Accuracy of hypernetwork over when trained directly on all 6 tasks.

Model	Accuracy (split CIFAR10/5)	Accuracy (altered split CIFAR10/5)
chunked	95.99±0.46	**96.38±0.28**
sparse MLPbias	**96.05±0.19**	96.30±0.11
sparse linear	95.70±0.27	96.06±0.35

Table 5.14: Mean accuracy of the multitasking hypernetwork on split CIFAR10/5 and altered split CIFAR10/5 with normal training.

Model	Accuracy
chunked	**89.67±1.29**
sparse MLPbias	87.92±0.82
sparse linear	88.38±0.93

Table 5.15: Mean accuracy of the multitasking hypernetwork on altered split CIFAR10/5 with normal training on classical split CIFAR10/5.

Altered tasks

In this experiment, we try to evaluate the transfer learning capabilities of a pretrained hypernetwork on altered tasks. To do so, we used the split CIFAR10/5 dataset and and altered version of this dataset. The altered version of split CIFAR10/5 contains the same tasks except that the images have been altered. The images are altered by changing the hue value with a random scaling factor drawn uniformly between 1.2 and 1.3. This basically corresponds to a change of color, which can make the sky pink instead of blue, the grass blue instead of green, etc.

First, we train the multitasking hypernetworks normally on both split CIFAR10/5 and the altered version. The results are shown in Table 5.14. The chunked and non-linear sparse hypernetworks achieve the best results. Interestingly, slightly better results are obtained on the altered version.

Before retraining the z values on the altered tasks, we first report the accuracy on the altered dataset of the models trained on the normal dataset. The results are shown in Table 5.15. As expected, they are lower than the results on the unaltered dataset but only by a few percents. The chunked hypernetwork maintains the best accuracy on this altered version.

Here, we retrain the z values on the altered dataset after pretraining the hypernetwork on the unaltered version. The accuracy obtained is shown in Table 5.16. As expected, the accuracy improves compared to the initial model (Table 5.15). However, the models do not achieve the best accuracy that can be obtained by training the entire hypernetwork on the altered data (Table 5.14). In this case, the linear hypernetwork had the best improvement to the point the it is the best model with 91.41% accuracy, although it had the worst accuracy on the initial unaltered split CIFAR10/5.

Model	Accuracy
chunked	91.20±1.38
sparse MLPbias	88.97±0.65
sparse linear	**91.41±0.30**

Table 5.16: Mean accuracy of the multitasking hypernetwork on altered split CIFAR10/5 with normal pretraining on classical split CIFAR10/5 followed by learning only new z values on the altered split CIFAR10/5. It can be compared with the accuracy obtained without retraining z values in Table 5.15

.

5.5.3 Model interpolation

In order to further assess the quality of the latent space, we interpolate models in the z space. We first train the hypernetwork on a set of tasks $\{t_k|k \in \{0, 1, ..., T-1\}\}$ and the corresponding embeddings $\{z_k|k \in \{0, 1, ..., T-1\}\}$. Then we evaluate the performance of the models generated by linear interpolation of 2 z, in particular z_0 and z_1 corresponding to the first and second tasks.

In a first experiment, we train the hypernetwork on only 2 tasks (the 2 first tasks of split CIFAR10/5) with their corresponding z_0 and z_1 task embeddings. The target network is a ResNet-32. The accuracy of linearly interpolated models between z_0 and z_1 is shown in Figure 5.6 and in Figure 5.7 for the accuracy on the second task.

In Figure 5.6, the best accuracy for the first task is obtained by the model produced by z_0, as expected. The models produced at z_1, however, obtain an accuracy better than 50% on the first task. This means that both tasks share some information as the models trained at z_1 are optimized for the second task, not the first. Generally, the interpolated models from the non-linear models performed way better than when it came from the linear hypermodels. Interestingly, even though the accuracy of the models produced by the midpoint between z_0 and z_1 is close to 50% for the linear hypermodel, the accuracy is over 50% near z_1, which means that there is also a share of knoledge between tasks for the linear models as for the non-linear ones.

The results are similar in the other direction (Figure 5.7) for the accuracy on the second task.

The two non-linear hypernetworks obtain good accuracy for linearly interpolated models. It is interesting to note that the model interpolated at the midpoint between z_0 and z_1 ($\frac{z_0+z_1}{2}$), provides a good trade-off between the model from z_0 and z_1. It has a relatively good accuracy on both tasks.

In the previous experiment, we only trained the multitasking hypernetwork on 2 tasks. What would happen if we increased the number of tasks? In this experiment, we train the model on 5 tasks with the 5 corresponding task embeddings (the 5 tasks of split CIFAR10/5). As in the previous experiment, we report the accuracy of the interpolated models between the task embeddings of the first and second tasks z_0 and z_1.

The results are shown in Figures 5.8 and 5.9. The results for the models

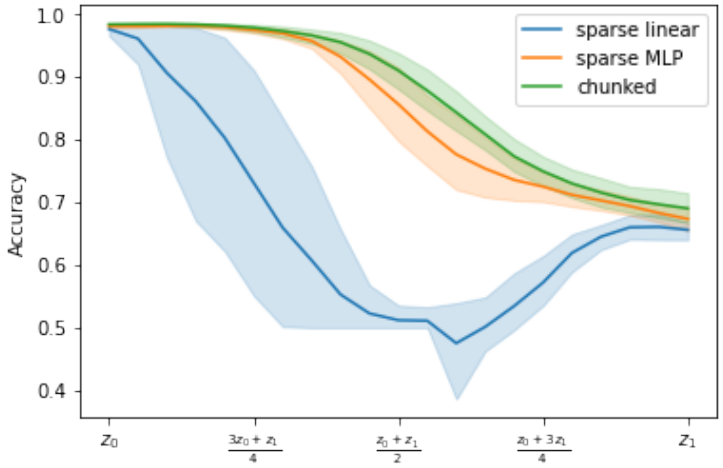


Figure 5.6: Accuracy of linearly interpolated model between z_0 and z_1 on the first task

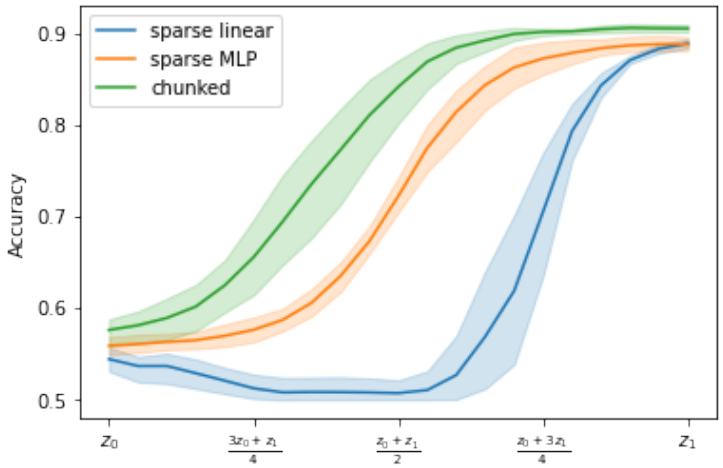


Figure 5.7: Accuracy of linearly interpolated model between z_0 and z_1 on the second task

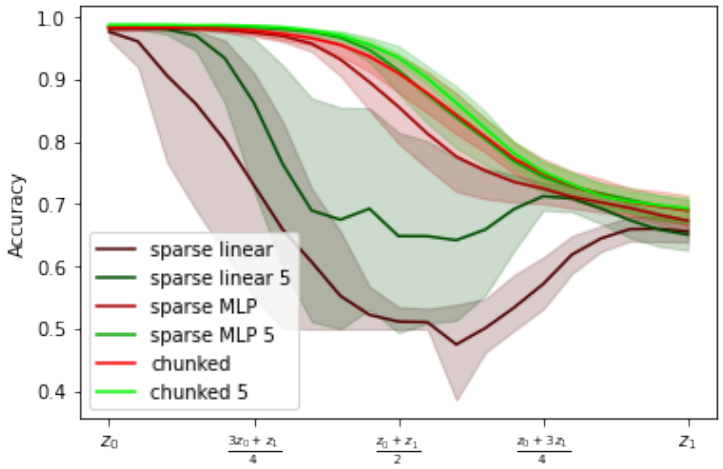


Figure 5.8: Accuracy of linearly interpolated model between z_0 and z_1 on the first task when trained on 5 tasks.

trained on 2 tasks in the previous case are shown in red shade for comparison. The results of the new models trained on five tasks are shown in green shade.

When closely looking at Figures 5.8 and 5.9, one can see that the interpolated models in green shade (which come from models trained on 5 tasks) generally outperform the corresponding counterparts (trained on 2 tasks) except for the chunked hypernetwork on the second task (Figure 5.9). This may indicate that multitasking hypermodels trained on more tasks can share more information between all these tasks and thus interpolated models perform better on all tasks. However, this multitasking model was trained on only five tasks and the results could be different with many more tasks.

5.5.4 Tasks composition

In the previous Sections 5.5.2 and 5.5.3, we learned that:

1. Pretrained multitasking hypernetwork can infer new target models for new tasks, especially if the new tasks share similarity with the pretraining tasks.

2. Linear interpolation of task embeddings can produce models that are good on all tasks. More specifically, the midpoint between 2 z values can lead to a target model that is good for both tasks on which the 2 z were trained.

These 2 observations motivate the present experiment. We define an average task which is a composition of all tasks. All classification tasks have disjoint domains. The average task is defined as follows: for a given image that may come from any of the tasks, the model should predict its corresponding label in

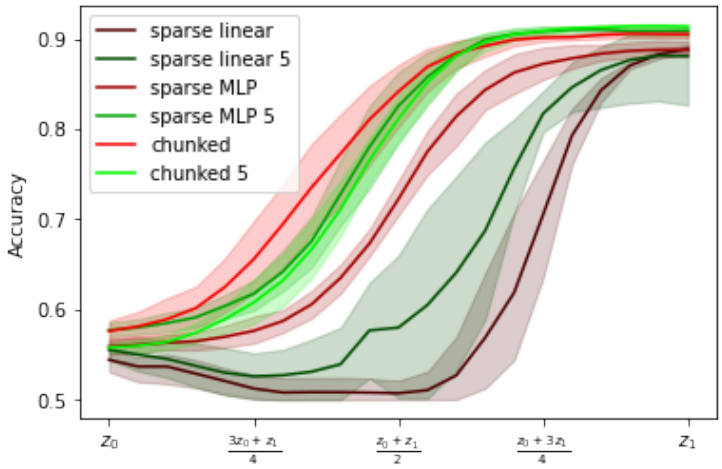


Figure 5.9: Accuracy of linearly interpolated model between z_0 and z_1 on the second task when trained on 5 binary tasks.

the corresponding task. We have seen in Section 5.5.3 that linear interpolation of models can perform well on all tasks, which means that the model can predict the correct label of images for each task and thus that this interpolated model could perform well on the average task. Consequently, we define the average model as the model produced by the average value of all trained z values and we hope that this average model can perform well on the average task.

In this experiment, we would like to evaluate the performance of the average model for the average task. There are actually several ways one could define the average model. In the first case, we simply choose the first previously proposed definition of average model (normal training). In the second case, we train a specific z for the average task (infer z for the average task).

Normal training

A hypernetwork $\mathcal{H}$ is trained on a set of T tasks together with a set of T task embeddings $\{z_t | t \in 0, 1, 2, ..., T-1\}$. In this case, it is trained on Split CIFAR100/10 (10 tasks of 10 classes).

We define the average model $\mathcal{M}_{avg} := \mathcal{H}(\bar{z}) = \mathcal{H}(\frac{1}{T}\sum_t z_t)$. It is the model obtained by giving the average of the z values to the trained hypernetwork.

Once the model has been trained on T tasks, we evaluate the accuracy of the average model $\mathcal{H}(\bar{z})$ (Table 5.17). We also report the evolution of the accuracy of the average model during training over a validation set (Figure 5.10).

The chunked hypernetwork and the sparse non-linear one, though not very good, reached up to 20% accuracy on the average task, which is significantly higher than the expected 10% accuracy of a random model for this classification task of 10 classes. The sparse linear, however, reached a 10% accuracy. This

Model	Mean z Accuracy	Mean Accuracy
Chunked	18.73±1.39	81.80±0.65
Sparse MLPbias	**19.70±0.82**	79.95±1.02
Sparse linear	9.67±0.89	78.42±1.11

Table 5.17: Accuracy of the average model $\mathcal{H}(\bar{z})$ on the average task and the base average accuracy over the different tasks. The chunked model and the sparse non linear one obtained accuracy much higher than the expected 10% accuracy of a random model for a classification task with 10 classes.

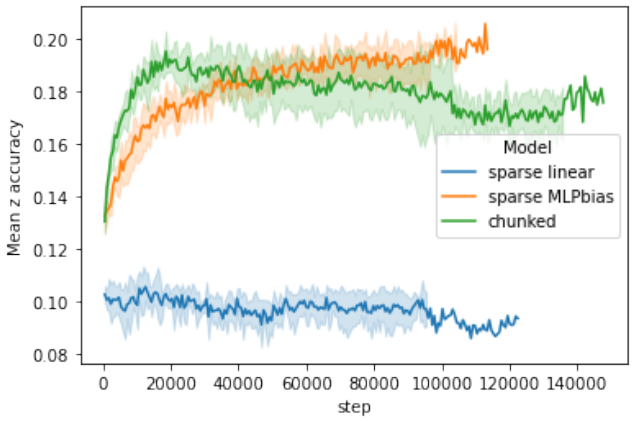


Figure 5.10: Evolution of the validation accuracy of the average model over the average task

Model \Task	0	1	2	3	4	5	6	7	8	9
Chunked	25.56	15.08	16.04	17.20	26.62	23.42	14.20	21.82	15.96	11.38
Sparse MLPbias	20.32	17.78	20.40	19.58	20.16	20.82	16.26	20.58	23.58	17.54
Sparse linear	10.78	11.64	7.88	9.60	8.88	9.86	8.90	11.08	10.44	7.66

Table 5.18: Accuracies of the average model on each task

agrees with the previous results from Section 5.5.3 where interpolated models from non-linear hypernetworks performed better than those from linear hypernetworks. This means that for the first 2 models, the average model is indeed (slightly) linked to the average task, while this is not the case for the last one. For the sparse non-linear model, the increase was relatively monotonic and correlated with the increase of the accuracy over each task. For the chunked one, the mean model peaked after ~ 15000 iterations and then slightly decreased, its accuracy is therefore less correlated with the other accuracies than the sparse non-linear one. We can also see that there is a little more variation for the chunked hypernetwork than for the sparse hypernetwork in the accuracy of the average model between every task (Table 5.18).

Infer z for average task

In the previous experiment, we tested the model defined by the average z on the average task. This choice was slightly arbitrary and there could be a better z for the average task for a given hypernetwork.

For this experiment, we still train the hypernetwork on T tasks as usual. Once it is trained, however, we freeze the hypernetwork parameters and train a new z on the average task. We experimented with two ways to train this new z. The first option is to simply learn a new vector z by gradient descent. The second option that we tried is to constrain the new z to be a convex combination of all the other z_t, i.e. $z_{avg} = \sum_t \omega_t z_t \;\; s.t. \sum_t \omega_t = 1$. In order to train this z, we learn the corresponding weights ω_t. Instead, we chose to learn a vector of logit weights α with $z_{avg} = Z\text{softmax}(\alpha)$ where Z is the matrix where all z_t are stacked (column-wise). All components of α are initialized with $\frac{1}{T}$, which yields the previous average model.

The results obtained are quite disappointing (Table 5.19). Learning a free z over the average task yielded an accuracy slightly higher than that of the previous average model. The best improvement was obtained with the sparse linear model. For the constrained z, the results are actually even worse than the average model. We hypothesize that this poor performance is due to the fact that the model was trained with accumulated batches of only 3 tasks and not all tasks, which could make the gradient descent too stochastic, especially when training z as a convex combination of the other z values.

In order to know how difficult this average task is, we train a hypernetwork together with a single z value on this task to see how it performs.

The results are shown in Table 5.20. The best results obtained, 47.40%, are

Model	Free z	Contrained z
Chunked	**21.18±0.76**	11.71±1.37
Sparse MLPbias	20.52±0.58	**15.03±2.30**
Sparse linear	15.67±1.17	10.02±0.28

Table 5.19: Accuracy over the average task for trained z

Model	Accuracy
Chunked	45.31±0.13
Sparse MLPbias	45.33±0.14
Sparse linear	47.40±0.99

Table 5.20: Accuracy over the average task for a single z trained together with the hypernetwork on the average task.

significantly above the one obtained by only retraining the z on a pretrained hypernetwork.

5.6 Discussion

The experiments presented in this chapter aimed to analyze different aspects of sparse hypernetworks. Here, we draw a conclusion on these results.

Hyperparameters. The sparse hypernetwork hyperparameter experiment (Section 5.2) tested different combinations of sparse hypernetwork hyperparameters proposed in Chapter 4.

This experiment, as well as the following, showed that linear hypernwetworks reached a good accuracy very close to that of non-linear hypernet, sometimes even better. Additionally, purely linear sparse hypernetworks have up to 33% less parameters than nonlinear sparse hypernets with bias term (Table 4.1).

It was also shown that the proposed method to increase connectivity at lower-dimensional layers of the sparse hypernets with linearly and exponentially decreasing connectivity patterns improves the accuracy while reducing the total number of parameters compared to the naive constant connectivity pattern.

Additionally, the proposed connectivity distribution that increases locality in sparse hypernetworks compared to the naive uniform distribution also showed improved performance. The mixed and normal distributions had better results than the randperm and uniform distributions. This may indicate that in the tested target networks, parameters that are close together may benefit from more information sharing than parameters that are further away. However, the gap between local and non-local distribution was not that large.

Comparison. The following experiments compared the sparse hypernetwork with the current solution of chunked hypernetworks and expert models. We

showed that sparse hypernetwork can match the accuracy of chunked hypernetwork and sometimes surpass it. However, the chunking method had a better accuracy on the much more complex task of split CIFAR100/10. The expert models were often surpassed by the multitasking hypernetworks which indicates some sharing between the different tasks.

Sparse targets. The target network sparsification experiment (Section 5.4) compared the behavior of sparse hypernetworks with sparse target networks. One could have expected the chunked hypernetwork to suffer more from the sparsification of the target network because of its architecture, which reuses the same model at different places. However, it was not observed, and all models seem to suffer similarly from the sparsification of the target network. However, in this experiment, the sparse hypernetwork was not adapted to deal with the sparse target network architecture. This means that the sparse hypernets still produce all parameters, even those that are set to zero, which leads to some inefficiency. It would be interesting to try to adapt the sparse hypernetworks architecture in order to have efficient sparse-to-sparse hypernetworks.

Generalization. Finally, the last set of experiments compared the generalization property of linear, non-linear sparse hypernets and chunked hypernets. During these experiments, it was shown that the nonlinear hypernets can match the chunked hypernetwork in terms of the performance of the models resulting from the interpolation of the z values. This means that their latent space can produce other target networks that are also good at the tasks on which they were trained.

The inference of new tasks with pretrained hypernets showed interesting results. Drawing an analogy with classical machine learning, where reducing the hypothesis space of the model can help avoid overfitting the data, reducing the multitasking hypernetwork representation power to a linear function seems to improve its generalization capability to new tasks. On all task inference problems tested, the linear hypernetwork was one of the best models, even though it had the lowest accuracy on the pretraining problem.

Conclusion. The results obtained by the sparse hypernetworks are promising. While they seem to slightly underperform compared to the chunked hypernetwork on multitasking problems, it showed interesting results in generalizing their knowledge to new tasks. The linear sparse hypernetwork manages to outperform the non-linear one and the chunked hypernetwork, especially for learning new tasks that are radically different from the pretraining task.

Chapter 6

Conclusion and future work

Conclusions. In this work, we started by reviewing the hypernetwork literature and we presented a typology of hypernetworks which clarifies the different methods and architecture of hypernetworks.

Then, we proposed a new architecture for building complete hypernetworks that are scalable to large target networks. This architecture takes the form of a sparse expanding MLP. Different improvements were proposed with different connectivity types and connectivity distributions. In particular, the normally distributed connections allow to change the locality of the connections. The linearly decreasing and exponentially decreasing connectivity patterns efficiently reduce the number of connections compared to the constant connectivity pattern.

We tested different types of sparse hypernetworks and showed that the improvements defined in Section 4.2 were effective in improving the accuracy obtained while keeping a lower number of parameters compared to the constant and uniformly distributed connectivity pattern. The mixed and normal distribution patterns, which increase the locality of connections compared to the rand-perm and uniform distribution, helped the model perform better. The linearly decreasing and exponentially decreasing connectivity types improve the performance by increasing the connectivity in the first layers of the sparse hypernetwork.

The experiments show that the sparse hypernetworks can match the average accuracy obtained by the current method of chunking on multitasking classification problems. However, chunked hypernetworks were slightly better on more complex problems like split CIFAR100/10. The experiments also interestingly showed that linear sparse hypernetworks reached performance close to the non-linear ones.

Finally, the final experiments on task inference provide interesting results. They showed that the sparse linear hypernetworks, which are much simpler models than the non-linear ones and the chunked ones, obtained lower accuracy when pretrained on a set of tasks but a higher accuracy than the other models when inferring new target models for new tasks, especially in the case of new

tasks that are completely different from the pretraining tasks.

Future work. The sparse hypernetwork model shows promising results. However, they could be improved in many ways.

Experiment 5.4, on sparse target networks, does not adapt sparse hypernetwork to deal with the sparsity of the target network, which means that some parameters are uselessly predicted and set to zero afterward. It would be interesting to change the structure of the sparse hypernetworks to take into account the sparsity of the target network, which would yield sparse-to-sparse hypernetworks (sparse hypernetworks producing sparse target networks). This would result in a better parameter efficiency of the sparse hypernetwork and could potentially increase the performance.

We provide different ways of connecting the neurons of the sparse hypernetworks. However, these choices are done before training. Sparse evolutionary training (Mocanu et al. [2017]) could be applied to further improve the results of these models. This method changes the connections of a sparse neural network during training.

Our study analyzes sparse hypernetworks on multitasking problems. It would be interesting to test these methods on input-based and noise-based hypernetworks and compare them with chunked hypernetworks. Sparse hypernetworks could provide more diversity of parameters than the chunked ones with noise-based hypernetworks.

We could also dive deeper into multitasking problems to try to build interpretable multitasking models. The last experiment in 5.5.4, tries to look at the performance of an average model on an average task. We could extend this concept and try to build hypernetworks that can compose the values of their latent space z in order to perform on the composition of the corresponding tasks. There are still some questions to answer, such as what kind of task compositions are possible with hypernetwork and how to actually compose tasks and models ?

The sparse linear hypernetwork showed promising results for task inference, but is still far from perfectly solving new tasks. Future studies could aim to improve hypernetworks capabilities, maybe by trying to regularize hypernetworks in order to avoid meta-overfitting on the task on which it is trained.

Finally, the proposed architecture, sparse expanding networks, has been applied to hypernetworks because they require a large output space. However, other problems require a large output space without having a particular structure unlike images. Extreme multi-label learning (XML) is a problem which requires a potentially large output space. The sparse expanding networks, which obtained good results with hypernetwork, could be applied in this setting in future research.

Appendix A

Proofs

A.1 Proof of 4.13

.

Proof of

$$\frac{n}{2^L}\left[(L+1)\sum_{l=1}^{L}2^l-\sum_{l=1}^{L}l2^l\right]\leq 4n$$

From geometric series theory, we have

$$\sum_{l=1}^{L}2^l=2^{L+1}-2$$

.

For $s=\sum_{l=1}^{L}l2^l$, we proceed as follows.

s	$=2$	$+2\times 2^2$	$+3\times 2^3$	$+\ ...$	$+L\times 2^L$	
2s	$=$	$+1\times 2^2$	$+2\times 2^3$	$+\ ...$	$+(L-1)\times 2^L$	$+L\times 2^{L+1}$
s-2s	$=2$	$+2^2$	$+2^3$	$+\ ...$	$+2^L$	$-L\times 2^{L+1}$

Therefore:

$$\begin{aligned}-s&=\sum_{l=1}^{L}2^l-L\times 2^{L+1}\\ -s&=2^{L+1}-2-L\times 2^{L+1}\\ s&=-2^{L+1}+2+L\times 2^{L+1}\\ s&=(L-1)\times 2^{L+1}+2\end{aligned}$$

Finally:

$$\begin{aligned}
\frac{n}{2^L}\left[(L+1)\sum_{l=1}^{L}2^l-\sum_{l=1}^{L}l2^l\right] &= \frac{n}{2^L}\left[(L+1)(2^{L+1}-2)-((L-1)2^{L+1}+2)\right] \\
&= \frac{n}{2^L}\left[(L+1)2^{L+1}-2L-2-(L-1)2^{L+1}-2\right] \\
&= \frac{n}{2^L}\left[(2)2^{L+1}-2L-4\right] \\
&\leq 4n
\end{aligned}$$

A.2 Explaination on uniform connectivity

With a uniform distribution of connection as described in Section 4.2.3, there will be approximatly $\frac{1}{e^{bc_l}}h_{l_1}$ input neurons that are not connected to any output neurons for large hidden layers. For a single draw of input neuron, there is a chance of $1-\frac{1}{h_{l-1}}$ that an input neuron is not chosen. For n independent draws, there is a $(1-\frac{1}{h_{l-1}})^n$ chance that an input neurons is not chosen. In the case of a sparse hypernetwork, we have $h_{l-1}=\frac{h_l}{b}$ (with b the expansion factor) and we make $c_l h_l$ draws. The chance that an input is not chosen after these $c_l h_l$ draws is $(1-\frac{b}{h_l})^{c_l h_l}$. However, we have $\lim_{n\to\infty}(1+\frac{x}{n})^n = e^x$. Therefore we have $\lim_{h_l\to\infty}(1-\frac{b}{h_l})^{c_l h_l} = e^{-bc_l}$, which is the asymptotic probability that an input neuron is not chosen for large hidden layers.

Appendix B

Experimental settings and further details

The main training hyperparameters are presented here. The experiments concern multitasking problems. In order to train multitasking hypernetworks conditioned on a task, we use batches composed of a single task but we averaged the gradients of 3 consecutive batches to estimate the gradient over multiple tasks. The tasks are sampled in a random order.

During training, we monitor the performance of the model on a validation set. We apply an early stopping strategy on the validation accuracy of the model, with a patience of 20 epochs by default. We also progressively reduce the learning rate by monitoring the validation accuracy. The learning rate is reduced by a factor of 2 after a series of non-improving epochs. We use a patience that is half of the early stopping patience. The training hyperparameters are shown in Table B.1, while the default hyperparameters for the sparse hypernetworks are shown in Table B.2.

Parameter	Value
learning rate	1e-3
batch size	25
number of batch accumulated	3
optimizer	Adam
gradient clipping value	1.0
learning rate reducing factor	0.5
validation accuracy monitor patience	20
learning rate reducing patience	validation accuracy monitor patience / 2
validation split size	10% of the initial traning set
number of trials (for averaging)	5

Table B.1: Default hyperparameters used during the experiments

Parameter	Value
expansion factor	2
non linearity	PReLU
constant connectivity	3
connectivity distribution	Gaussian
latent size (d)	64
σ	latent size / 4
connectivity type	linearly decreasing
input normalization	yes

Table B.2: Default hyperparameters for sparse hypernetwork used during the experiments

Models The chunked hypernetwork is implemented as an MLP with hidden layers $2d - 2d - \lceil \frac{n}{C} \rceil$, where d is the latent size of z, n is the number of parameters of the target network and C is the number of chunks. The input is the concatenation of the d-dimensional vector z with a d-dimensional chunk embedding vector (there are C of them), hence the $2d$ input size.

The expert models for a multitasking problem with t tasks simply consists in t identical models that are trained on each task. For comparison with hypernetworks, the model chosen for an expert is the target network of the hypernetworks.

Inference time of sparse hypernetworks Another measure of the performance of hypernetworks is the time required to forward them. Sparse hypernetworks use sparse representation of matrix parameters instead of the typical dense implementation, which may have an impact on performance. Here, we report the time to predict a target network on CPU (Intel core i5-1035G1) in Table B.3.

Model	Nbr. parameters	Output size	Time
Sparse Hypernetwork	4.13M	1048576≈1M	24 ± 3.12ms
Chunked Hypernetwork	4.13M	1048576≈1M	7.34 ± 0.4ms
Sparse Hypernetwork	66.2M	16777216≈16.8M	400 ± 6ms
Chunked Hypernetwork	65.6M	16777216≈16.8M	115 ± 0.4ms

Table B.3: Time to predict a target network of different sizes (mean±std of 5 runs of 10 loops). The chunked hypernetwork has 33 chunks and the dimension of the input z is 64. The chunked hypernetwok is more than 3 time faster than the sparse one.

Bibliography

J. Ba, K. Swersky, S. Fidler, and R. Salakhutdinov. Predicting Deep Zero-Shot Convolutional Neural Networks using Textual Descriptions. *arXiv:1506.00511 [cs]*, Sept. 2015. URL `http://arxiv.org/abs/1506.00511`. arXiv: 1506.00511.

P. Bachman, R. Islam, A. Sordoni, and Z. Ahmed. VFunc: a Deep Generative Model for Functions. *arXiv:1807.04106 [cs, stat]*, July 2018. URL `http://arxiv.org/abs/1807.04106`. arXiv: 1807.04106.

L. Bertinetto, J. F. Henriques, J. Valmadre, P. H. S. Torr, and A. Vedaldi. Learning feed-forward one-shot learners. *arXiv:1606.05233 [cs]*, June 2016. URL `http://arxiv.org/abs/1606.05233`. arXiv: 1606.05233.

D. W. Blalock, J. J. G. Ortiz, J. Frankle, and J. V. Guttag. What is the state of neural network pruning? *CoRR*, abs/2003.03033, 2020. URL `https://arxiv.org/abs/2003.03033`.

A. Brock, T. Lim, J. M. Ritchie, and N. Weston. SMASH: One-Shot Model Architecture Search through HyperNetworks. Feb. 2018. URL `https://openreview.net/forum?id=rydeCEhs-`.

O. Chang, L. Flokas, and H. Lipson. Principled Weight Initialization for Hypernetworks. Sept. 2019. URL `https://openreview.net/forum?id=H1lma24tPB`.

W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing Neural Networks with the Hashing Trick. *arXiv:1504.04788 [cs]*, Apr. 2015. URL `http://arxiv.org/abs/1504.04788`. arXiv: 1504.04788.

L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

L. Deutsch, E. Nijkamp, and Y. Yang. A Generative Model for Sampling High-Performance and Diverse Weights for Neural Networks. *arXiv:1905.02898 [cs, stat]*, May 2019. URL `http://arxiv.org/abs/1905.02898`. arXiv: 1905.02898.

A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020. URL `https://arxiv.org/abs/2010.11929`.

V. Dwaracherla, X. Lu, M. Ibrahimi, I. Osband, Z. Wen, and B. Van Roy. Hypermodels for Exploration. *arXiv:2006.07464 [cs, math, stat]*, June 2020. URL `http://arxiv.org/abs/2006.07464`. arXiv: 2006.07464.

B. Ehret, C. Henning, M. R. Cervera, A. Meulemans, J. von Oswald, and B. F. Grewe. Continual Learning in Recurrent Neural Networks. *arXiv:2006.12109 [cs, stat]*, Mar. 2021. URL `http://arxiv.org/abs/2006.12109`. arXiv: 2006.12109.

J. A. Feldman. Dynamic connections in neural networks. *Biological Cybernetics*, 46(1):27–39, Dec. 1982. ISSN 1432-0770. doi: 10.1007/BF00335349. URL `https://doi.org/10.1007/BF00335349`.

C. Finn, P. Abbeel, and S. Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. Mar. 2017. URL `http://xxx.itp.ac.cn/abs/1703.03400`.

T. Galanti and L. Wolf. On the Modularity of Hypernetworks. *arXiv:2002.10006 [cs, stat]*, Nov. 2020. URL `http://arxiv.org/abs/2002.10006`. arXiv: 2002.10006.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

D. Ha, A. Dai, and Q. V. Le. HyperNetworks. *arXiv:1609.09106 [cs]*, Dec. 2016. URL `http://arxiv.org/abs/1609.09106`. arXiv: 1609.09106.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015a. URL `http://arxiv.org/abs/1502.01852`.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015b. URL `http://arxiv.org/abs/1502.01852`.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015c. URL `http://arxiv.org/abs/1512.03385`.

C. Henning, J. von Oswald, J. Sacramento, S. C. Surace, J.-P. Pfister, and B. F. Grewe. Approximating the Predictive Distribution via Adversarially-Trained Hypernetworks. In *Henning, Christian; von Oswald, Johannes; Sacramento, Joao; Surace, Simone Carlo; Pfister, Jean-Pascal; Grewe, Benjamin F (2018). Approximating the Predictive Distribution via Adversarially-Trained*

Hypernetworks. In: Bayesian Deep Learning Workshop, NeurIPS 2018, Montreal, 7 December 2018 - 7 December 2018., Montréal, Canada, Dec. 2018. Yarin. doi: 10.5167/uzh-168578. URL http://bayesiandeeplearning.org/2018/papers/121.pdf.

C. Henning, M. R. Cervera, F. D'Angelo, J. von Oswald, R. Traber, B. Ehret, S. Kobayashi, J. Sacramento, and B. F. Grewe. Posterior Meta-Replay for Continual Learning. *arXiv:2103.01133 [cs]*, June 2021. URL http://arxiv.org/abs/2103.01133. arXiv: 2103.01133.

G. E. Hinton. Using fast weights to deblur old memories. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, pages 177–186, 1987. URL https://ci.nii.ac.jp/naid/10008951938. Publisher: Erlbaum.

Y. Huang, K. Xie, H. Bharadhwaj, and F. Shkurti. Continual Model-Based Reinforcement Learning with Hypernetworks. *arXiv:2009.11997 [cs]*, Mar. 2021. URL http://arxiv.org/abs/2009.11997. arXiv: 2009.11997.

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive Mixtures of Local Experts. *Neural Computation*, 3(1):79–87, Mar. 1991. ISSN 0899-7667. doi: 10.1162/neco.1991.3.1.79. URL https://doi.org/10.1162/neco.1991.3.1.79.

M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu. Spatial transformer networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 2017–2025, Cambridge, MA, USA, Dec. 2015. MIT Press.

X. Jia, B. De Brabandere, T. Tuytelaars, and L. V. Gool. Dynamic Filter Networks. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://papers.nips.cc/paper/2016/hash/8bf1211fd4b7b94528899de0a43b9fb3-Abstract.html.

M. I. Jordan and R. A. Jacobs. Hierarchical Mixtures of Experts and the EM Algorithm. In M. Marinaro and P. G. Morasso, editors, *ICANN '94*, pages 479–486, London, 1994. Springer. ISBN 978-1-4471-2097-1. doi: 10.1007/978-1-4471-2097-1_113.

T. Karaletsos and T. D. Bui. Hierarchical Gaussian Process Priors for Bayesian Neural Network Weights. *arXiv:2002.04033 [cs, stat]*, Feb. 2020. URL http://arxiv.org/abs/2002.04033. arXiv: 2002.04033.

T. Karaletsos, P. Dayan, and Z. Ghahramani. Probabilistic Meta-Representations Of Neural Networks. *ArXiv*, 2018.

T. Karras, S. Laine, and T. Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. *arXiv:1812.04948 [cs, stat]*, Mar. 2019. URL http://arxiv.org/abs/1812.04948. arXiv: 1812.04948.

B. Klein, L. Wolf, and Y. Afek. A Dynamic Convolutional Layer for short rangeweather prediction. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4840–4848, June 2015. doi: 10.1109/CVPR.2015.7299117. ISSN: 1063-6919.

S. Klocek, L. Maziarka, M. Wolczyk, J. Tabor, J. Nowak, and M. Smieja. Hypernetwork functional image representation. *arXiv:1902.10404 [cs, stat]*, June 2019. doi: 10.1007/978-3-030-30493-5_48. URL `http://arxiv.org/abs/1902.10404`. arXiv: 1902.10404.

B. Knyazev, M. Drozdzal, G. W. Taylor, and A. Romero-Soriano. Parameter prediction for unseen deep architectures. *CoRR*, abs/2110.13100, 2021. URL `https://arxiv.org/abs/2110.13100`.

S. Kobayashi, J. von Oswald, and B. Grewe. On the reversed bias-variance tradeoff in deep ensembles. page 85, 2021. doi: 10.3929/ethz-b-000501624. URL `https://www.research-collection.ethz.ch/handle/20.500.11850/501624`. Accepted: 2021-08-23T04:47:20Z.

I. Kostiuk, P. Stachura, S. K. Tadeja, T. Trzciński, and P. Spurek. HyperColor: A HyperNetwork Approach for Synthesizing Auto-colored 3D Models for Game Scenes Population. *arXiv:2108.01411 [cs]*, Aug. 2021. URL `http://arxiv.org/abs/2108.01411`. arXiv: 2108.01411.

A. Krizhevsky. Learning multiple layers of features from tiny images. 2009.

D. Krueger, C.-W. Huang, R. Islam, R. Turner, A. Lacoste, and A. Courville. Bayesian Hypernetworks. *arXiv:1710.04759 [cs, stat]*, Apr. 2018. URL `http://arxiv.org/abs/1710.04759`. arXiv: 1710.04759.

A. Lamb, E. Saveliev, Y. Li, S. Tschiatschek, C. Longden, S. Woodhead, J. M. Hernández-Lobato, R. E. Turner, P. Cameron, and C. Zhang. Contextual HyperNetworks for Novel Feature Adaptation. Sept. 2020. URL `https://openreview.net/forum?id=CYHMIhbuLFl`.

Y. Li, S. Gu, K. Zhang, L. Van Gool, and R. Timofte. DHP: Differentiable Meta Pruning via HyperNetworks. *arXiv:2003.13683 [cs, eess]*, Aug. 2020. URL `http://arxiv.org/abs/2003.13683`. arXiv: 2003.13683.

X. Lin, Z. Yang, Q. Zhang, and S. Kwong. Controllable Pareto Multi-Task Learning. *arXiv:2010.06313 [cs, stat]*, Feb. 2021. URL `http://arxiv.org/abs/2010.06313`. arXiv: 2010.06313.

Lior Wolf. Hypernetworks: a versatile and powerful tool, Feb. 2020. URL `https://www.youtube.com/watch?v=KY9DoutzH6k`.

G. Littwin and L. Wolf. Deep Meta Functionals for Shape Representation. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1824–1833, Oct. 2019. doi: 10.1109/ICCV.2019.00191. ISSN: 2380-7504.

L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han. On the variance of the adaptive learning rate and beyond. *CoRR*, abs/1908.03265, 2019a. URL http://arxiv.org/abs/1908.03265.

Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, T. K.-T. Cheng, and J. Sun. MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning. *arXiv:1903.10258 [cs]*, Aug. 2019b. URL http://arxiv.org/abs/1903.10258. arXiv: 1903.10258.

R. K. Mahabadi, S. Ruder, M. Dehghani, and J. Henderson. Parameter-efficient Multi-task Fine-tuning for Transformers via Shared Hypernetworks. *arXiv:2106.04489 [cs]*, June 2021. URL http://arxiv.org/abs/2106.04489. arXiv: 2106.04489.

L. Mescheder, M. Oechsle, M. Niemeyer, S. Nowozin, and A. Geiger. Occupancy Networks: Learning 3D Reconstruction in Function Space. *arXiv:1812.03828 [cs]*, Apr. 2019. URL http://arxiv.org/abs/1812.03828. arXiv: 1812.03828.

D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta. Evolutionary training of sparse artificial neural networks: A network science perspective. *CoRR*, abs/1707.04780, 2017. URL http://arxiv.org/abs/1707.04780.

T. Munkhdalai and H. Yu. Meta Networks. *arXiv:1703.00837 [cs, stat]*, June 2017. URL http://arxiv.org/abs/1703.00837. arXiv: 1703.00837.

A. Navon, A. Shamsian, G. Chechik, and E. Fetaya. Learning the Pareto Front with Hypernetworks. *arXiv:2010.04104 [cs]*, Apr. 2021. URL http://arxiv.org/abs/2010.04104. arXiv: 2010.04104.

H. Noh, P. H. Seo, and B. Han. Image Question Answering using Convolutional Neural Network with Dynamic Parameter Prediction. *arXiv:1511.05756 [cs]*, Nov. 2015. URL http://arxiv.org/abs/1511.05756. arXiv: 1511.05756.

N. Pawlowski, A. Brock, M. C. H. Lee, M. Rajchl, and B. Glocker. Implicit Weight Uncertainty in Neural Networks. *arXiv:1711.01297 [cs, stat]*, May 2018. URL http://arxiv.org/abs/1711.01297. arXiv: 1711.01297.

E. Perez, F. Strub, H. de Vries, V. Dumoulin, and A. Courville. FiLM: Visual Reasoning with a General Conditioning Layer. *arXiv:1709.07871 [cs, stat]*, Dec. 2017. URL http://arxiv.org/abs/1709.07871. arXiv: 1709.07871.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv:1910.10683 [cs, stat]*, July 2020. URL http://arxiv.org/abs/1910.10683. arXiv: 1910.10683.

N. Ratzlaff and L. Fuxin. HyperGAN: A Generative Model for Diverse, Performant Neural Networks. *arXiv:1901.11058 [cs, stat]*, July 2020. URL http://arxiv.org/abs/1901.11058. arXiv: 1901.11058.

G. Riegler, S. Schulter, M. Ruther, and H. Bischof. Conditioned Regression Models for Non-blind Single Image Super-Resolution. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 522–530, Santiago, Chile, Dec. 2015. IEEE. ISBN 978-1-4673-8391-2. doi: 10.1109/ICCV.2015.67. URL http://ieeexplore.ieee.org/document/7410424/.

M. Ruchte and J. Grabocka. Efficient Multi-Objective Optimization for Deep Learning. *arXiv:2103.13392 [cs]*, Mar. 2021. URL http://arxiv.org/abs/2103.13392. arXiv: 2103.13392.

A. A. Rusu, D. Rao, J. Sygnowski, O. Vinyals, R. Pascanu, S. Osindero, and R. Hadsell. Meta-Learning with Latent Embedding Optimization. *arXiv:1807.05960 [cs, stat]*, Mar. 2019. URL http://arxiv.org/abs/1807.05960. arXiv: 1807.05960.

J. Schmidhuber. Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks. *Neural Computation*, 4(1):131–139, Jan. 1992. ISSN 0899-7667. doi: 10.1162/neco.1992.4.1.131. URL https://doi.org/10.1162/neco.1992.4.1.131.

B. Shakibi. *Predicting parameters in deep learning*. PhD thesis, University of British Columbia, 2014. URL https://open.library.ubc.ca/soa/cIRcle/collections/ubctheses/24/items/1.0165555.

A.-S. Sheikh, K. Rasul, A. Merentitis, and U. Bergmann. Stochastic Maximum Likelihood Optimization via Hypernetworks. *arXiv:1712.01141 [cs, stat]*, Jan. 2018. URL http://arxiv.org/abs/1712.01141. arXiv: 1712.01141.

P. Spurek, S. Winczowski, J. Tabor, M. Zamorski, M. Zieba, and T. Trzcinski. Hypernetwork approach to generating point clouds. *arXiv:2003.00802 [cs]*, Oct. 2020a. URL http://arxiv.org/abs/2003.00802. arXiv: 2003.00802.

P. Spurek, M. Zieba, J. Tabor, and T. Trzcinski. HyperFlow: Representing 3D Objects as Surfaces. *arXiv:2006.08710 [cs, eess]*, June 2020b. URL http://arxiv.org/abs/2006.08710. arXiv: 2006.08710.

P. Spurek, A. Kasymov, M. Mazur, D. Janik, S. Tadeja, L. Struski, J. Tabor, and T. Trzcinski. HyperPocket: Generative Point Cloud Completion. *arXiv:2102.05973 [cs]*, Feb. 2021. URL http://arxiv.org/abs/2102.05973. arXiv: 2102.05973.

R. Srivastava, P. Shyam, F. W. Mutz, W. Jaśkowski, and J. Schmidhuber. Training Agents using Upside-Down Reinforcement Learning. *ArXiv*, 2019.

Y. Tay, Z. Zhao, D. Bahri, D. Metzler, and D.-C. Juan. HyperGrid Transformers: Towards A Single Model for Multiple Tasks. Sept. 2020. URL https://openreview.net/forum?id=hiq1rHO8pNT.

K. Ukai, T. Matsubara, and K. Uehara. Bayesian estimation and model averaging of convolutional neural networks by hypernetwork. *Nonlinear Theory and Its Applications, IEICE*, 10(1):45–59, 2019. doi: 10.1587/nolta.10.45.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017a. URL http://arxiv.org/abs/1706.03762.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017b. URL http://arxiv.org/abs/1706.03762.

C. von der Malsburg. The Correlation Theory of Brain Function. In E. Domany, J. L. van Hemmen, and K. Schulten, editors, *Models of Neural Networks: Temporal Aspects of Coding and Information Processing in Biological Systems*, Physics of Neural Networks, pages 95–119. Springer, New York, NY, 1994. ISBN 978-1-4612-4320-5. doi: 10.1007/978-1-4612-4320-5_2. URL https://doi.org/10.1007/978-1-4612-4320-5_2.

J. von Oswald, C. Henning, J. Sacramento, and B. F. Grewe. Continual learning with hypernetworks. In *von Oswald, Johannes; Henning, Christian; Sacramento, João; Grewe, Benjamin F (2020). Continual learning with hypernetworks. In: ICLR 2020, Virtual Conference, 26 April 2020 - 1 May 2020.*, Virtual Conference, May 2020. ICLR. doi: 10.5167/uzh-200390. URL https://www.zora.uzh.ch/id/eprint/200390/.

J. von Oswald, S. Kobayashi, J. Sacramento, A. Meulemans, C. Henning, and B. F. Grewe. Neural networks with late-phase weights. *arXiv:2007.12927 [cs, stat]*, Apr. 2021. URL http://arxiv.org/abs/2007.12927. arXiv: 2007.12927.

F. Wenzel, J. Snoek, D. Tran, and R. Jenatton. Hyperparameter Ensembles for Robustness and Uncertainty Quantification. *arXiv:2006.13570 [cs, stat]*, Jan. 2021. URL http://arxiv.org/abs/2006.13570. arXiv: 2006.13570.

W. Xu, R. T. Q. Chen, X. Li, and D. Duvenaud. Infinitely Deep Bayesian Neural Networks with Stochastic Differential Equations. *arXiv:2102.06559 [cs, stat]*, Aug. 2021. URL http://arxiv.org/abs/2102.06559. arXiv: 2102.06559.

C. Zhang, M. Ren, and R. Urtasun. Graph HyperNetworks for Neural Architecture Search. Sept. 2018. URL https://openreview.net/forum?id=rkgW0oA9FX.

T. Zhang, B. Wu, X. Wang, J. Gonzalez, and K. Keutzer. Domain-Aware Dynamic Networks. *arXiv:1911.13237 [cs, stat]*, Nov. 2019. URL http://arxiv.org/abs/1911.13237. arXiv: 1911.13237.