Autogenerative Networks

Oscar Chang

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

Autogenerative Networks

Oscar Chang

Artificial intelligence powered by deep neural networks has seen tremendous improvements in the last decade, achieving superhuman performance on a diverse range of tasks. Many worry that it can one day develop the ability to recursively self-improve itself, leading to an intelligence explosion known as the Singularity. Autogenerative networks, or neural networks generating neural networks, is one major plausible pathway towards realizing this possibility. The object of this thesis is to study various challenges and applications of small-scale autogenerative networks in domains such as artificial life, reinforcement learning, neural network initialization and optimization, gradient-based meta-learning, and logical networks. Chapters 2 and 3 describe novel mechanisms for generating neural network weights and embeddings. Chapters 4 and 5 identify problems and propose solutions to fix optimization difficulties in differentiable mechanisms of neural network generation known as Hypernetworks. Chapters 6 and 7 study implicit models of network generation like backpropagating through gradient descent itself and integrating discrete solvers into continuous functions. Together, the chapters in this thesis contribute novel proposals for non-differentiable neural network generation mechanisms, significant improvements to existing differentiable network generation mechanisms, and an assimilation of different learning paradigms in autogenerative networks.

Table of Contents

Acknowledgements

The research endeavor undertaken in this PhD was made possible by the kind guidance from my advisor, Prof. Hod Lipson, and the support of my colleagues in the Creative Machines Lab. I also wish to thank the following people for their contribution to the papers that we submitted during the course of my PhD: Siyuan Chen, Robert Kwiatkowski, Yuling Yao, David Williams-King, Lampros Flokas, and Michael Spranger. The funding for my PhD was generously provided by the Fu Foundation Presidential Distinguished Fellowship, the US Defense Advanced Research Project Agency (DARPA) Lifelong Learning Machines Program (grant HR0011-18-2-0020), and the Columbia Computer Science department.

Motivation

The Threat of Artificial Intelligence The doomsday scenario where machines take over the world has been laid out repeatedly in science fiction. In the Matrix, our robot overlords have enslaved most humans in virtual reality to feed on us as a source of energy. In Ex Machina, a rogue AI charms her way into convincing a human test participant to let her out of the lab, leading her to kill all the researchers involved and escape out into the world. In Terminator, the AI defense system known as Skynet becomes self-aware, and initiates a nuclear holocaust.

In light of breakthroughs made in deep learning in recent years, Elon Musk has likened AI research to 'summoning the demon', warning that it is humankind's 'biggest existential threat' [1], while Stephen Hawking cautioned that it could 'spell the end of the human race' [2]. They were both signatories to a decidedly less apocalyptic but nevertheless ominous open letter [3], co-signed by many machine learning and computer science luminaries, entreating the urgent need for AI research to be focused on safety and robustness.

Figure 1: Science fiction portrayal of the dangers of AI

Stuart Russell, Berkeley, Professor of Computer Science, director of the Center for Intelligent Systems, and co-author of the standard textbook Artificial Intelligence: a Modern Approach.
Tom Dietterich, Oregon State, President of AAAI, Professor and Director of Intelligent Systems
Eric Horvitz, Microsoft research director, ex AAAI president, co-chair of the AAAI presidential panel on long-term AI futures
Bart Selman, Cornell, Professor of Computer Science, co-chair of the AAAI presidential panel on long-term AI futures
Francesca Rossi, Padova & Harvard, Professor of Computer Science, IJCAI President and Co-chair of AAAI committee on impact of AI and Ethical Issues
Demis Hassabis, co-founder of DeepMind
Shane Legg, co-founder of DeepMind
Mustafa Suleyman, co-founder of DeepMind
Dileep George, co-founder of Vicarious
Scott Phoenix, co-founder of Vicarious
Yann LeCun, head of Facebook's Artificial Intelligence Laboratory
Geoffrey Hinton, University of Toronto and Google Inc.
Yoshua Bengio, Université de Montréal
Peter Norvig, Director of research at Google and co-author of the standard textbook Artificial Intelligence: a Modern Approach
Oren Etzioni, CEO of Allen Inst. for AI
Guruduth Banavar, VP, Cognitive Computing, IBM Research
Michael Wooldridge, Oxford, Head of Dept. of Computer Science, Chair of European Coordinating Committee for Artificial Intelligence
Leslie Pack Kaelbling, MIT, Professor of Computer Science and Engineering, founder of the Journal of Machine Learning Research
Tom Mitchell, CMU, former President of AAAI, chair of Machine Learning Department
Toby Walsh, Univ. of New South Wales & NICTA, Professor of AI and President of the AI Access Foundation
Murray Shanahan, Imperial College, Professor of Cognitive Robotics
Michael Osborne, Oxford, Associate Professor of Machine Learning
David Parkes, Harvard, Professor of Computer Science
Laurent Orseau, Google DeepMind

Figure 2: Screenshot of a subset of the over 8000 people who have signed the open letter on AI safety

Singularity Hypothesis In theory, how might AI become super-intelligent? How might a computer reach and subsequently surpass human-level abilities in a wide range of tasks?

One line of thought goes as follows: Humans will keep improving AI technology so long as automation brings economic and military benefits to society. The progress in AI technology will occur at a higher rate than progress in human knowledge and intelligence. At some point in the future (known as the Singularity), an AI agent will develop the powerful ability to design a more intelligent AI agent. This self-improvement ability will then be applied recursively over successive generations, culminating in what the statistician I. J. Good terms an 'intelligence explosion.' [4]

This is known as the Singularity Hypothesis.

For context, most machine learning researchers do not think it is very probable. [5] surveyed 352 researchers who published at the 2015 NeurIPS and ICML conferences (21% of the 1634 authors), and asked them for their subjective probability of the Singularity happening two years after 'unaided machines can accomplish every task better and more cheaply than human workers.' The median probability was found to be 10%, even though 48% of respondents agreed with the claim that 'society should prioritize research aimed at minimizing the potential risks of AI.'

Recursive Self-Improvement Humans have the ability to improve themselves, for example, by reading a book to acquire new knowledge. The potential for recursive self-improvement, however, is constrained by the biological realities of a limited memory and a physical body that deteriorates over time and ultimately dies. A priori, there is no reason to think that such constraints will apply to an AI — it can copy its software to a new hard drive and build a new robot to house its 'mind' before 'dying.'

We can conceive of an AI as fundamentally being a computer program written in a given programming language.

One of the hallmarks of a mature programming language is the ability to compile the language in the language itself (i.e. bootstrapping). For example, the first Python compiler, CPython, which is also the reference implementation, is written in C. But eventually, it became feasible to build a Python compiler in Python itself. PyPy is the most prominent such example, and surprisingly (or not), it is actually (~7x) faster than CPython on a wide range of benchmarks using techniques like meta-JIT and meta-tracing [6].

In his Turing award lecture, [7] described a compiler that can 'learn' new patterns via first adding in new source code to encode the pattern, and later compiling itself to erase traces of ever having added the source code. This seems to be a good blueprint for how recursive self-improvement would manifest itself in computer programs. To recursively self-improve itself, a computer program needs at minimum to be a program-handling program, whether that is a compiler, an assembler, a loader, a linker, or even hardware microcode. A compiler is the most plausible candidate for a recursively self-improving AI, because it has the capability of reading its

own source code and modifying it. But that is not the only possibility, since humans can improve themselves without being able to read and modify genes or neural connections.

Neural Networks Generating Neural Networks Deep learning, also known as artificial neural networks, represents the most powerful form of AI known to date. Deep learning programs have shown superhuman performance in domains as diverse as image recognition [8], speech recognition [9], Atari games [10], and Go competitions [11], while displaying near-human performance in speech synthesis [12] and machine translation [13], among numerous other tasks. Furthermore, deep learning has also powered state of the art generative modeling techniques capable of synthesizing photo-realistic images [14], human-like speech [12], natural language sentences [15], temporally consistent videos [16], and even protein structures [17], among other objects of interest.

The ultimate challenge, however, is not generating images, or audio, or video. The ultimate challenge for a neural network generator is to generate other neural networks. Deep neural networks thus present a unique opportunity to explore the potential for recursive self-improvement in deep learning programs. We term the concept of neural network based generation of neural networks Autogenerative Networks, and it is the primary object of research interest in this thesis.

Chapter 1: Overview

1.1 Preliminaries

1.1.1 Limits of Recursive Computation

One way a program P can improve itself it to first diagnose if it can solve task T, and then make steps towards learning how to solve it if it cannot do so already. If T is the question of determining whether an arbitrary program will eventually halt given enough time, then this is an impossible task, since the Halting Problem is undecidable.

This implies that there are certain theoretical limits inherent in recursive computation. We note four relevant results in the theory of computation that demarcate the expressiveness of program-handling programs and neural networks in general.

Firstly, [18] prove the non-existence of universality for a large class of finite state automata. A Turing Machine is known to be universal (i.e. it can simulate other Turing Machines), but it is unclear if there are automata weaker than a Turing Machine that can simulate other automata in its class.

Secondly, Rice's theorem states that non-trivial properties about Turing Machines are undecidable (by Turing Machines). More specifically, given some non-trivial language L (i.e. there exists a Turing Machine that recognizes L and there exists a Turing Machine that recognizes its complement), it is impossible to decide if an arbitrary Turing Machine belongs to L.

Thirdly, feed-forward neural networks are known to be universal function approximators. Specifically, a neural network with one hidden layer can approximate any continuous function on a compact subset of Euclidean space, to any degree of precision [19, 20]. This is done via a two-step process: (1) Sigmoidal activation functions can approximate a step function given appropriate choice of weights, (2) Any continuous function on a compact subspace can be approximated with

a combination of step functions.

Fourthly, recurrent neural networks have been shown to be Turing Complete [21]. 2-way Pushdown Automata are computationally equivalent to a Turing Machine, and with suitable choice of weights, there exists a recurrent network that can emulate a 1-way Pushdown Automata. Turing completeness can hence be achieved by assimilating two recurrent networks into one.

1.1.2 Importance of Good Representations

Before a neural network can operate on another neural network, it has to have a mechanism for reading in a neural network as data.

Humans understand concepts in different levels of abstraction, and the wrong level of abstraction will often prevent understanding from taking place at all. As one example, probing the voltage levels in the circuitry of a microprocessor, which is akin to probing neural spike trains in a brain, makes it very difficult to understand anything about the actual information processing done by the microprocessor [22].

Likewise, a program has to be given data in the appropriate encoding for it to process the data efficiently and successfully. NLP researchers found that using one-hot vectors to encode words in a vocabulary leads to sparse representations and do not meaningfully describe the semantic similarity of related words. The use of word embeddings circumvent this problem by learning linear latent structures. Famously, [23] showed that 'King' − 'Queen' is similar to 'Man' − 'Woman.' Going further, subword level embeddings have proved useful in named entity recognition [24], part-of-speech tagging [25], dependency parsing [26], among other common NLP tasks.

Clearly, finding the 'right' data representations for a neural network is essential to the success of autogenerative Networks.

[27] proposed NeuroEvolution of Augmenting Topologies (NEAT) as a method to implement Topology and Weight Evolving Artificial Neural Networks (TWEANNs), representing the connections and weights in a neural network (phenotype) as an annotated string (genotype). The string is annotated with historical markers that track when a new connection is made or when a new neu-

ron is introduced into the network. This helps to allow (1) disparate topologies to cross over in a meaningful way, (2) preserve topological innovation so that niches do not disappear prematurely, and (3) minimize topologies without the need for a fitness function that measures complexity.

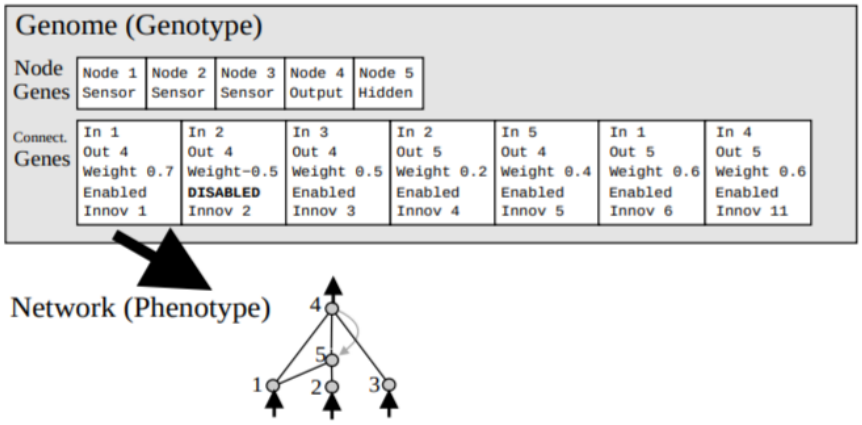

Figure 1.1: Depiction of NEAT

Many extensions to the basic NEAT algorithm have been proposed.

HyperNEAT [28] uses a Compositional Pattern Producing Network (CPPN) to indirectly encode the network by mapping each connection with a weight, allowing symmetries in the CPPN to generate weights for bigger networks without further training. In a Differentiable Pattern Producing Network (DPPN), the topology is evolved while the weights are learned via backpropagation [29]. When the DPPN was used to produce the weights for a denoising autoencoder trained on images, an approximate convolutional structure was found embedded within the fully connected architecture. While NEAT is a direct encoding, the use of CPPN and DPPN represents indirect encodings since a coordinate system is needed to reproduce the neural network from the encoding.

DeepNEAT encodes connections between layers instead of individual neurons, the type of each layer (convolutional, fully connected, or recurrent), properties of each layer (number of neurons, kernel size, activation function, etc), and a table of weights for each layer. CoDeepNEAT uses DeepNEAT as a subroutine, and co-evolves the topology, components and hyperparameters of a deep neural network, allowing neuroevolution to optimize different pieces of the model all at once [30]. [31] use a similar strategy to represent possible neural networks as data, but opts to use

reinforcement learning instead of evolution to select components for the network.

While NEAT itself used evolution to optimize both the topology and weights of the network, modern extensions of NEAT use evolution to optimize the topology, while using backpropagation to optimize the weights. This suggests that it might not be necessary to optimize the topology and the weights simultaneously. It is possible to first train an overparametrized model and subsequently (1) compress it to a smaller model [32], (2) use it as a teacher model to guide the training of a slimmer student model [33, 34], or (3) selectively use different portions of the large network for different tasks and settings [35, 36, 37, 38]. Some recent (empirical) evidence even suggests that we need large networks to maximize the chances of success for gradient descent based optimization [39].

Finally, [40] proposed to view a neural network as essentially different modules of weights and examined different orderings for re-using these weights. [41] proposed the memory bank encoding, which views different neural network operations as reading and writing to a memory bank.

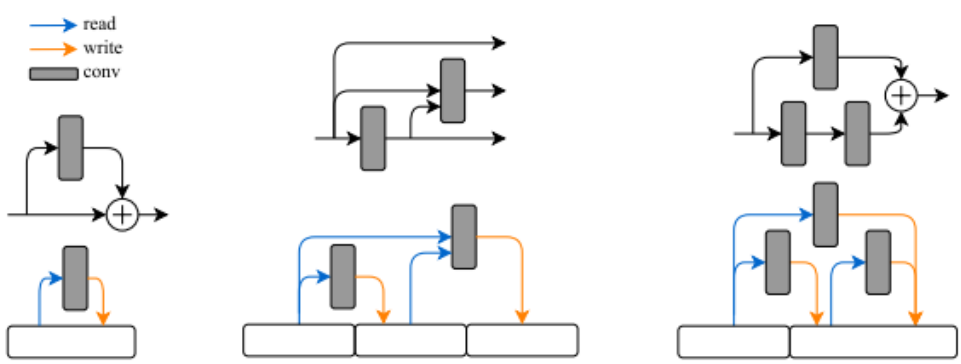

Figure 1.2: Memory Bank encoding for ResNet, DenseNet, and FractalNet

1.2 Overview of Common Deep Learning Methods

In this section, we briefly cover some of the main deep learning based methods that appear in the rest of the thesis.

Adam is an optimizer based on stochastic gradient descent. It scales the gradient g_t at every time step using a running mean (with multiplicative weights β_1, β_2) of the first and second order

raw moments m_t, v_t.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_t &= \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
\tag{1.1}
$$

Batch Normalization is a regularization mechanism that centers the activations of each layer in a neural network using batch statistics to mitigate layer-wise covariate shift. It stores a running mean of these statistics during training time to be used at test time. After the normalization, it uses learnable parameters γ and β to scale and shift the normalized activations.

$$
\begin{aligned}
\mu_{\mathcal{B}} &= \frac{1}{m} \sum_{i=1}^{m} x_i \\
\sigma_{\mathcal{B}}^2 &= \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \\
\hat{x}_i &= \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\
\mathrm{BN}(x_i)_{\gamma,\beta} &= y_i = \gamma \hat{x}_i + \beta
\end{aligned}
\tag{1.2}
$$

Dropout is a regularization mechanism that randomly drops activations with probability p at training time. At test time, none of the activations are dropped, but we scale them by $1-p$ so that their order of magnitude is similar on expectation to that at training time.

$$
\mathrm{Dropout}(x_i) = \mathbb{1}_z x_i, \quad z \sim \text{Bernoulli(1-p)} \tag{1.3}
$$

Gradient-based Meta-Learning MAML is the canonical algorithm upon which most gradient-based meta-learning methods are founded. Given T tasks and K steps of gradient descent, MAML backpropagates through the gradient descent process itself for each task to compute a meta-gradient that is then used to make weight updates.

$$\begin{aligned} \theta_{t,0} &= \theta \\ \theta_{t,k} &= \theta - \alpha \nabla_{\theta_{k-1}} \mathcal{L}_t(\theta_{k-1}) \\ \theta &= \theta - \beta \sum_t \nabla_\theta \mathcal{L}_t(\theta_K) \end{aligned} \tag{1.4}$$

Hypernetworks are meta neural networks H parametrized by ϕ that generate the weights θ of a main neural network G from some embedding e to minimize a given task loss $\mathcal{L}$. Since ϕ are the model parameters in this case, gradient descent optimizes ϕ and not θ.

$$\begin{aligned} \theta &= H_\phi(e) \\ \phi &= \phi - \nabla_\phi \mathcal{L}(G_\theta) \end{aligned} \tag{1.5}$$

Variational Auto-Encoders are auto-encoders that constrain the latent space to be close to a Gaussian by optimizing a variational lower bound on the marginal likelihood. This resolves to a variational term and the reconstruction loss (hence, the name variational auto-encoder).

$$\mathcal{L}(\theta) = \frac{1}{2} \sum_{j=1}^{J} \left(1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2\right) + \frac{1}{L} \sum_{l=1}^{L} \log p_\theta(x|z), \quad z = \mu + \sigma\epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{1.6}$$

1.3 Related Work

1.3.1 Early Prior Work (Pre-2000s)

Early prior work from researchers that we consider machine learning luminaries today proposed several candidates for an autogenerative network. Jurgen Schmidhuber's PhD thesis was titled "Evolutionary Principles in Self-Referential Learning" [42], which proposed programming recursively self-improving genetic algorithms. He also proposed reinforcement learning with self-modifying policies [43], and neural networks that can modify their own weights [44]. Yoshua Bengio proposed to parametrize the learning rule for a neural network, and then meta-learn the learning rule itself by optimizing those parameters with an evolutionary algorithm [45, 46]. Sebastian Thrun described multiple different approaches to meta-learning including multi-task learning (using a single algorithm to learn to solve multiple tasks at the same time by sharing knowledge) and continual learning (sequential, rather than simultaneous, multi-task learning where the challenge is to not forget previously encountered tasks) [47].

1.3.2 Recent Prior Work

While early prior work mostly involved abstract thought experiments and proofs of concept, recent prior work tends to make specific and concrete contributions to a specific application of meta-learning with neural networks.

A quote from Richard Hamming's talk titled 'Learning to Learn: You and Your Research' [48] is appropriate here. Hamming says:

> "In all the 30 years I spent at Bell Telephone Laboratories (before it was broken up) no one to my knowledge worked on time travel, teleportation, or anti-gravity. Why? Because they had no attack on the problem. Thus an important aspect of any problem is that you have a good attack, a good starting place, some reasonable idea of how to begin."

Making true progress on abstract ideas like autogenerative networks is difficult, and the best way to accomplish this ambitious research program might be to identify and work on small well-defined problems along the way. Below, we enumerate a non-exhaustive list of such research problems:

1. **Hyperparameter Optimization** We can use a meta neural network to model the search space for the hyperparameters of a different neural network [49]. [31] used a policy gradient neural network as a reinforcement learning agent to select architectural choices (like the width of the convolution kernel or the operations in a recurrent cell) in the design of another neural network. This is known as 'Neural Architecture search' (NAS). NAS is extremely computationally demanding. Using NAS to find a good convolutional architecture for CIFAR-10 required training and comparing between 12800 different deep neural networks. Several efficiency improvements to the original idea have since been proposed. [50] used a surrogate function to estimate the performance of an NAS candidate before training it, thus reducing the number of candidates (5x reduction) that have to be evaluated. [51] made the observation that each component of a candidate network did not have to be trained from scratch each time. Training time can be drastically reduced (a 1000x reduction) by starting training from the weights of the component in another candidate network. [52] proposed to make NAS a differentiable process through a continuous relaxation of the discrete actions made by the reinforcement learning agent. [53] used NAS-like techniques to compress existing models by searching for models under compute constraints. Furthermore, there have been proposals to turn NAS into a one-shot learning problem through the use of a hypernetwork [41], and the use of a single giant network that selectively drops out components [35, 37]. For a more comprehensive survey on NAS, please refer to [54]. NAS is arguably the most active area of research into autogenerative networks in the machine learning community, thus we made a conscious decision to avoid it in this thesis and instead attend to under-studied applications of autogenerative networks in our research.

2. **Hypernetworks** [55] coined the term 'Hypernetwork' to describe a meta neural network

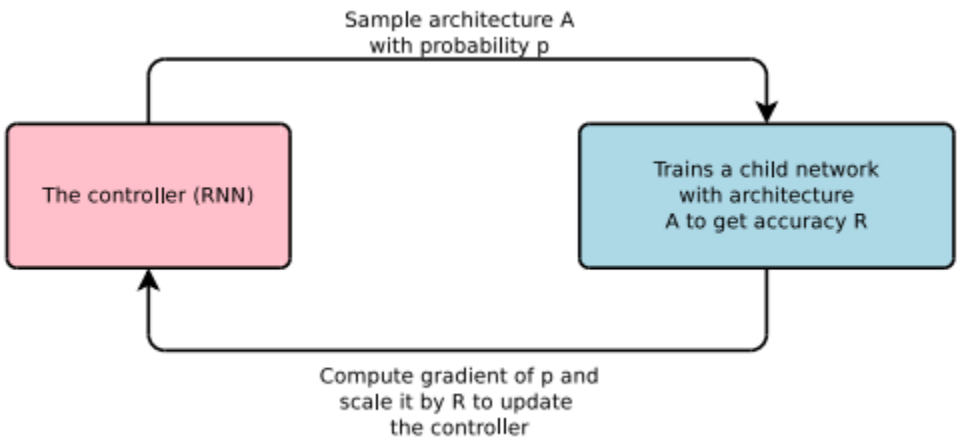

Figure 1.3: Neural Architecture Search

that generates the weights of a main neural network with a differentiable function. This allows changes in the weights of the generated main network to be backpropagated to the hypernet itself. Hypernetworks were originally intended as a model compression mechanism through soft weight-sharing, where the meta network essentially compresses the much larger number of model parameters in the main network into a significantly smaller model. Today, hypernetworks have found numerous applications including but not limited to: weight pruning [56], neural architecture search [41, 57], Bayesian neural networks [58, 59, 60, 61, 62], multi-task learning [63, 64, 65, 66, 67], continual learning [68], generative models [69, 70], ensemble learning [71], hyperparameter optimization [72], and adversarial defense [73].

3. **Transfer Learning** The meta network can store information from previously encountered tasks and generate neural networks that adapt that information to the task at hand [74, 75]. NAS can be seen as an example of transfer learning, since we typically learn the architecture by training on one dataset, but then transfer that knowledge by testing on a different dataset. Instead of doing a search over all possible architectures, it is also possible to fix a given architecture and allow a certain component of the architecture to vary. This component is said to be meta-learned, since it is learned by another neural network instead of fixed by a human designer. [74, 75] used an external LSTM to meta-learn the optimization function

used to update a child network. [76] used NAS-like techniques to meta-learn an activation function. The Discriminator in Generative Adversarial Networks [77] can be thought of as a network that meta-learns the cost function in place of the standard cross-entropy loss.

4. **Few-Shot Learning** Humans learn using small number of examples, which is a stark contrast to the vast number of labeled training examples a deep neural network needs for supervised learning. Machine learning, under training conditions that are limited to small number of training examples, is known as 'Few-Shot Learning'. While the number of data points is small, the number of datasets can be comparatively large to enable transfer learning. This can be done using model-based methods like a hypernetwork, as described above [41], or through gradient-based methods where we treat gradient descent itself as a differentiable function and backpropagate through it [78].

5. **Multi-Task Learning** More broadly, instead of having a strict distinction between a meta network and a task network, we can also have a very large meta network that contains small module networks that adapt to a given task [48]. When trained on multiple different tasks at once, this is known as 'Multi-Task Learning'. When trained on these tasks in a sequential fashion, multi-task learning is known as 'Continual Learning' or 'Lifelong Learning'. This scenario is arguably more challenging because of the tendency for neural networks to memorize recent information and forget past information [79]. One way to overcome such catastrophic forgetting is to use a hypernet to generate networks conditioned on a task embedding where the generation process can be regularized to minimize forgetting [68].

1.4 Summary of Our Contributions

In this thesis, we choose to study various different themes surrounding autogenerative networks that are relatively under-studied by the academic community. The major difference between early prior work and recent work is that the former is too abstract and conducted using many thought experiments, while the latter might be overly focused on specific applications. In this PhD

thesis, we walk the middle ground by exploring ideas that might seem overly ambitious for immediately practical applications, but are nonetheless grounded in specific empirical experiments.

In Chapter 2, we start by training neural networks that can generate individual weights for another network. When turned on itself, these techniques enable a computational model for self-replication, and have been used in chemical simulations [80]. In Chapter 3, we generate, from embeddings, entire neural networks instead of individual weights, and show how they can be used to improve the interpretability of reinforcement learning agents. The methods of neural network generation used in Chapters 2 and 3 do not scale well to big models, because of the non-differentiability of the generation process. Therefore, in Chapters 4 and 5, we study initialization and optimization issues in hypernetworks respectively, and propose solutions to some of these problems while highlighting remaining open challenges. Finally, we turn to implicit models of network generation like gradient-based meta-learning (Chapter 6), where we generate the gradients and not the weights, and logical networks (Chapter 7), which combine both discrete and continuous optimization in a single neural network.

Together, the chapters in this thesis contribute novel proposals for non-differentiable neural network generation mechanisms, significant improvements to existing differentiable network generation mechanisms, and an assimilation of different learning paradigms in auto-generative networks.

Below, we provide a more in-depth summary for each of the upcoming chapters in this thesis.

Chapter 2: Neural Network Quine We describe how to build and train self-replicating neural networks. The network replicates itself by learning to predict its own weights via a loss function that can be optimized with either gradient-based or non-gradient-based methods. We also describe a method called generational replication to train the network without explicit optimization by injecting the network with predictions of its own parameters. The best solution for a self-replicating network was found by alternating between generation and optimization steps. Finally, we describe a design for a self-replicating neural network that can solve an auxiliary task like MNIST image classification. Interestingly, we observe that there is a trade-off between the network's ability to

classify images and its ability to replicate, but training is biased towards increasing its specialization at image classification at the expense of replication. This is analogous to the trade-off between reproduction and survival observed in nature. Among other reasons, a replication mechanism for artificial intelligence is useful because it introduces the possibility of intelligent artificial life, allowing for self-improving AI agents where improvements result via natural selection.

Chapter 3: Agent Embeddings We show that it is possible to reduce a high-dimensional object like a neural network agent into a low-dimensional vector representation with semantic meaning that we call *agent embeddings*, akin to word or face embeddings. This can be done by collecting examples of existing networks, vectorizing their weights, and then learning a generative model over the weight space in a supervised fashion. We investigate a pole-balancing task, Cart-Pole, as a case study and show that multiple *new* pole-balancing networks can be generated from their agent embeddings without direct access to training data from the Cart-Pole simulator. In general, the learned embedding space is helpful for mapping out the space of solutions for a given task. We observe in the case of Cart-Pole the surprising finding that good agents make different decisions despite learning similar representations, whereas bad agents make similar (bad) decisions while learning dissimilar representations. Linearly interpolating between the latent embeddings for a good agent and a bad agent yields an agent embedding that generates a network with intermediate performance, where the performance can be tuned according to the coefficient of interpolation. Linear extrapolation in the latent space also results in performance boosts, up to a point.

Chapter 4: Hypernetwork Initialization Hypernetworks are meta neural networks that generate weights for a main neural network in an end-to-end differentiable manner. Despite extensive applications ranging from multi-task learning to Bayesian deep learning, the problem of optimizing hypernetworks has not been studied to date. We observe that classical weight initialization methods like [81] and [82], when applied directly on a hypernet, fail to produce weights for the mainnet in the correct scale. We develop principled techniques for weight initialization in hypernets, and show that they lead to more stable mainnet weights, lower training loss, and faster convergence.

Chapter 5: Hypernetwork Optimization Training hypernetworks by gradient descent results in different update rules for the main network due to the reparametrization. We study a special class of replicator hypernetworks called hypergenerative networks where both the input and output are the same neural network, and derive update rules for simple hypernetwork architectures. Different hypergenerative networks give rise to different update rules depending on their architecture, and it can be shown that standard gradient descent falls under a special case. Interestingly, we show that some of these update rules can be generalized so that when they are applied in a recursive fashion to train the hypernetworks, we recover the original updates. We verify experimentally that some of these non gradient descent update rules can be used to train big neural networks successfully with comparable levels of accuracy as standard gradient descent.

Chapter 6: Gradient-Based Meta-Learning The success of gradient-based meta-learning is primarily attributed to its ability to leverage related tasks to learn task-invariant information. However, the absence of interactions between different tasks in the inner loop leads to task-specific over-fitting in the initial phase of meta-training. While this is eventually corrected by the presence of these interactions in the outer loop, it comes at a significant cost of slower meta-learning. To address this limitation, we explicitly encode task relatedness via an inner loop regularization mechanism inspired by multi-task learning. Our algorithm shares gradient information from previously encountered tasks as well as concurrent tasks in the same task batch, and scales their contribution with meta-learned parameters. We show using two popular few-shot classification datasets that gradient sharing enables meta-learning under bigger inner loop learning rates and can accelerate the meta-training process by up to 134%.

Chapter 7: Logical Networks SATNet is an award-winning MAXSAT solver that can be used to infer logical rules and integrated as a differentiable layer in a deep neural network [83]. It had been shown to solve Sudoku puzzles visually from examples of puzzle digit images, and was heralded as an impressive achievement towards the longstanding AI goal of combining pattern recognition with logical reasoning. In this chapter, we clarify SATNet's capabilities by showing that in the absence of intermediate labels that identify individual Sudoku digit images with

their logical representations, SATNet completely fails at visual Sudoku (0% test accuracy). More generally, the failure can be pinpointed to its inability to learn to assign symbols to perceptual phenomena, also known as the symbol grounding problem [84], which has long been thought to be a prerequisite for intelligent agents to perform real-world logical reasoning. We propose an MNIST based test as an easy instance of the symbol grounding problem that can serve as a sanity check for differentiable symbolic solvers in general. Naive applications of SATNet on this test lead to performance worse than that of models without logical reasoning capabilities. We report on the causes of SATNet's failure and how to prevent them.

1.5 Publications

Some of the research in this thesis has been published and presented at conferences and workshops. We list the relevant papers below:

1. *Chang, O., & Lipson, H. (2018). Neural Network Quine. In Artificial Life Conference Proceedings (pp. 234-241). One Rogers Street, Cambridge, MA 02142-1209 USA journals-info@mit.edu: MIT Press.*

2. *Chang, O., Kwiatkowski, R., Chen, S., & Lipson, H. (2019). Agent Embeddings: A Latent Representation for Pole-Balancing Networks. In Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (pp. 656-664). International Foundation for Autonomous Agents and Multiagent Systems.*

3. *Chang, O., Flokas, L., & Lipson, H. (2019). Principled Weight Initialization for Hypernetworks. In Proceedings of the International Conference on Learning Representations. Oral Presentation (top 1.9% of 2594 papers).*

4. *Chang, O., Flokas, L., & Lipson, H. (2020). Accelerating Meta-Learning by Sharing Gradients. BeTR-RL Workshop at ICLR 2020.*

5. *Chang, O., Flokas, L., Lipson, H., & Spranger, M. (2020). Assessing SATNet's Ability to Solve the Symbol Grounding Problem. In Proceedings of the Thirty-fourth Annual Conference on Neural Information Processing Systems.*

Chapter 2: Neural Network Quine

2.1 Introduction

The concept of an artificial self-replicating machine was first proposed by John von Neumann in the 1940s prior to the discovery of DNA's role as the physical mechanism for biological replication. Specifically, Von Neumann demonstrated a configuration of initial states and transformation rules for a cellular automaton that produces copies of the initial cell states after running for a fixed number of steps [85]. [86] later coined the term 'quine' in *Gödel, Escher, Bach: an Eternal Golden Braid* after the philosopher Willard Van Orman Quine, to describe self-replicating expressions such as: 'is a sentence fragment' is a sentence fragment.

In the context of programming language theory, quines are computer programs that print their own source code. A trivial example of a quine is the empty string, which in most languages, the compiler transforms into the empty string. The following code snippet is an example of a non-trivial Python quine written in two lines.

```
s = 's = %r\nprint(s%%s)'
print(s%s)
```

In this chapter, we identify and solve the challenges involved in building and training a self-replicating neural network. Specifically, we propose to view a neural network as a differentiable computer program composed of a sequence of tensor operations. Our objective then is to construct a neural network quine that prints its own weights.

We tested our approach using three distinct classes of methods: gradient-based optimization methods, non-gradient-based optimization methods, and a novel method called generational replication. We further designed a neural network quine which has an auxiliary objective in addition to

the job of self-replication. In this chapter, the chosen auxiliary task is MNIST image classification [88], which involves classifying images of digits from 0 to 9, and is commonly used as a 'hello world' example for machine learning.

We observed a trade-off between the network's ability to self-replicate and its ability to solve the auxiliary task. This is analogous to the trade-off between reproduction and survival observed in nature. The two objectives are usually aligned, but for example, when an animal has been put in starving conditions, its sex hormones are usually down-regulated to optimize for survival at the expense of reproduction. The opposite occurs as well: for example, in male dark fishing spiders, the act of copulation results in a sudden irreversible change to its blood pressure, immobilizing it and leaving it vulnerable to cannibalization by the female spider [89].

2.1.1 Motivations

Modern artificial intelligence is primarily powered by deep neural networks for applications as diverse as tracking moving objects [90], detecting diabetic retinopathy [91], synthesizing human-like speech [12, 92], and executing strategic decisions in Starcraft [93]. In line with the ambition of going beyond current AI technology, we list several motivations for studying self-replicating neural networks.

- Biological life began with the first self-replicator [94], and natural selection kicked in to favor organisms that are better at replication, resulting in a self-improving mechanism. Analogously, we can construct a self-improving mechanism for artificial intelligence via natural selection if AI agents had the ability to replicate themselves.

- Neural networks are capable of learning powerful representations across many different domains of data [95]. But can a neural network learn a good representation of itself? Self-replication involves a degree of self-awareness, and can be viewed as enforcing a soft weight-sharing constraint between a network and past versions of itself, which is helpful for lifelong learning.

- Learning how to enhance or diminish the ability for AI programs to self-replicate is useful for computer security. For example, we might want an AI to be able to execute its source code without being able to read or reverse-engineer it, either through its own volition or interaction with an adversary.

- Self-replication functions as the ultimate mechanism for self-repair in damaged physical systems [96]. The same may apply to AI, where a self-replication mechanism can serve as the last resort for returning a damaged or out-of-control AI system back to normal.

2.1.2 Related Work

Quines have been written for a variety of programming languages. The Quine Page [97] contains code contributions of quines written in 55 different languages. An Ouroboros set of programs extends the concept of a quine by having a program in language A generate the source code for a program in language B, which then generates the source code for a program in a language C, and so on, until it finally generates back the source code for the initial program in language A. [98] made an Ouroboros with 128 programming languages in it.

There has also been work done in making physical self-replicators. Notable examples include molecules [99], polymers [100], and robots [96].

Our work focuses on building a self-replication mechanism via weight prediction. [101] demonstrated the presence of redundancy in neural networks by using a portion of the weights to predict the rest. There are also neural networks that can modify the weights of other neural networks [102, 103], which have been shown to be useful in meta-learning an optimizer [75, 74]. [104] proposed an architecture and a training algorithm for a self-referential recurrent neural network, which is philosophically very similar to our work in that the network refers to itself rather than another network. To our knowledge, our work is the first to attempt the task of self-replication in neural networks.

2.2 Building the Network

2.2.1 How can a neural network refer to itself?

Problem with Direct Reference

A neural network is parametrized by a set of parameters Θ, and our goal is to build a network that outputs Θ itself. This is difficult to do directly. Suppose the last layer of a feed-forward network has A inputs and B outputs. Already, the size of the weight matrix in a linear transformation is the product AB which is greater than B for any $A > 1$.

We also looked at open-source implementations of two popular generative models for images, DCGAN [105] and DRAW [106]. They use 12 million and 1 million parameters respectively to generate MNIST images with 784 pixels.

In general, the set of parameters Θ is a lot larger than the size of the output. To circumvent this, we need an indirect way of referring to Θ.

Indirect Reference

HyperNEAT [107] is a neuro-evolution method that describes a neural network by identifying every topological connection with a coordinate and a weight. We pursue the same strategy in building a quine. Instead of having the quine output its weights directly, we shall set it up so that it inputs a coordinate (in a one-hot encoding) and outputs the weight at that coordinate.

This overcomes the problem of Θ being larger than the output, since we are only outputting a scalar Θ_c for each coordinate c.

2.2.2 Vanilla Quine

We define the *vanilla quine* as a feed-forward neural network whose only job is to output its own weights.

Suppose the number of weights is A, and the number of units in the first hidden layer is B, then the size of the projection matrix would be the product AB which is greater than A for any

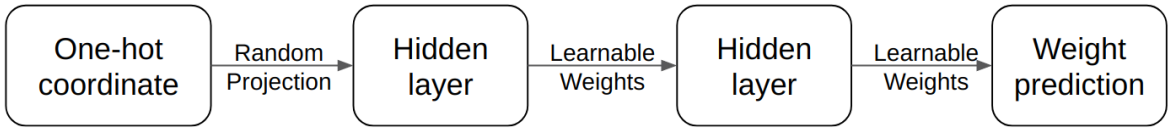

Figure 2.1: Structure of a vanilla quine

$B > 1$. Hence, we cannot have the projection itself be a parameter of the network due to the one-hot representation. We thus decide to use a fixed random projection to connect the one-hot encoding of the coordinate to the hidden layer. All other connections, namely the connections between the hidden layers as well as the connections between the last hidden layer and the output layer, are variable parameters of the neural network.

Von Neumann argued that a non-trivial self-replicator necessarily includes three components that by themselves do not suffice to be self-replicators: (1) a description of the replicator, (2) a copying mechanism that can clone descriptions, and (3) a mechanism that can embed the copying mechanism within the replicator itself [85]. In this case, the coordinate system that assigns each of the weights a point in the one-hot space corresponds to (1). The function computed by the neural network corresponds to (2). The fixed random projection corresponds to (3). We explain below reasons for our choices of (1), (2), and (3), while keeping in mind that alternatives to them are interesting future research directions.

(1) One-hot Input Encoding

A one-hot encoding is a vector that contains exactly one 1 and is 0 everywhere else. If we directly input the coordinate instead of using a one-hot encoding, then the network will not be sufficiently expressive. This is because for any coordinate c, the difference between $f(c)$ and $f(c+1)$ is constrained by the network's Lipschitz bound, hence the network cannot accurately output the weights at c and c+1 if their difference is sufficiently big. We demonstrate a visualization of this in Figure 2.2: contiguous weights might be very different, but contiguous outputs cannot be

very different.

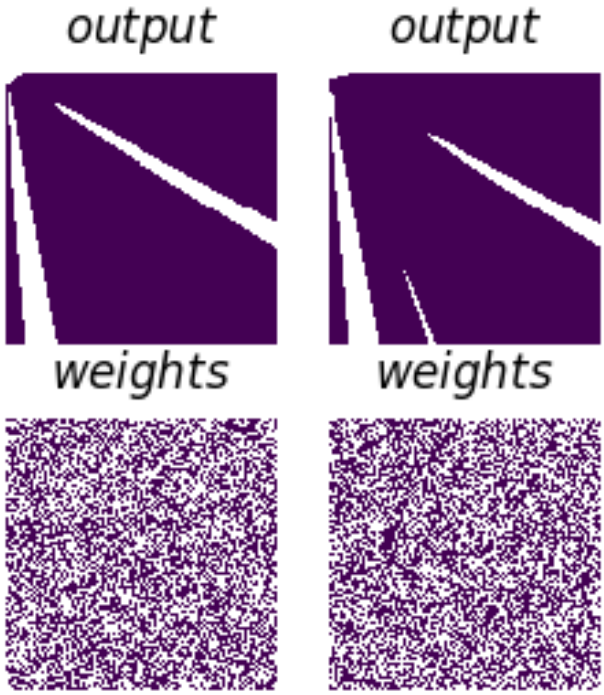

Figure 2.2: Log-normalized illustration of a quine without one-hot encoding

(2) Multi-Layered Perceptrons

$$y_i = \sigma_i(W_i x_i + b_i) \tag{2.1}$$

Multi-layered perceptrons (MLPs) are feed-forward neural networks that consist of repeated applications of Equation 2.1, where at the ith layer of the network, σ_i is an activation function, W_i a weight matrix, b_i a bias vector, x_i the input vector, and y_i the output vector. MLPs are known to be good function approximators, specifically a feedforward neural network with at least one hidden layer forms a class of functions that is dense in the space of continuous functions under a compact domain [20, 19]. While not precluding other kinds of generative neural network architectures, this makes an MLP seem like a suitable candidate for a neural network quine, because we think it is expressive enough to derive and store a representation of itself.

(3) Random Projections

We think random projections are a good choice as an embedding layer to connect a one-hot representation into the network because of their distance-preserving property [109] and the fact that random features have been shown to work well both in theory and practice [110]. Indeed, they form a key component of Extreme Learning Machines [111] which are feed-forward neural networks that have proven useful in classification and regression problems.

2.2.3 Auxiliary Quine

We define the *auxiliary quine* to be a vanilla quine that solves an auxiliary task in addition to self-replication. It is responsible for taking in an auxiliary input and returning an auxiliary output.

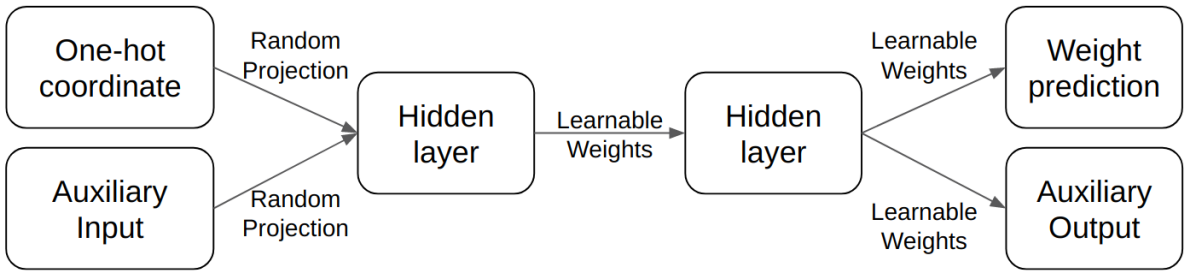

Figure 2.3: Structure of an auxiliary quine

In this chapter, we chose image classification as the auxiliary task. The MNIST dataset [88] contains square images (28 pixels by 28 pixels) of handwritten digits from 0 to 9, which are going to be what is fed in as the auxiliary input. It is possible to make the connection from the auxiliary input to the network a parameter rather than a random projection, but in this chapter, we only report results for the latter. The auxiliary output is a probability distribution over the ten classes, where the class with the maximum probability will be chosen as the predicted classification. 60000 images are used for training and 10000 images are used for testing; we have no need for a validation set since we are not strictly trying to optimize for the performance of the classifier. Our primary aim in this chapter is to demonstrate a proof of concept for a neural network quine, which makes

MNIST a suitable auxiliary task as it is considered an easy problem for modern machine learning algorithms.

2.3 Training the Network

2.3.1 Network Architecture

Before describing how the neural network quines are trained, we specify the exact network architecture used in our experiments below for both the vanilla quine and the auxiliary quine. In both cases, they are MLPs composed of two hidden layers with 100 hidden units each where every layer is followed by a SeLU [112] activation function. In the case of the auxiliary quine, the one-hot coordinate is projected to the first 50 hidden units, while the MNIST input is projected to the next 50 hidden units. The auxiliary output is a vector of size 10 (number of classes) computed by a softmax.

The total number of parameters is 20100 for the vanilla quine, and 21100 for the auxiliary quine. The nature of the quine problem and our choice of the one-hot encoding means that the input vector will be of the same size as the number of parameters. These are small networks by modern deep learning standards where millions of parameters are the norm, but it is a challenge to handle input vectors with dimensions much larger than 20000.

2.3.2 How do we train a neural network quine?

Self-Replicating Loss

We define the *self-replicating loss* to be the sum of the squared difference between the actual weight and its predicted value. A vanilla quine is achieved when this loss is exactly zero. Because of numerical imprecision errors, we can expect that in practice, optimizing this loss will nonetheless result in a number slightly above zero, except for the trivial *zero quine* where all the weights are exactly zero to begin with.

$$L_{SR} = \sum_{c \in C} \left\| f_\Theta(c) - \Theta_c \right\|_2^2 \tag{2.2}$$

Auxiliary Loss

It is possible to jointly optimize an existing loss function with the self-replicating loss so that a neural network gains the ability to self-replicate in addition to an auxiliary task it specializes in. We define the *auxiliary loss* to be the sum of the self-replicating loss L_{SR} and the loss from the auxiliary task L_{Task}, with a hyperparameter λ to scale both losses to a similar magnitude. An auxiliary quine can be trained by optimizing on the auxiliary loss, but we do not expect to see a near-zero loss, unless it is also perfect at the auxiliary task. In our MNIST experiment, L_{Task} is the cross-entropy loss, which is commonly used for classification problems.

$$L_{Aux} = L_{SR} + \lambda L_{Task} \tag{2.3}$$

Training Methods

There are three distinct classes of methods that we can use to train our neural network quines.

- **Gradient-based methods** Stochastic gradient descent (SGD) and its variants are the workhorse algorithm for training deep neural networks today. In our case, the loss function is a moving target, since Θ_c changes after each gradient update. Updating the loss function after every mini-batch update is expensive. To avoid that, we split the set of possible coordinates into random mini-batches of size 10, and update the loss function after every training epoch. In other words, each training epoch will consist of running through the set of all possible coordinates. We do not use a validation set for our experiments, while the test loss is computed at the end of every training epoch after updating the loss function. Below is pseudo-code for training a vanilla quine. A similar procedure is used to train an auxiliary quine with L_{SR} replaced with L_{Aux} to account for the auxiliary task.

- **Non-gradient-based methods** Optimization methods that do not make use of gradient information can also be used to train neural networks. For example, evolutionary algorithms have been used successfully to train reinforcement learning agents with over four million pa-

Algorithm 1: Pseudo-code for training a vanilla quine via optimization

```
Initialize set of parameters Θ_C
Initialize number of training epochs T
for t ← 0 to T do
    Θ_t := Θ_C
    Divide Θ_t into random mini-batches
    for each mini-batch do
        Compute L_SR
        Θ_C := optimize(Θ_C, L_SR)
```

rameters [113, 114]. For the same reasons of computational efficiency as mentioned above, we shall choose to execute non-gradient-based optimization in mini-batches. (The training algorithm is identical to the pseudo-code shown above, except with `optimize` being non-gradient-based) We only consider hill-climbing in this chapter, which is equivalent to an evolutionary algorithm with a population frontier of size one.

- **Generational Replication** Perhaps somewhat surprisingly, it is also possible to train a vanilla quine without explicitly optimizing for the self-replicating loss. We do so by replacing the current set of parameters with the weight predictions made by the quine. Each such replacement is called a *generation*. We then alternate between running a generation and a round of optimization to achieve a low but non-trivial self-replicating loss. We note that generational replication is sensitive to choices of weight initialization and activation function.

2.4 Results and Discussion

In the experimental results produced below, we used a mini-batch of size 10 for training. λ in L_{Aux} and the temperature for the softmax in the auxiliary output are set to 0.01.

2.4.1 Vanilla Quine

We trained a vanilla quine with classical SGD ($lr = 0.01$), SGD with momentum ($lr = 0.01$, $\rho = 0.9$), ADAM [115], Adagrad [116], Adamax [115], and RMSprop [117] with default

Algorithm 2: Pseudo-code for Generational Replication

```
Initialize set of parameters Θ_C
Initialize number of generation epochs G
Initialize number of optimization epochs T
for g ← 0 to G do
    // Optimization
    for t ← 0 to T do
        Θ_t := Θ_C
        Divide Θ_t into random mini-batches
        for each mini-batch do
            Compute L_SR
            Θ_C := optimize(Θ_C, L_SR)
        // Generation
        for c ← C do
            Θ_c := f_{Θ_C}(c)
```

hyperparameter settings on the self-replicating loss for 30 epochs. The quine was initialized with the same procedure as in [82], and the initial loss L_{SR} prior to any training was 90.16. We observe in Figure 2.4 that Adamax performed the best, while Adagrad exhibited increasing loss rather than plateauing. RMSprop (not plotted) was found to explode the loss right from the start of training. We carried on training the quine on Adamax for 100 epochs, achieving a best test loss of 32.10 by the end of training, which is a third of its pre-trained value.

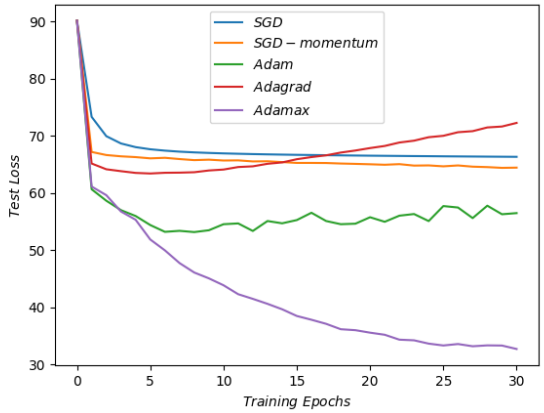

Figure 2.4: Comparison of gradient-based optimization methods used to train a vanilla quine

2.4.2 Is this a quine?

It is hard to quantify how significant it is to reduce the self-replicating loss to a third of its pre-trained value. After all, our goal was to produce a self-replicator, but if the loss we achieved is not close to zero, then it seems that we have not reached our goal. On the other hand, replication mechanisms are rarely perfect. Even in nature, replication mechanisms often contain high levels of noise, sometimes referred to as 'mutation'.

[118] constructed a mathematical framework to calculate the self-replicating quotient of a replicator, which measures the likelihood of a perfect self-replication happening via the replicator's noisy replication mechanism as opposed to it happening by chance. For example, [96] estimate the self-replicating quotient of Penrose Tiling [119] to be below $log2$ and that of animals to be at least 10^{20}. This framework is useful for distinguishing between trivial and non-trivial replicators, but present theoretical understanding of the learning dynamics in a neural network does not suffice to estimate the likelihood of a network being in a certain state.

Another measure we can look at is the *average weight prediction margin*, which is defined as the average absolute difference between the weights and the weight predictions. The pre-training loss of 90.16 corresponds to an average weight prediction margin of 0.067, while the post-training loss of 32.10 corresponds to an average weight prediction margin of 0.040. This suggests we still have significant room for improvement. However, it is worth pointing out that the relatively small pre-training weight prediction margin reflects the fact that modern best practices for the choice of weight initialization and activation function keep the output in the same order of magnitude as the input.

2.4.3 Hill-climbing

Next, we use a hill-climbing algorithm to train the vanilla quine. The algorithm works by iteratively perturbing the parameters of the network with diagonal Gaussian noise and keeping the perturbation if it results in an improvement. This is equivalent to an evolutionary algorithm with a population size of 1. In this case, we do not need the gradients, hence the training process only

requires the forward and not the backward pass, which makes each training epoch computationally cheaper. Nonetheless, it takes around 5000 epochs to find a solution that is on par with that found by classical SGD after 10 epochs. We found that doing hill-climbing on the solution that SGD converged to improves it significantly, but the same does not hold true for the solution that Adamax converged to. This suggests that the solution found by Adamax is already a local optima.

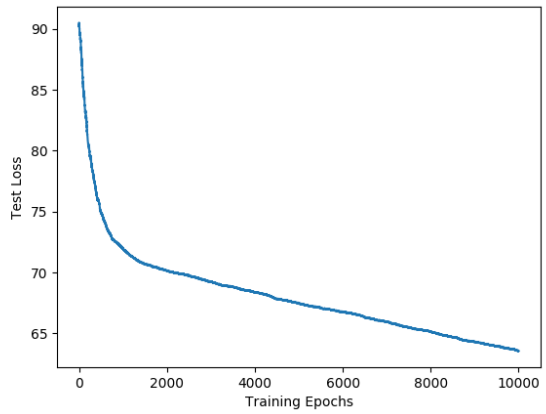

Figure 2.5: Training a vanilla quine via hill-climbing

2.4.4 Generational Replication

Finally, we use generational replication to train the vanilla quine, setting $T = 1$ with Adamax as the optimizer. Each generation epoch is very computationally expensive as it involves as many forward passes as there are parameters in the network to replace its actual weights with its predictions. However, one epoch suffices to reduce the test loss substantially, with the best self-replicating loss of 0.86 found after ten generation epochs. This corresponds to an average weight prediction margin of 0.0065, which is an order of magnitude better than the best solution found previously.

One might wonder if the solution we found via generational replication might be trivial, i.e. if it has learned a solution by zero-ing most weights. Indeed, we find that iteratively injecting the network with its predicted weights has a similar effect as statistical shrinkage. It effectively learns to reduce the self-replicating loss by shrinking the order of magnitudes of the weights, thus

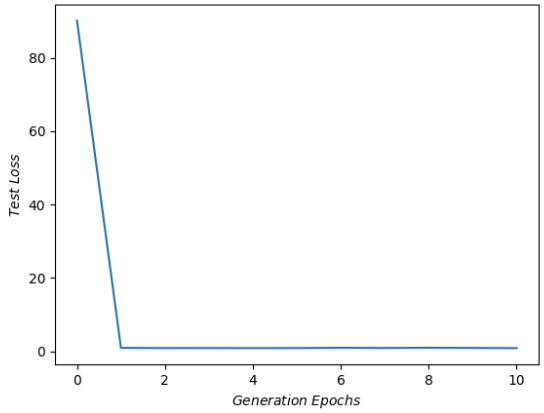

Figure 2.6: Training a vanilla quine via generational replication

creating a small weight prediction margin. Without the optimization step (when $T = 0$), a visual inspection of the network reveals that it rapidly converges to the trivial zero quine. However, with the optimization step, the solution found appears to be non-trivial: the order of magnitude of the weights are in line with what we would observe in a normal neural network.

Figure 2.7 shows a visualization of the solution found by generational replication.

2.4.5 Auxiliary Quine

We trained an auxiliary quine on the MNIST image classification task with Adamax using the default hyperparameter settings on 30 epochs. The quine was also initialized with He init, and the initial loss L_{Aux} prior to any training was 1072.05. We observe in Figure 2.8 that somewhat counter-intuitively, after the initial drop, the auxiliary loss actually increases over time instead of converging. This is due to the network prioritizing the task loss L_{Task} over the self-replicating loss L_{SR} despite the fact that it is being optimized on their sum. The same trend is observed when we repeat the experiment on other gradient-based optimization methods besides Adamax. After 30 epochs, the network achieved an accuracy of 90.41% on the held-out test set, which is comparable to the 96.33% achieved by an identical network whose only objective is MNIST image classification. This shows that self-replication occupies a significant portion of the neural network's

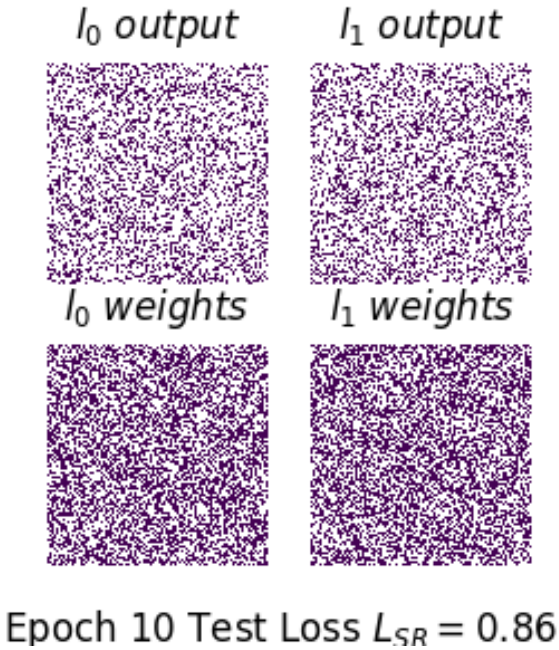

Figure 2.7: Log-normalized illustration of the weights and weight predictions of two hidden layers in a vanilla quine that has been trained with generational replication

capacity, but it is heartening nonetheless that joint optimization of the objectives is possible. If we leave the auxiliary quine running, the task loss eventually converges, while ignoring the exploding self-replicating loss.

This is an interesting finding: it is more difficult for a network that has increased its specialization at a particular task to self-replicate. This suggests that the two objectives are at odds with each other, but that the gradient-based optimization procedure prefers to maximize the network's specialization at solving the MNIST task, even at the expense of a reduction in its ability to self-replicate. (It is not immediately obvious from Figure 2.8, but the first few training epochs reduce the self-replicating loss too.)

There are parallels to be drawn between self-replication in the case of a neural network quine and biological reproduction in nature, as well as specialization at the auxiliary task and survival in nature. The mechanisms for survival are usually aligned with the mechanisms for reproduction, however when they come into conflict with each other, the survival mechanism usually is prioritized at the expense of the reproduction mechanism (except in rare cases like that of the male dark

fishing spider).

Hill-climbing progressed too slowly for us to observe anything meaningful, but we do not expect to observe the same behavior because the algorithm, by definition, does not allow for harmful changes to the overall loss to be made. Generational replication cannot be used in this case, because we require the auxiliary input for each generation and random inputs do not work well.

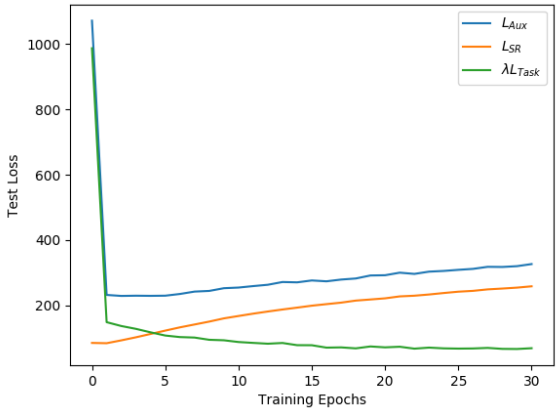

Figure 2.8: Training an auxiliary quine with Adamax

2.5 Conclusion

In this chapter, we have described how to build and train a self-replicating neural network. Specifically, we proposed to treat the problem of self-replication in a neural network as a problem of weight prediction, and devised various encoding and training schemes to solve this problem. This allowed us to create a neural network quine, which akin to a computer program quine, prints its own source code (weights in this case).

We identify three interesting future directions for research. Firstly, we can seek to improve weight prediction by assuming a low-rank matrix factorization for the network's weights as in [101]. Secondly, we can attempt to build neural network quines using more sophisticated models and representations, for example a convolutional neural network quine might be interesting. Thirdly, we can extend the concept of self-replication to universal replication: a neural network

that can replicate other neural networks.

Chapter 3: Agent Embeddings

3.1 Introduction

Many modern artificially intelligent agents are trained with deep reinforcement learning algorithms [11, 120, 121]. But neural networks have long been criticized for being uninterpretable black boxes that cannot be relied upon in safety-critical applications [122, 123].

It is important to note, however, that human brains are uninterpretable as well. For example, we know what a face is, because our brains have evolved to detect facial features, and yet, it is nearly impossible to communicate in words what a face is. This problem is especially acute for patients with severe prosopagnosia, who have to rely on other visual cues to identify their friends and family. In fact, it is also quite difficult to communicate precisely the meaning of words. Try talking to a philosopher or a translator about what otherwise ordinary words might mean, *precisely*, and one can be sure to spark a huge debate.

Nonetheless, it is possible to program a computer to detect faces, by reducing high-dimensional images of faces into low-dimensional vector representations with semantic meaning [124, 14]. It is also possible to perform sophisticated natural language processing tasks by representing words in a high dimensional vocabulary as low-dimensional vectors [125, 126]. Remarkably, these embeddings are amenable to simple linear arithmetic. Take the difference between the latent codes for a face with a mustache and one without a mustache, and one gets something approximating a 'mustache' vector. Famously, [125] showed 'King' - 'Queen' = 'Man' - 'Woman'.

We propose that a similar strategy can be applied to even something as high-dimensional and complicated as a deep reinforcement learning agent. Our aim is to demonstrate that neural network agents can be compressed into low-dimensional vector representations with semantic meaning, which we term *agent embeddings*. In this chapter, we propose to learn agent embeddings by

collecting existing examples of neural network agents, vectorizing their weights, and then learning a generative model over the weight space in a supervised fashion.

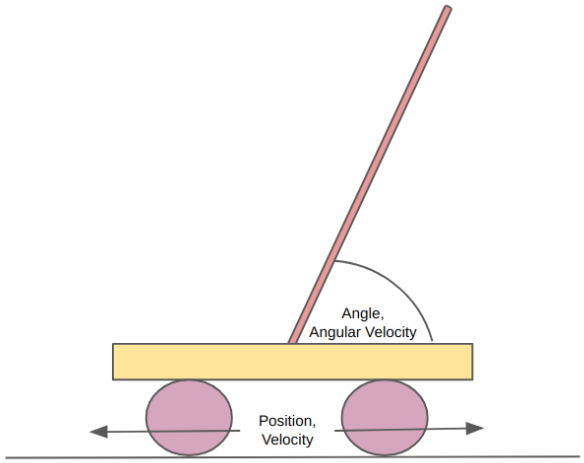

Figure 3.1: Cart-Pole is a game of pole-balancing

3.1.1 Our Contribution

As a proof of concept, we report on a series of experiments involving agent embeddings for policy gradient networks that play Cart-Pole, a game of pole-balancing.

We present three interesting findings:

1. The embedding space learned by the generative model can be used to answer questions of convergent learning [127], i.e. how similar are different neural networks that solve the same task. To our knowledge, we are the first to investigate convergent learning in the context of reinforcement learning agents rather than image classifiers. We extend [127]'s work on convergent learning by proposing a new distance metric for measuring convergence between two neural networks. We observe surprisingly that good pole-balancing networks make different decisions despite learning similar representations, whereas bad pole-balancing networks make similar (bad) decisions while learning dissimilar representations.

2. It has been demonstrated that linear structure between semantic attributes exist in the latent

space of a good generative model in the domain of natural language words [125] and faces [14], among other kinds of data. We show that a similar linear structure can be learned in an embedding space for reinforcement learning agents that can be used to directly control the performance of the policy gradient network generated.

3. We demonstrate that the generative model can be used to recover missing weights in the policy gradient network via a simple and straightforward rejection sampling method. More sophisticated methods of conditional generation are left to future work.

The rest of the chapter is organized as follows: we survey the relevant literature (*Related Work*), introduce the pole-balancing task and describe how we learn agent embeddings for it (*Learning Agent Embeddings for Cart-Pole*), present the above-mentioned findings (*Experimental Results and Discussion*), discuss the shortcomings of our approach (*Limitations of Supervised Generation*), speculate on potential applications (*Potential Applications for AI*), and finally summarize the chapter at the end (*Conclusion*).

3.2 Related Work

There are four areas of research that are related to our work: interpretability, generative modeling, meta-learning, and Bayesian neural networks.

3.2.1 Interpretability

There has been a lot of recent interest in making reinforcement learning agents and policies interpretable. This is especially important in high-stake domains like health care and education. [128] proposed to learn policies in a human-readable programming language, while [129] proposed to learn certificates that provides guarantees on policy outcomes. [130] demonstrated utility in learning embeddings for action traces in path planning. [131]'s work is very similar to ours - they proposed a tool to compare phenotypic differences between solutions found by evolutionary algorithms as a way to explore the geometry of the problem space.

One line of work that has proven useful in increasing our understanding of deep neural network models is that of convergent learning [127], which measures correlations between the weights of different neural networks with the same architecture to determine the similarity of representations learned by these different networks. Convergent learning investigations have hitherto, to our knowledge, only been done on image classifiers, but we extend them to reinforcement learning agents in this chapter.

3.2.2 Generative Modeling

Generative modeling is the technique of learning the underlying data distribution of a training set, with the objective of generating new data points similar to those from the training set. Deep neural networks have been used to build generative models for images [14], audio [132], video [133], natural language sentences [134], DNA sequences [135], and even protein structures [17]. Complex semantic attributes can often be reduced to simple linear vectors and linear arithmetic in the latent spaces of these generative models.

The ultimate (meta) challenge for neural network based generative models is not to generate images or audio, but other neural networks. We use existing networks as meta-training points and use them to train a neural network generator that can produce new pole-balancing networks that do not then need to be further trained with training data from the Cart-Pole simulator. A key advantage of using the same learning framework for both the meta learner and the learner is that this approach could potentially be applied recursively (cue the Singularity).

3.2.3 Meta-Learning

The salient aspect of meta-learning that our work is connected to is the use of neural networks to generate other neural networks. This has been done before in the context of hyperparameter optimization, where one neural network is used to tune the hyperparameters of another neural network [31, 51, 52, 49]. [55] proposed the concept of a HyperNet, a neural network that generates the weights of another neural network with a differentiable function. This allows changes in the

weights of the generated network to be backpropagated to the HyperNet itself. [136] used a neural network to generate its own weights as a way to implement artificial self-replication.

3.2.4 Bayesian Neural Networks

Bayesian neural networks [137] maintain a probabilistic model over the weights of a neural network. In this framework, traditional optimization is viewed as finding the maximum likelihood estimate of the probabilistic model. Posterior inference in this case is typically intractable, but variational approximations can be used [138, 58, 139]. Our work involves learning a generative model over the weights of a neural network using existing examples of networks, which is philosophically akin to learning an 'empirical Bayesian' prior over the weights in a Bayesian neural network.

3.3 Learning Agent Embeddings for Cart-Pole

3.3.1 Supervised Generation

We propose to learn agent embeddings for neural networks using a two-step process we call *Supervised Generation*. First, we train a collection of neural networks of a fixed architecture to solve a particular task. Next, the weights are saved and used as training input to a generative model. This is a supervised method because we are learning the mapping from a latent distribution to the space of neural network weights by feeding input-output pairs to the model. (There are some obvious downsides to *Supervised Generation* as a method of learning agent embeddings. See the *Limitations of Supervised Generation* section for a detailed discussion.)

In this case, we trained a variational autoencoder (*CartPoleGen*) on the parameter space of a small network (*CartPoleNet*) used to play Cart-Pole.

3.3.2 Cart-Pole

Cart-Pole is a pole balancing task introduced by [140] with a modern implementation in the OpenAI Gym [141]. It is also known as the inverted pendulum task and is a classic control problem.

The agent chooses to move left or right at every time step with the objective of preventing the pole from falling over for as long as possible. We chose this task because it is easy - around 200 times easier than MNIST on one measure [142] - and hence can be solved with small neural networks.

3.3.3 CartPoleNet

We devised a simple policy gradient neural network we call *CartPoleNet* with exactly one hidden layer of dimension 30 (see Figure 3.2) using the exponential linear unit [143] as the activation function. We collected 74000 such networks by training them in the Cart-Pole simulator with varying amounts of time, hyperparameters and random seeds for over a week on a cloud computing platform. The 212-dimensional weight vectors belonging to these 74000 networks were then used as the training data for the generative model.

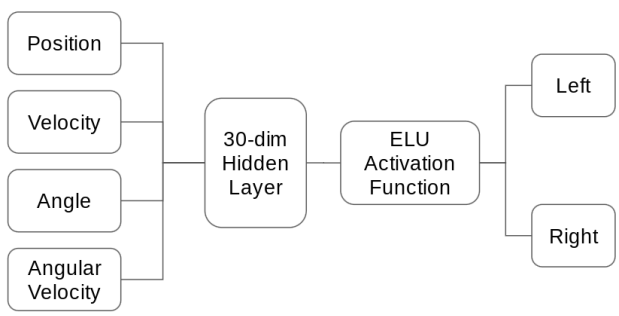

Figure 3.2: Architecture of CartPoleNet

A policy gradient neural network approximates the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \mid s_t = s, a_t = a, \pi\right] \tag{3.1}$$

which is the maximum expected sum of rewards r_i discounted by γ and achieved by a policy $P(a \mid s)$ that makes an action a after observing state s. Cart-Pole assigns a reward of 1 for every step taken, and each episode terminates whenever the pole angle exceeds 12°, the position exceeds the edge of the display, or once the pole has been successfully balanced for more than 200 time steps.

At each epoch, we sample state-action pairs with an epsilon-decreasing policy and store them with their rewards in an experience replay buffer to train the neural network. Note that the neural network only takes state s as input, and its Q-value at action a is represented by the corresponding activation on the last layer. Parametrizing the Q-function with a state-action pair as input is possible but more computationally expensive because it requires $\mid A \mid$ number of forward passes where A is the action space [144].

3.3.4 CartPoleGen

CartPoleGen is a variational autoencoder with a diagonal Gaussian latent space of dimension 32. It contains skip connections (with concatenation not addition) and uses the exponential linear unit as the activation function as in CartPoleNet (see Figure 3.3).

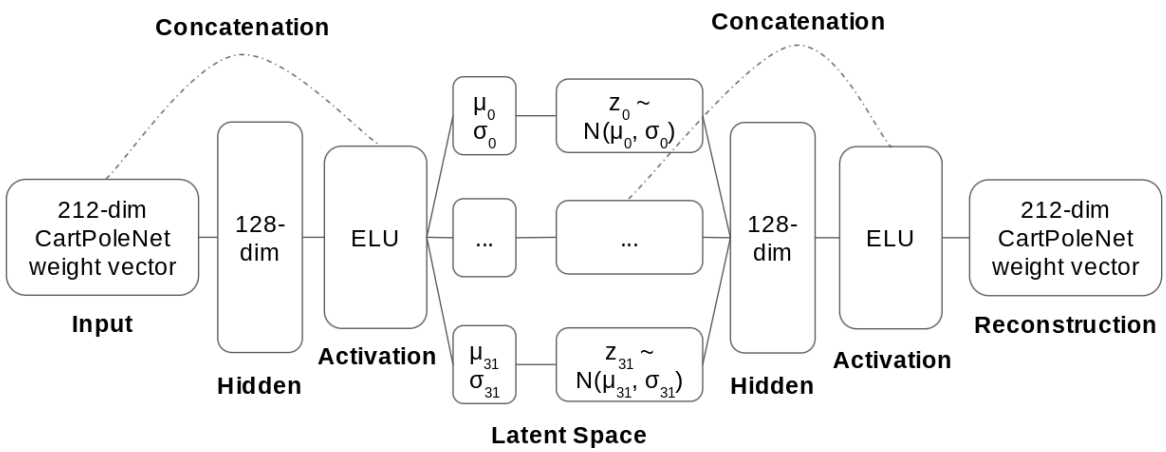

Figure 3.3: Architecture of CartPoleGen

A variational autoencoder [138] is a latent variable model with latent $\mathbf{z}$ and data $\mathbf{x}$. We assume the prior over the latent space to be the spherical Gaussian $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$ and the conditional likelihood $p_\theta(\mathbf{x} \mid \mathbf{z})$ to be Gaussian, which we compute with a neural network decoder parametrized by θ. The true posterior $p(\mathbf{z} \mid \mathbf{x})$ is intractable in this case, but we assume that it can be approximated by a Gaussian with a diagonal covariance structure that we can compute with a neural network encoder $q_\phi(\mathbf{z} \mid \mathbf{x})$ parametrized by ϕ.

Sampling from the posterior involves reparametrizing $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})$ to $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \epsilon$ where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ to allow the gradients to backpropagate through to $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$.

We can train the variational autoencoder by maximizing the variational lower bound on the marginal log likelihood of data point $\mathbf{x}$:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\mathcal{D}_{KL}(q_\phi(\mathbf{z} \mid \mathbf{x}) \mid\mid p(\mathbf{z})) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] \tag{3.2}$$

The Monte Carlo estimator (with latent dimension $k = 32$ and noise mini-batch of size $M = 1$) for equation (2), also known as the SGVB estimator, becomes

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \frac{1}{2}\sum_k (1 + \log \sigma_k^2 - \mu_k^2 - \sigma_k^2) + \frac{1}{M}\sum_{m=1}^{M} \log p_\theta(\mathbf{x} \mid \mathbf{z}^{(m)}) \tag{3.3}$$

Notice that maximizing the above lower bound involves maximizing the model's log-likelihood, which is equivalent to minimizing its negative log-likelihood. Minimizing the negative log-likelihood of a Gaussian model is equivalent to minimizing the mean squared error, which is simply the reconstruction cost in an autoencoder.

3.3.5 Sampling from CartPoleGen

We divided the 74000 networks into four groups depending on the network's *survival time*, which we measure as the average number of steps before the episode terminates across 100 random testing episodes. The survival time is quite a robust measure of CartPoleNet's performance; it varies ±5 at most due to the stochasticity of the Cart-Pole simulator.

We trained CartPoleGen in two settings. The first setting involves training on all 74000 networks, and then measuring the survival time of 200 new samples drawn from the posterior distribution of the variational autoencoder. The second setting involves training a separate CartPoleGen conditioned on each group with a conditional VAE setup [145]. The survival time in the second setting is also measured with 200 new samples drawn from the posterior of the conditional generative model.

The training was conducted using ADAM [115] for 20 epochs with a batch size of 10. The results are summarized in Table 3.1. For comparison, an agent that randomly selects actions lasts on average 22 steps, and an agent that makes the same action at every time step lasts only 9 steps. The Cart-Pole simulation ends once an agent has survived 200 steps, so it is not possible to survive longer than that.

Figure 3.4 shows that the CartPoleGen does not accurately capture the exact distribution of the training data, but that it does offer an approximation to it. Training on better networks tends to lead to better generated networks, with the exception of the $151 - 200$ survival time group. We surmise that this is a consequence of the unimodal variational approximation.

Curiously, CartPoleGen seems to display zero-avoiding rather than zero-forcing behavior, which show that the behavioral properties of neural network agents do not directly match their weight space properties. It is interesting that in some cases, we are able to sample new networks that dramatically outperform the original networks that were in the training set. In the conditional groups, the generated samples typically display much higher variance than is found in the training set, but this does not hold true in the combined setting.

We hypothesize that the approximation gap is partially due to the limitations of the variational autoencoder and can be narrowed with a more expressive generative model. We experimented with various other neural architectures for the encoder and decoder, but did not manage to find significant improvements. In fact, the architecture of CartPoleGen presented here approximates a similar distribution when the encoder and decoder are trained with linear layers.

We also experimented with using GANs [77, 14] as the generative model for CartPoleGen, but did not manage to successfully train them. In our experiments, the discriminator was not able to provide a good teaching signal to the generator because it managed to rapidly distinguish between the fake and real samples.

Table 3.1: Sampling new instances of CartPoleNet

Group	Trainset Size	(Mean, Std) of Survival Time in Trainset	(Mean, Std) of Survival Time in Generated Samples
1 – 50 steps	25608	21.8, 11.5	11.0, 9.7
51 – 100 steps	9400	69.7, 14.2	77.3, 46.5
101 – 150 steps	10103	132.6, 13.1	127.0, 55.3
151 – 200 steps	28889	184.9, 16.3	116.4, 58.6
Combined	74000	106.7, 73.3	136.7, 42.8

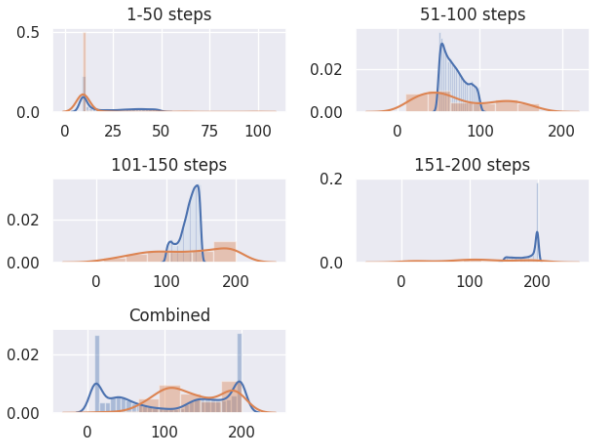

Figure 3.4: The figures are plotted as histograms, with KDE curves fitted on them. The x-axis denotes the survival time, and the y-axis denotes the percentage of networks with that survival time. The figures in blue represent the networks from the trainset, while the figures in orange represent the sampled networks.

3.4 Experimental Results and Discussion

In this section, we perform three experiments using the agent embeddings learned by CartPole-Gen in the previous section. These experiments involve (1) deciding if different CartPoleNets of similar ability learn similar representations, (2) exploring the latent space learned by CartPoleGen, and (3) repairing missing weights in a CartPoleNet.

3.4.1 Convergent Learning

[127] posed the question of convergent learning: do different neural networks learn the same representations? In the case of convolutional neural networks used as image classifiers, they found

that shallow representations that resemble Gabor-like edge detectors are reliably learned, while more semantic representations sometimes differ.

Success is usually not an accident. Prima facie, for a given complex task, it seems like there can be a million ways to fail it, but only a handful of ways to successfully solve it. We hypothesize this to be the case for Cart-Pole, but found surprisingly that the reverse was true.

[127] measured activations on a reference set of images from the ImageNet Large Scale Visual Recognition Challenge 2012 dataset [8], and calculated the correlation of such activations between pairs of convolutional neural networks. For CartPoleNets, the inputs are environment states in Cart-Pole, so we had to first collect a reference set of 10000 diverse states in the Cart-Pole simulator before computing CartPoleNet activations on them.

We follow the same methodology as [127] with the slight modification that we use the absolute value of the activations. This is because we use ELUs in CartPoleNet which have important negative activations that ReLU-based networks do not.

$$\text{Mean} : \mu_i = \mathbb{E}[|X_i|] \tag{3.4}$$

$$\text{Std} : \sigma_i = \sqrt{\mathbb{E}[(|X_i| - \mu_i)^2]} \tag{3.5}$$

$$\text{Corr} : \rho_{i,j} = \mathbb{E}[(|X_i| - \mu_i)(|X_j| - \mu_j)]/\sigma_i\sigma_j \tag{3.6}$$

The correlation between activations of a pair of networks can then be used to pair units from the first network with units from the second. In a bipartite matching, we assign each pair by matching units with the highest correlation, taking them out of consideration, and repeating the process until all the units have been paired. Hence, each unit belongs to exactly one pair. This can be done efficiently with the Hopcroft-Kraft algorithm [146]. In a semi-matching, we sequentially assign each unit i from the first network using the unit j from the second network with the highest correlation $\rho_{i,j}$. It is thus possible that some units will belong to multiple pairs, while others will

not get paired at all.

Two networks are in some sense equivalent if we can arrive at one network by permuting the ordering of the units of the other. The *convergence distance (CD)* between two networks can hence be quantitatively measured as the distance between the bipartite matching and the semi-matching (see Equation 3.7). There is exactly one bipartite matching of maximum cardinality, but multiple possible semi-matchings depending on the order of assignment. We compute the convergence distance using the *canonical* semi-matching, defined as the semi-matching performed in descending order from the most highly correlated to the least highly correlated pair in the bipartite matching.

$$\mathrm{CD}(\mathrm{Net1}, \mathrm{Net2}) = \sum_i \rho_{i,\mathrm{Bipartite}(i)} - \rho_{i,\mathrm{Semi}(i)} \tag{3.7}$$

We sampled ten networks with survival time ∼191 (from the conditional CartPoleGen trained on the 151-200 survival time group) and ten networks with survival time ∼29 (from the conditional CartPoleGen trained on the 0-50 survival time group) to represent good and bad networks respectively. Randomly selecting actions results in a survival time of 22, so 29 represents a bad network that is nonetheless acting better than random. The average all-pairs convergence distance in the good group and in the bad group are then computed, with the results summarized in Table 3.2. We visualize the convergence distances in the hidden and output layer between selected pairs of CartPoleNets in Figures 3.5 and 3.6 respectively.

Table 3.2: Convergence of Good vs. Bad Networks (Higher CDs correspond to divergence, while lower CDs correspond to convergence)

Group	Survival Time	Mean, Std CD (Hidden)	Mean, Std CD (Output)
Good	191	2.75, 1.96	**0.32**, 0.49
Bad	29	**3.13**, 1.7	0.09, 0.11

The data suggests that for the task of Cart-Pole that there are more ways to be successful than to be bad. In other words, given a random state in the environment, the good networks can diverge in their decision to move left or right to balance the pole, but the bad networks uniformly make

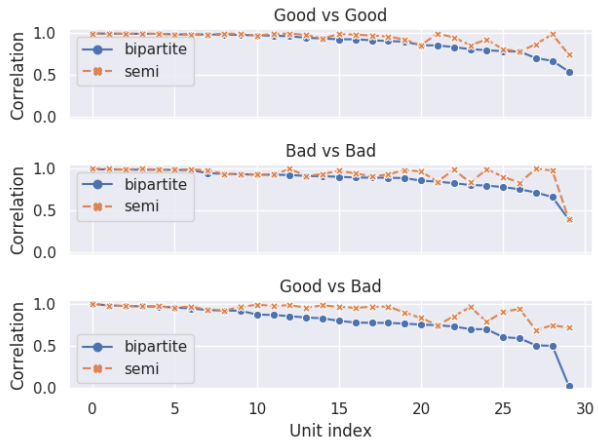

Figure 3.5: The figure shows correlations between *hidden* activations of a pair of good CartPoleNets, a pair of bad CartPoleNets, and a pair with one good and one bad CartPoleNet. For the networks used in this figure, the convergence distances between the pairs are 1.51, 1.75 and 3.91 respectively.

the wrong decision. Surprisingly also, despite the good networks displaying divergence in their actions, they pick up on more convergent (good) representations.

It is quite interesting that there are more ways to balance a pole successfully than poorly, but the skills needed for the different paths to success are similar. We hypothesize that this is because the order of actions might be less important than the overall composition of the two actions. Consider a sequence of four actions. *{Left, Right, Left, Right}* would be highly negatively correlated with *{Right, Left, Right, Left}* but on average, they might produce the same outcome of keeping the pole balanced. On the other hand, *{Left, Left, Left, Left}* is highly correlated with *{Left, Left, Left, Left}* and they both cause the pole to quickly lose its balance.

3.4.2 Exploring the Latent Space

The latent space in CartPoleGen gives us semantic information about the kinds of networks that can be generated. We selected pairs of agent embeddings and sampled 20 new embeddings from $\alpha = 0.0$ to $\alpha = 1.5$ where α represents the coefficient of linear interpolation between the pair of embeddings. $0 < \alpha < 1$ represents interpolation, while $\alpha > 1$ represents extrapolation. The

Figure 3.6: The figure shows correlations between *output* activations of a pair of good CartPoleNets, a pair of bad CartPoleNets, and a pair with one good and one bad CartPoleNet. For the networks used in this figure, the convergence distances between the pairs are 0.32, 0.07 and 0.28 respectively.

results are summarized in Figure 3.7.

The top left graph represents a pair of agent embeddings with a hidden CD of 1.82, the top right 12.5, the bottom left 2.13, and the bottom right 2.77. We observe that linearly interpolating within the latent space of CartPoleGen is not the same as simply interpolating within the weight space of CartPoleNet, given that CartPoleGen is non-linear in nature. In many cases, moving from a worse agent embedding to a better one tracks a similar improvement in survival time, as is the case in the top left and bottom right graphs. Furthermore, extrapolation results in a performance boost, up to a point.

However, we also observed many cases where interpolation resulted in agent embeddings whose network performed far worse or far better than the two embeddings used as endpoints for the interpolation. Interestingly, when the interpolated embeddings performed far better, it is often the case that the hidden CDs of the networks used for the two endpoint embeddings is fairly large. In the case of the top right graph, the hidden CD is in fact a few standard deviations above the mean.

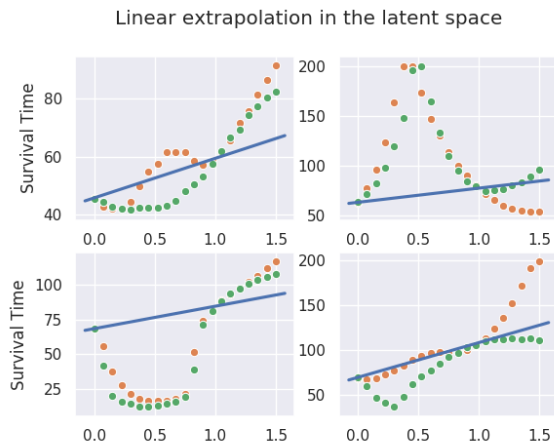

Figure 3.7: The x axis represents the coefficient of interpolation α, while the y axis represents the survival time of the sampled networks. The orange dots represent networks sampled from interpolating within the latent space, while the green dots represent networks interpolated within the weight space with the same coefficient of interpolation. The blue line is a straight line drawn from the survival time of the network sampled from the first agent embedding to the survival time of the network sampled from the second agent embedding.

3.4.3 Repairing Missing Weights

The generative model can be used to repair CartPoleNets with missing weights. We propose a simple rejection sampling based method (see Algorithm 3) to continuously sample new CartPoleNets from the model until suitable candidates are found to fill out the missing weights. We experiment with two possible criteria that can be used to pick the candidate.

$$W = \text{Existing} \cup \text{Missing} \tag{3.8}$$

$$C = \text{Candidate} \tag{3.9}$$

The *Missing Criterion* (see Equation 3.10) picks out the candidate who is most similar to the damaged CartPoleNet when we are only comparing the existing weights.

$$C^* = \arg\min_{C} \sum_{i \in \text{Existing}} (W_i - C_i)^2 \tag{3.10}$$

The *Whole Criterion* (see Equation 3.11) picks out the candidate who is most similar to the damaged CartPoleNet. This biases the selection towards finding candidates with tiny weights in the missing space.

$$C^* = \arg\min_{C} \sum_{i \in W} (W_i - C_i)^2 \tag{3.11}$$

Algorithm 3: Rejection sampling based method to repair missing weights in a CartPoleNet W. Let $ST()$ represent the survival time of a network.

```
γ = 200
k = 10
ε = 5
Sample γ networks from CartPoleGen
Pick k best candidates C* using a Criterion
for i ∈ [k] do
    if | ST(C_i) − ST(W) | < ε then
        return Success, C_i
    end
end
return Failure, ∅
```

We can probe the limits of our generative model for the task of weight repair by determining how much degradation can be reversed with a fixed computational budget (i.e. γ and k are fixed). To investigate this, we fix a given CartPoleNet, degrade it at a fixed level (i.e. zero out a fixed fraction of the weights at random), and repair it using the rejection sampling based algorithm proposed. The results are summarized in Figure 3.8.

We observe that the two criteria seem to perform similarly, with *Whole Criterion* performing slightly better, and we managed to successfully recover the network at some levels of degradation. While we do not recover the network completely (below the acceptable threshold of 5) in many cases, it is hopeful to note that there is partial recovery (the difference in survival times is at most 15). It is also interesting that it is possible to recover the network at complete degradation; this suggests perhaps that CartPoleGen has memorized this network.

The scheme described here can also be straightforwardly applied to the task of repairing (or verifying) corrupted weights instead of missing weights. We note that rejection sampling is an in-

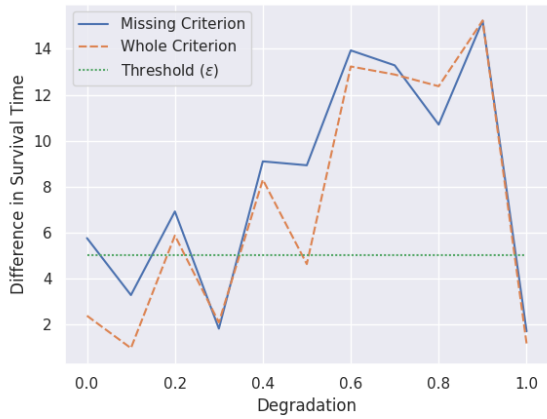

Figure 3.8: The figure shows the performance of the two criteria (in terms of the difference in survival time between the original network and the recovered network) used to repair missing weights at ten different levels of degradation. The threshold ε represents what we consider a successful level of recovery, so all the points below the threshold represent successful reversal of degradation.

efficient method of doing weight repair, and more sophisticated methods of conditional generation should be used if efficiency is of concern.

3.5 Limitations of Supervised Generation

We note three main limitations of the *Supervised Generation* method in learning agent embeddings.

3.5.1 High Sample Complexity

One of the primary drawbacks of the *Supervised Generation* method is the two-step process needed to first collect the data then train a generative model on it. This requires training a very large number of networks to provide the generative model with data. Figure 3.9 shows progressively worse approximations when we decrease the number of sampled networks by an order of magnitude.

In principle, an agent embedding does not have to be learned in this manner. For example, it might be possible to do *Online Generation* where a generative model learns to generate new

networks on-the-fly with an online algorithm. *Online Generation* will probably be more sample efficient.

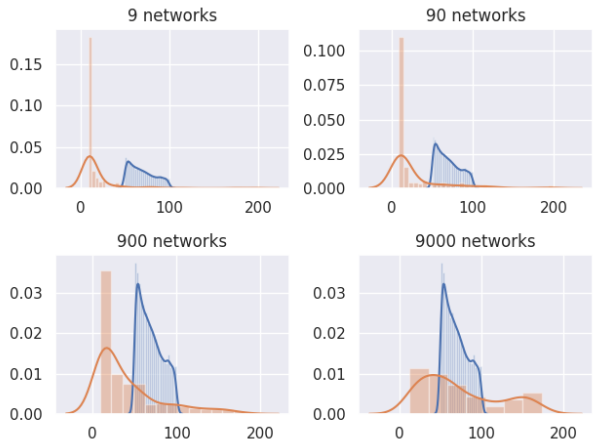

Figure 3.9: If we try to train the distribution in the 51-100 survival time group referred to in Figure 3.4 with fewer number of samples, we get worse approximations.

3.5.2 Subpar Model Performance

CartPoleGen does not approximate the training distribution very well (see Figure 3.4). This might potentially be fixed with a better generative model that also has access to online training data. For example, Bayesian HyperNetworks [58] might be a promising candidate.

3.5.3 Scaling Issues

We tried using a variational autoencoder to learn a 21840-dimensional weight vector for a small neural network that does MNIST image classification. Reinforcement learning agents that process images with CNNs would most likely contain weights at this order of magnitude at minimum. We trained it on a dataset of 10000 networks each with >95% accuracy, but none of the sampled networks managed to perform with >30% accuracy on a test set.

It might be difficult to scale the *Supervised Generation* method to large networks, even with significant advances made in generative modeling techniques. This is because even state of the art supervised generative models typically deal with data of much lower dimensions (<1000). A

notable exception is WaveNet [132], but it deals with audio data which is relatively smooth and can tolerate high amounts of error, while the weights of a neural network are very discontinuous and are not robust to small amounts of additive noise.

3.6 Potential Applications for AI

The ultimate challenge for neural network based generative systems is not generating images, sounds, or videos. The ultimate challenge is the generation of other neural networks. Learning agent embeddings is therefore a very difficult goal to accomplish, but we outline several potential applications for AI in general.

- AI systems powered by neural networks are often criticized for being uninterpretable. Agent embeddings provide us with a tool to gain insight into its internal workings and the space of possible solutions, which we have demonstrated with the task of pole balancing in this chapter.

- The generative model can be conditioned to prevent it from generating networks that have undesirable properties like biases or security vulnerabilities. This is helpful for improving the fairness and security of AI systems. We showed how CartPoleGen can be used to repair weights in a network for example, which increases the data integrity of the system.

- It is helpful for an AI system to be able to generate worker AIs in a modular fashion. Each worker AI can be represented with its own agent embedding, and the generative model can be a factory that delivers a custom solution conditioned on the task given.

- Reinforcement learning agents perform better when they have access to a model of their environment. We think they will also perform better in multi-agent systems when they have access to compressed embeddings of other agents.

3.7 Conclusion

In this chapter, we presented the concept of agent embeddings, a way to reduce a reinforcement learning agent into a small, meaningful vector representation. As a proof of concept, we trained an autoencoder neural network CartPoleGen on a large number of policy gradient neural networks collected to solve the pole-balancing task Cart-Pole. We showcased three interesting experimental findings with CartPoleGen and described the challenges of the *Supervised Generation* method.

Chapter 4: Hypernetwork Initialization

4.1 Introduction

Meta-learning describes a broad family of techniques in machine learning that deals with the problem of learning to learn. An emerging branch of meta-learning involves the use of hypernetworks, which are meta neural networks that generate the weights of a main neural network to solve a given task in an end-to-end differentiable manner. Hypernetworks were originally introduced by [55] as a way to induce weight-sharing and achieve model compression by training the same meta network to learn the weights belonging to different layers in the main network. Since then, hypernetworks have found numerous applications including but not limited to: weight pruning [56], neural architecture search [41, 57], Bayesian neural networks [58, 59, 60, 61, 62], multi-task learning [63, 64, 65, 66, 67], continual learning [68], generative models [69, 70], ensemble learning [71], hyperparameter optimization [72], and adversarial defense [73].

Despite the intensified study of applications of hypernetworks, the problem of optimizing them to this day remains significantly understudied. In fact, even the problem of initializing hypernetworks has not been studied. Given the lack of principled approaches, prior work in the area is mostly limited to ad-hoc approaches based on trial and error (c.f. Section 4.3). For example, it is common to initialize the weights of a hypernetwork by sampling a "small" random number. Nonetheless, these ad-hoc methods do lead to successful hypernetwork training primarily due to the use of the Adam optimizer [115], which has the desirable property of being invariant to the scale of the gradients. However, even Adam will not work if the loss diverges (i.e. overflow) at initialization, which will happen in sufficiently big models. The normalization of badly scaled gradients also results in noisy training dynamics where the loss function suffers from bigger fluctuations during training compared to vanilla stochastic gradient descent (SGD). [147, 148] showed

that while adaptive optimizers like Adam may exhibit lower training error, they fail to generalize as well to the test set as non-adaptive gradient methods. Moreover, Adam incurs a computational overhead and requires 3X the amount of memory for the gradients compared to vanilla SGD.

Small random number sampling is reminiscent of early neural network research [149] before the advent of classical weight initialization methods like Xavier init [81] and Kaiming init [82]. Since then, a big lesson learned by the neural network optimization community is that architecture specific initialization schemes are important to the robust training of deep networks, as shown recently in the case of residual networks [150]. In fact, weight initialization for hypernetworks was recognized as an outstanding open problem by prior work [62] that had questioned the suitability of classical initialization methods for hypernetworks.

Our results We show that when classical methods are used to initialize the weights of hypernetworks, they fail to produce mainnet weights in the correct scale, leading to exploding activations and losses. This is because classical network weights transform one layer's activations into another, while hypernet weights have the added function of transforming the hypernet's activations into the mainnet's weights. Our solution is to develop principled techniques for weight initialization in hypernetworks based on variance analysis. The hypernet case poses unique challenges. For example, in contrast to variance analysis for classical networks, the case for hypernetworks can be asymmetrical between the forward and backward pass. The asymmetry arises when the gradient flow from the mainnet into the hypernet is affected by the biases, whereas in general, this does not occur for gradient flow in the mainnet. This underscores again why architecture specific initialization schemes are essential. We show both theoretically and experimentally that our methods produce hypernet weights in the correct scale. Proper initialization mitigates exploding activations and gradients or the need to depend on Adam. Our experiments reveal that it leads to more stable mainnet weights, lower training loss, and faster convergence.

Section 4.2 briefly covers the relevant technical preliminaries, and Section 4.3 reviews problems with the ad-hoc methods currently deployed by hypernetwork practitioners. We derive novel weight initialization formulae for hypernetworks in Section 4.4, empirically evaluate our proposed

methods in Section 4.5, and finally conclude in Section 4.6.

4.2 Preliminaries

Definition. *A **hypernetwork** is a meta neural network H with its own parameters ϕ that generates the weights of a main network θ from some embedding e in a differentiable manner: $\theta = H_\phi(e)$. Unlike a classical network, in a hypernetwork, the weights of the main network are not model parameters. Thus the gradients $\Delta\theta$ have to be further backpropagated to the weights of the hypernetwork $\Delta\phi$, which is then trained via gradient descent $\phi_{t+1} = \phi_t - \lambda\Delta\phi_t$.*

This fundamental difference suggests that conventional knowledge about neural networks may not apply directly to hypernetworks and novel ways of thinking about weight initialization, optimization dynamics and architecture design for hypernetworks are sorely needed.

4.2.1 Ricci Calculus

We propose the use of Ricci calculus, as opposed to the more commonly used matrix calculus, as a suitable mathematical language for thinking about hypernetworks. Ricci calculus is useful because it allows us to reason about the derivatives of higher-order tensors with notational ease. For readers not familiar with the index-based notation of Ricci calculus, please refer to [151] for a good introduction to the topic written from a machine learning perspective.

For a general nth-order tensor $T^{i_1,\ldots,i_k,\ldots,i_n}$, we use d_{i_k} to refer to the dimension of the index set that i_k is drawn from. We include explicit summations where the relevant expressions might be ambiguous, and use Einstein summation convention otherwise. We use square brackets to denote different layers for added clarity, so for example $W[t]$ denotes the t-th weight layer.

4.2.2 Xavier Initialization

[81] derived weight initialization formulae for a feedforward neural network by conducting a variance analysis over activations and gradients. For a linear layer $y^i = W^i_j x^j + b^i$, suppose we make

the following **Xavier Assumptions** at initialization: **(1)** The W^i_j, x^j, and b^i are all independent of each other. **(2)** $\forall i, j : \mathbb{E}[W^i_j] = 0$. **(3)** $\forall j : \mathbb{E}[x^j] = 0$. **(4)** $\forall i : b^i = 0$.

Then, $\mathbb{E}[y^i] = 0$ and $\mathrm{Var}(y^i) = \mathrm{d}_j \mathrm{Var}(W^i_j)\mathrm{Var}(x^j)$. To keep the variance of the output and input activations the same, i.e. $\mathrm{Var}(y^i) = \mathrm{Var}(x^j)$, we have to sample W^i_j from a distribution whose variance is equal to the reciprocal of the fan-in: $\mathrm{Var}(W^i_j) = \frac{1}{\mathrm{d}_j}$.

If analogous assumptions hold for the backward pass, then to keep the variance of the output and input gradients the same, we have to sample W^i_j from a distribution whose variance is equal to the reciprocal of the fan-out: $\mathrm{Var}(W^i_j) = \frac{1}{\mathrm{d}_i}$.

Thus, the forward pass and backward pass result in symmetrical formulae. [81] proposed an initialization based on their harmonic mean: $\mathrm{Var}(W^i_j) = \frac{2}{\mathrm{d}_j + \mathrm{d}_i}$.

In general, a feedforward network is non-linear, so these assumptions are strictly invalid. But odd activation functions with unit derivative at 0 results in a roughly linear regime at initialization.

4.2.3 Kaiming Initialization

[82] extended [81]'s analysis by looking at the case of ReLU activation functions, i.e. $y^i = W^i_j \,\mathrm{ReLU}(x^j) + b^i$. We can write $z^j = \mathrm{ReLU}(x^j)$ to get

$$\mathrm{Var}(y^i) = \sum_j \mathbb{E}[(z^j)^2]\mathrm{Var}(W^i_j) = \sum_j \frac{1}{2}\,\mathbb{E}[(x^j)^2]\mathrm{Var}(W^i_j) = \frac{1}{2}\,\mathrm{d}_j \mathrm{Var}(W^i_j)\mathrm{Var}(x^j).$$

This results in an extra factor of 2 in the variance formula. W^i_j have to be symmetric around 0 to enforce Xavier Assumption 3 as the activations and gradients propagate through the layers. [82] argued that both the forward or backward version of the formula can be adopted, since the activations or gradients will only be scaled by a depth-independent factor. For convolutional layers, we have to further divide the variance by the size of the receptive field.

'Xavier init' and 'Kaiming init' are terms that are sometimes used interchangeably. Where there might be confusion, we will refer to the forward version as fan-in init, the backward version as fan-out init, and the harmonic mean version as harmonic init.

4.3 Review of Current Methods

In the seminal [55] paper, the authors identified two distinct classes of hypernetworks: dynamic (for recurrent networks) and static (for convolutional networks). They proposed Orthogonal init [152] for the dynamic class, but omitted discussion of initialization for the static class. The static class has since proven to be the dominant variant, covering all kinds of non-recurrent networks (not just convolutional), and thus will be the central object of our investigation.

Through an extensive literature and code review, we found that hypernet practitioners mostly depend on the Adam optimizer, which is invariant to and normalizes the scale of gradients, for training and resort to one of four weight initialization methods:

M1 Xavier or Kaiming init (as found in [60, 153, 66, 68]).

M2 Small random values (as found in [58, 72]).

M3 Kaiming init, but with the output layer scaled by $\frac{1}{10}$ (as found in [59]).

M4 Kaiming init, but with the hypernet embedding set to be a suitably scaled constant (as found in [67]).

M1 uses classical neural network initialization methods to initialize hypernetworks. This fails to produce weights for the main network in the correct scale. Consider the following illustrative example of a one-layer linear hypernet generating a linear mainnet with $T + 1$ layers, given embeddings sampled from a standard normal distribution and weights sampled entry-wise from a zero-mean distribution. We leave the biases out for now, and assume the input data $x[1]$ is standardized.

$$\begin{gathered} x[t+1]^{i_{t+1}} = W[t]_{i_t}^{i_{t+1}} x[t]^{i_t}, \qquad W[t]_{i_t}^{i_{t+1}} = H[t]_{i_t k_t}^{i_{t+1}} e[t]^{k_t}, \qquad 1 \leq t \leq T. \\ \mathrm{Var}(x[T+1]^{i_{t+1}}) = \mathrm{Var}(x[1]^{i_1}) \prod_{t=1}^{T} \mathrm{d}_{i_t} \mathrm{Var}(W[t]_{i_t}^{i_{t+1}}) = \mathrm{Var}(x[1]^{i_1}) \prod_{t=1}^{T} \mathrm{d}_{i_t} \mathrm{d}_{k_t} \mathrm{Var}(H[t]_{i_t k_t}^{i_{t+1}}). \end{gathered} \tag{4.1}$$

In this case, if the variance of the weights in the hypernet $\mathrm{Var}(H[t]_{i_t k_t}^{i_{t+1}})$ is equal to the reciprocal of

the fan-in d_{k_t}, then the variance of the activations $\mathrm{Var}(x[T+1]^{i_{t+1}}) = \prod_{t=1}^{T} \mathrm{d}_{i_t}$ explodes. If it is equal to the reciprocal of the fan-out $\mathrm{d}_{i_t}\mathrm{d}_{i_{t+1}}$, then the activation variance $\mathrm{Var}(x[T+1]^{i_{t+1}}) = \prod_{t=1}^{T} \frac{\mathrm{d}_{k_t}}{\mathrm{d}_{i_t+1}}$ is likely to vanish, since the size of the embedding vector is typically small relatively to the width of the mainnet weight layer being generated.

Where the fan-in is of a different scale than the fan-out, the harmonic mean has a scale close to that of the smaller number. Therefore, the fan-in, fan-out, and harmonic variants of Xavier and Kaiming init will all result in activations and gradients that scale exponentially with the depth of the mainnet.

M2 and M3 introduce additional hyperparameters into the model, and the ad-hoc manner in which they work is reminiscent of pre deep learning neural network research, before the introduction of classical initialization methods like Xavier and Kaiming init. This ad-hoc manner is not only inelegant and consumes more compute, but will likely fail for deeper and more complex hypernetworks.

M4 proposes to set the embeddings $e[t]^{k_t}$ to a suitable constant ($\mathrm{d}_{i_t}^{-1/2}$ in this case), such that both $W[t]_{i_t}^{i_{t+1}}$ and $H[t]_{i_t k_t}^{i_{t+1}}$ can seem to be initialized with the same variance as Kaiming init. This ensures that the variance of the activations in the mainnet are preserved through the layers, but the restrictions on the embeddings might not be desirable in many applications.

Luckily, the fix appears simple — set $\mathrm{Var}(H[t]_{i_t k_t}^{i_{t+1}}) = \frac{1}{\mathrm{d}_{i_t}\mathrm{d}_{k_t}}$. This results in the variance of the generated weights in the mainnet $\mathrm{Var}(W[t]_{i_t}^{i_{t+1}}) = \frac{1}{\mathrm{d}_{i_t}}$ resembling conventional neural networks initialized with fan-in init. This suggests a general hypernet weight initialization strategy: initialize the weights of the hypernet such that the mainnet weights approximate classical neural network initialization. We elaborate on and generalize this intuition in Section 4.4.

4.4 Hyperfan Initialization

Most hypernetwork architectures use a linear output layer so that gradients can pass from the mainnet into the hypernet directly without any non-linearities. We make use of this fact in developing methods called hyperfan-in init and hyperfan-out init for hypernetwork weight initialization

based on the principle of variance analysis.

4.4.1 Hyperfan-in

Proposition. *Suppose a hypernetwork comprises a linear output layer. Then, the variance between the input and output activations of a linear layer in the mainnet $y^i = W^i_j x^j + b^i$ can be preserved using fan-in init in the hypernetwork with appropriately scaled output layers.*

Case 1. The hypernet generates the weights but not the biases of the mainnet. The bias in the mainnet is initialized to zero. We can write the weight generation in the form $W^i_j = H^i_{jk} h(e)^k + \beta^i_j$ where h computes all but the last layer of the hypernet and (H, β) form the output layer. We make the following **Hyperfan Assumptions** at initialization: **(1)** Xavier assumptions hold for all the layers in the hypernet. **(2)** The H^i_{jk}, $h(e)^k$, β^i_j, x^j, and b^i are all independent of each other. **(3)** $\forall i, j, k : \mathbb{E}[H^i_{jk}] = 0$. **(4)** $\mathbb{E}[x^j] = 0$. **(5)** $\forall i : b^i = 0$.

Use fan-in init to initialize the weights for h. Then, $\mathrm{Var}(h(e)^k) = \mathrm{Var}(e^l)$. If we initialize H with the formula $\mathrm{Var}(H^i_{jk}) = \frac{1}{\mathrm{d}_j \mathrm{d}_k \mathrm{Var}(e^l)}$ and β with zeros, we arrive at $\mathrm{Var}(W^i_j) = \frac{1}{\mathrm{d}_j}$, which is the formula for fan-in init in the mainnet. The Hyperfan assumptions imply the Xavier assumptions hold in the mainnet, thus preserving the input and output activations.

$$\begin{aligned}\mathrm{Var}(y^i) &= \sum_j \mathrm{Var}(W^i_j)\mathrm{Var}(x^j) = \sum_j \sum_k \mathrm{Var}(H^i_{jk})\mathrm{Var}(h(e)^k)\mathrm{Var}(x^j) \\ &= \sum_j \sum_k \frac{1}{\mathrm{d}_j \mathrm{d}_k \mathrm{Var}(e^l)} \mathrm{Var}(e^l)\mathrm{Var}(x^j) = \mathrm{Var}(x^j).\end{aligned} \tag{4.2}$$

Case 2. The hypernet generates both the weights and biases of the mainnet. We can write the weight and bias generation in the form $W^i_j = H^i_{jk} h(e[1])^k + \beta^i_j$ and $b^i = G^i_l g(e[2])^l + \gamma^i$ respectively, where h and g compute all but the last layer of the hypernet, and (H, β) and (G, γ) form the output layers. We modify Hyperfan Assumption 2 so it includes G^i_l, $g(e[2])^l$, and γ^i, and further assume $\mathrm{Var}(x^j) = 1$, which holds at initialization with the common practice of data standardization.

Use fan-in init to initialize the weights for h and g. Then, $\mathrm{Var}(h(e[1])^k) = \mathrm{Var}(e[1]^m)$ and

$\mathrm{Var}(g(e[2])^l) = \mathrm{Var}(e[2]^n)$. If we initialize H with the formula $\mathrm{Var}(H^i_{jk}) = \frac{1}{2\mathrm{d}_j\mathrm{d}_k\mathrm{Var}(e[1]^m)}$, G with the formula $\mathrm{Var}(G^i_l) = \frac{1}{2\mathrm{d}_l\mathrm{Var}(e[2]^n)}$, and β, γ with zeros, then the input and output activations in the mainnet can be preserved.

$$\begin{aligned}
\mathrm{Var}(y^i) &= \sum_j \left[\mathrm{Var}(W^i_j)\mathrm{Var}(x^j)\right] + \mathrm{Var}(b^i) \\
&= \sum_j \left[\sum_k \mathrm{Var}(H^i_{jk})\mathrm{Var}(h(e[1])^k)\mathrm{Var}(x^j)\right] + \sum_l \mathrm{Var}(G^i_l)\mathrm{Var}(g(e[2])^l) \\
&= \sum_j \left[\sum_k \frac{1}{2\mathrm{d}_j\mathrm{d}_k\mathrm{Var}(e[1]^m)}\mathrm{Var}(e[1]^m)\mathrm{Var}(x^j)\right] + \sum_l \frac{1}{2\mathrm{d}_l\mathrm{Var}(e[2]^n)}\mathrm{Var}(e[2]^n) \\
&= \frac{1}{2}\mathrm{Var}(x^j) + \frac{1}{2} = \mathrm{Var}(x^j).
\end{aligned} \tag{4.3}$$

If we initialize G^i_j to zeros, then its contribution to the variance will increase during training, causing exploding activations in the mainnet. Hence, we prefer to introduce a factor of $1/2$ to divide the variance between the weight and bias generation, where the variance of each component is allowed to either decrease or increase during training. This becomes a problem if the variance of the activations in the mainnet deviates too far away from 1, but we found that it works well in practice.

4.4.2 Hyperfan-out

Case 1. The hypernet generates the weights but not the biases of the mainnet. A similar derivation can be done for the backward pass using analogous assumptions on gradients flowing

$$\begin{aligned}
\text{in the mainnet:} \quad & \frac{\partial L}{\partial x[t]^{i_t}} = \frac{\partial L}{\partial x[t+1]^{i_{t+1}}} W[t]^{i_{t+1}}_{i_t}, \\
\text{through mainnet weights:} \quad & \frac{\partial L}{\partial W[t]^{i_{t+1}}_{i_t}} = \frac{\partial L}{\partial x[t+1]^{i_{t+1}}} x[t]^{i_t}, \frac{\partial L}{\partial h[t](e)^{k_t}} = \frac{\partial L}{\partial W[t]^{i_{t+1}}_{i_t}} H[t]^{i_{t+1}}_{i_t k_t}, \\
\text{and through mainnet biases:} \quad & \frac{\partial L}{\partial b[t]^{i_{t+1}}} = \frac{\partial L}{\partial x[t+1]^{i_{t+1}}}, \frac{\partial L}{\partial g[t](e)^{l_t}} = \frac{\partial L}{\partial b[t]^{i_{t+1}}} G[t]^{i_{t+1}}_{l_t}.
\end{aligned} \tag{4.4}$$

If we initialize the output layer H with the analogous hyperfan-out formula $\mathrm{Var}(H[t]^{i_{t+1}}_{i_t k_t}) = \frac{1}{\mathrm{d}_{i_{t+1}} \mathrm{d}_{k_t} \mathrm{Var}(e^{k_t})}$ and the rest of the hypernet with fan-in init, then we can preserve input and output gradients on the mainnet: $\mathrm{Var}(\frac{\partial L}{\partial x[t]^{i_t}}) = \mathrm{Var}(\frac{\partial L}{\partial x[t+1]^{i_{t+1}}})$. However, note that the gradients will shrink when flowing from the mainnet to the hypernet: $\mathrm{Var}(\frac{\partial L}{\partial h[t](e)^{k_t}}) = \frac{\mathrm{d}_{i_t}}{\mathrm{d}_{k_t} \mathrm{Var}(e^{k_t})} \mathrm{Var}(\frac{\partial L}{\partial W[t]^{i_{t+1}}_{i_t}})$, and scaled by a depth-independent factor due to the use of fan-in rather than fan-out init.

Case 2. The hypernet generates both the weights and biases of the mainnet. In the classical case, the forward version (fan-in init) and the backward version (fan-out init) are symmetrical. This remains true for hypernets if they only generated the weights of the mainnet. However, if they were to also generate the biases, then the symmetry no longer holds, since the biases do not affect the gradient flow in the mainnet but they do so for the hypernet (c.f. Equation 4.4). Nevertheless, we can initialize G so that it helps hyperfan-out init preserve activation variance on the forward pass as much as possible (keeping the assumption that $\mathrm{Var}(x^j) = 1$ as before):

$$
\begin{aligned}
\mathrm{Var}(y^i) &= \sum_j \left[\mathrm{Var}(W^i_j x^j)\right] + \mathrm{Var}(b^i) \\
&= \mathrm{d}_j \mathrm{d}_k \mathrm{Var}(e[1]^m) \mathrm{Var}(H[\text{hyperfan-out}]^i_{jk}) \mathrm{Var}(x^j) + \mathrm{d}_l \mathrm{Var}(e[2]^n) \mathrm{Var}(G^i_l) \\
&= \mathrm{d}_j \mathrm{d}_k \mathrm{Var}(e[1]^m) \mathrm{Var}(H[\text{hyperfan-in}]^i_{jk}) \mathrm{Var}(x^j) \qquad (4.5)
\end{aligned}
$$

Plugging in the formulae for Hyperfan-in and Hyperfan-out from above, we get

$$
\implies \mathrm{Var}(G^i_l) = \frac{(1 - \frac{\mathrm{d}_j}{\mathrm{d}_i})}{\mathrm{d}_l \mathrm{Var}(e[2]^n)}.
$$

We summarize the variance formulae for hyperfan-in and hyperfan-out init in Table 4.1. It is not uncommon to re-use the same hypernet to generate different parts of the mainnet, as was originally done in [55]. We discuss this case in more detail in Appendix Section A.1.

4.5 Experiments

We evaluated our proposed methods on four sets of experiments involving different use cases of hypernetworks: feedforward networks, continual learning, convolutional networks, and Bayesian neural networks. In all cases, we optimize with vanilla SGD and sample from the uniform distri-

Table 4.1: Hyperfan-in and Hyperfan-out Variance Formulae for $W^i_j = H^i_{jk} h(e[1])^k + \beta^i_j$. If $y^i = \text{ReLU}(W^i_j x^j + b^i)$, then $\mathbb{1}_{\text{ReLU}} = 1$, else if $y^i = W^i_j x^j + b^i$, then $\mathbb{1}_{\text{ReLU}} = 0$. If $b^i = G^i_l g(e[2])^l + \gamma^i$, then $\mathbb{1}_{\text{HBias}} = 1$, else if $b^i = 0$, then $\mathbb{1}_{\text{HBias}} = 0$. We initialize h and g with fan-in init, and $\beta^i_j, \gamma^i = 0$. For convolutional layers, we have to further divide $\text{Var}(H^i_{jk})$ by the size of the receptive field. Uniform init: $X \sim \mathcal{U}(-\sqrt{3\text{Var}(X)}, \sqrt{3\text{Var}(X)})$. Normal init: $X \sim \mathcal{N}(0, \text{Var}(X))$.

Initialization	**Variance Formula**	**Initialization**	**Variance Formula**
Hyperfan-in	$\text{Var}(H^i_{jk}) = \frac{2^{\mathbb{1}_{\text{ReLU}}}}{2^{\mathbb{1}_{\text{HBias}}} \text{d}_j \text{d}_k \text{Var}(e[1]^m)}$	Hyperfan-out	$\text{Var}(H^i_{jk}) = \frac{2^{\mathbb{1}_{\text{ReLU}}}}{\text{d}_i \text{d}_k \text{Var}(e[1]^m)}$
Hyperfan-in	$\text{Var}(G^i_l) = \frac{2^{\mathbb{1}_{\text{ReLU}}}}{2\text{d}_l \text{Var}(e[2]^n)}$	Hyperfan-out	$\text{Var}(G^i_l) = \max(\frac{2^{\mathbb{1}_{\text{ReLU}}}(1-\frac{\text{d}_j}{\text{d}_i})}{\text{d}_l \text{Var}(e[2]^n)}, 0)$

bution according to the variance formula given by the init method. More experimental details can be found in Appendix Section A.2.

4.5.1 Feedforward Networks on MNIST

As an illustrative first experiment, we train a feedforward network with five hidden layers (500 hidden units), a hyperbolic tangent activation function, and a softmax output layer, on MNIST across four different settings: (1) a classical network with Xavier init, (2) a hypernet with Xavier init that generates the weights of the mainnet, (3) a hypernet with hyperfan-in init that generates the weights of the mainnet, (4) and a hypernet with hyperfan-out init that generates the weights of the mainnet.

The use of hyperfan init methods on a hypernetwork reproduces mainnet weights similar to those that have been trained from Xavier init on a classical network, while the use of Xavier init on a hypernetwork causes exploding activations right at the beginning of training (see Figure 4.1). Observe in Figure 4.2 that when the hypernetwork is initialized in the proper scale, the magnitude of generated weights stabilizes quickly. This in turn leads to a more stable training regime, as seen in Figure 4.3. More visualizations of the activations and gradients of both the mainnet and hypernet can be viewed in Appendix Section A.2.1. Qualitatively similar observations were made when we replaced the activation function with ReLU and Xavier with Kaiming init, with Kaiming

init leading to even bigger activations at initialization.

Suppose now the hypernet generates both the weights and biases of the mainnet instead of just the weights. We found that this architectural change leads the hyperfan init methods to take more time (but still less than Xavier init), to generate stable mainnet weights (c.f. Figure A.19 in the Appendix).

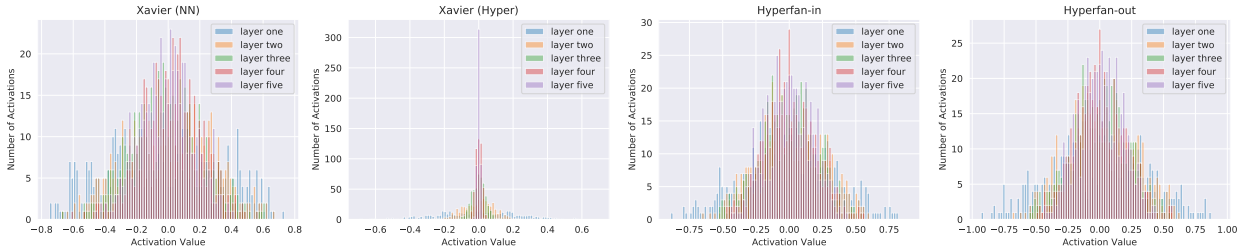

Figure 4.1: Mainnet Activations before the Start of Training on MNIST.

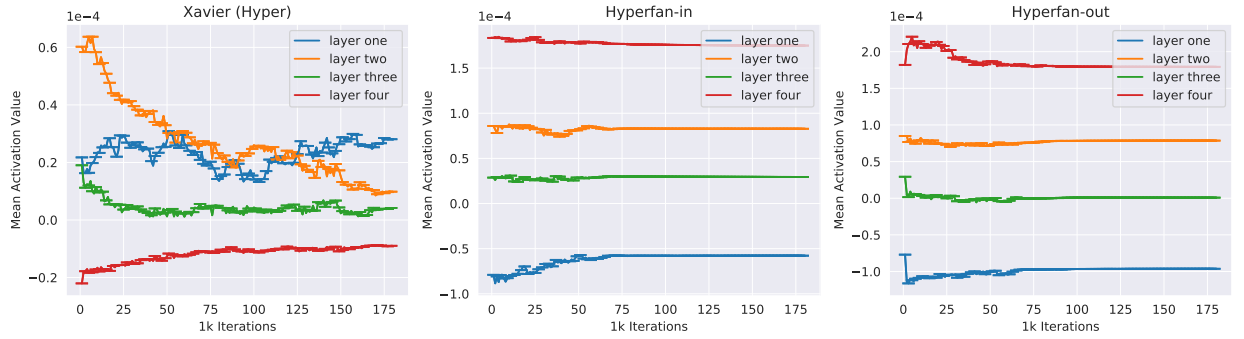

Figure 4.2: Evolution of Hypernet Output Layer Activations during Training on MNIST. Xavier init results in unstable mainnet weights throughout training, while hyperfan-in and hyperfan-out init result in mainnet weights that stabilize quickly.

4.5.2 Continual Learning on Regression Tasks

Continual learning solves the problem of learning tasks in sequence without forgetting prior tasks. [68] used a hypernetwork to learn embeddings for each task as a way to efficiently regularize the training process to prevent catastrophic forgetting. We compare different initialization schemes on their hypernetwork implementation, which generates the weights and biases of a ReLU mainnet with two hidden layers to solve a sequence of three regression tasks.

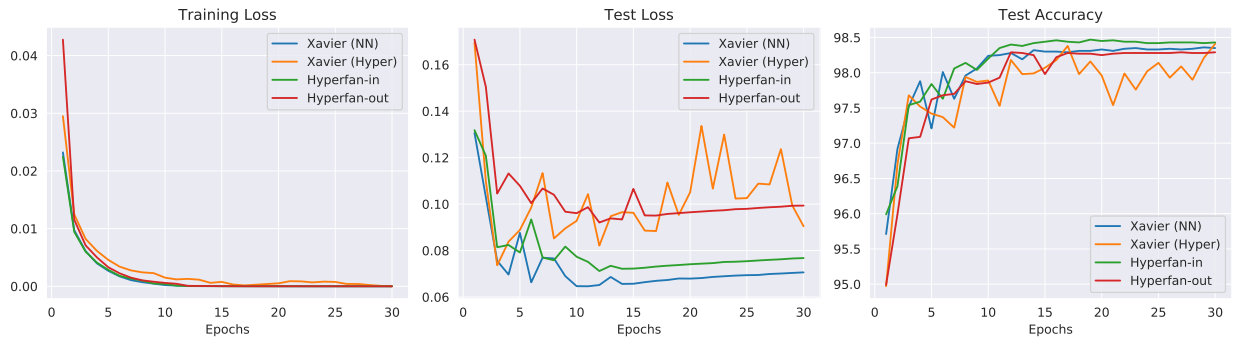

Figure 4.3: Loss and Test Accuracy Plots on MNIST.

In Figure 4.4, we plot the training loss averaged over 15 different runs, with the shaded area showing the standard error. We observe that the hyperfan methods produce smaller training losses at initialization and during training, eventually converging to a smaller loss for each task.

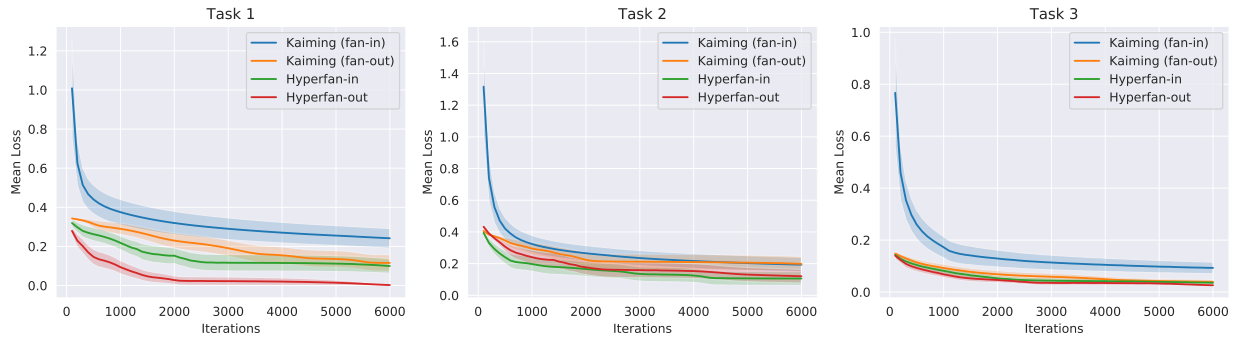

Figure 4.4: Continual Learning Loss on a Sequence of Regression Tasks.

4.5.3 Convolutional Networks on CIFAR-10

[55] applied a hypernetwork on a convolutional network for image classification on CIFAR-10. We note that our initialization methods do not handle residual connections, which were in their chosen mainnet architecture and are important topics for future study. Instead, we implemented their hypernetwork architecture on a mainnet with the All Convolutional Net architecture [154] that is composed of convolutional layers and ReLU activation functions.

After searching through a dense grid of learning rates, we failed to enable the fan-in version of Kaiming init to train even with very small learning rates. The fan-out version managed to begin

delayed training, starting from around epoch 270 (see Figure 4.5). By contrast, both hyperfan-in and hyperfan-out init led to successful training immediately. This shows a good init can make it possible to successfully train models that would have otherwise been unamenable to training on a bad init.

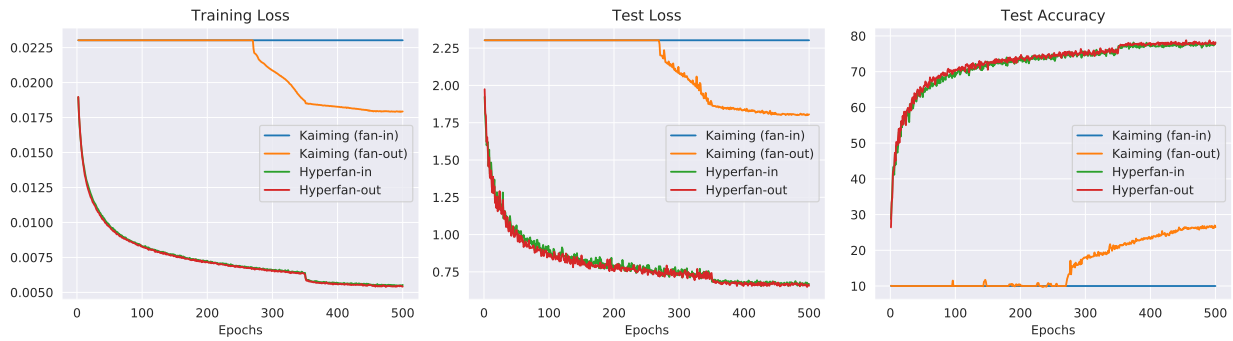

Figure 4.5: Loss and Test Accuracy Plots on CIFAR-10.

4.5.4 Bayesian Neural Networks on ImageNet

Bayesian neural networks improve model calibration and provide uncertainty estimation, which guard against the pitfalls of overconfident networks. [59] developed a Bayesian neural network by using a hypernetwork to simulate an expressive prior distribution. We trained a similar hypernetwork by applying [59]'s methods on ImageNet, but differed in our choice of MobileNet [155] as a mainnet architecture that does not have residual connections.

In the work of [59], it was noticed that even with the use of batch normalization in the mainnet, classical initialization approaches still led to diverging losses (due to exploding activations, c.f. Section 4.3). We observe similar results in our experiment (see Figure 4.6) — the fan-in version of Kaiming init, which is the default initialization in popular deep learning libraries like PyTorch and Chainer, resulted in substantially higher initial losses and led to slower training than the hyperfan methods. We found that the observation still stands even when the last layer of the mainnet is not generated by the hypernet. This shows that while batch normalization helps, it is not the solution for a bad init that causes exploding activations. Our approach solves this problem in a principled

way, and is preferable to the trial-and-error based heuristics that [59] had to resort to in order to train their model.

Surprisingly, the fan-out version of Kaiming init led to similar results as the hyperfan methods, suggesting that batch normalization might be sufficient to correct the bad initializations that result in vanishing activations. That being said, hypernet practitioners should not expect batch normalization to be the panacea for problems caused by bad initialization, especially in memory-constrained scenarios. In a Bayesian neural network application (especially in hypernet architectures without relaxed weight-sharing), the blowup in the number of parameters limits the use of big batch sizes, which is essential to the performance of batch normalization [156]. For example, in this experiment, our hypernet model requires 32 times as many parameters as a classical MobileNet.

To the best of our knowledge, the interaction between batch normalization and initialization is not well-understood, even in the classical case, and thus, our findings prompt an interesting direction for future research.

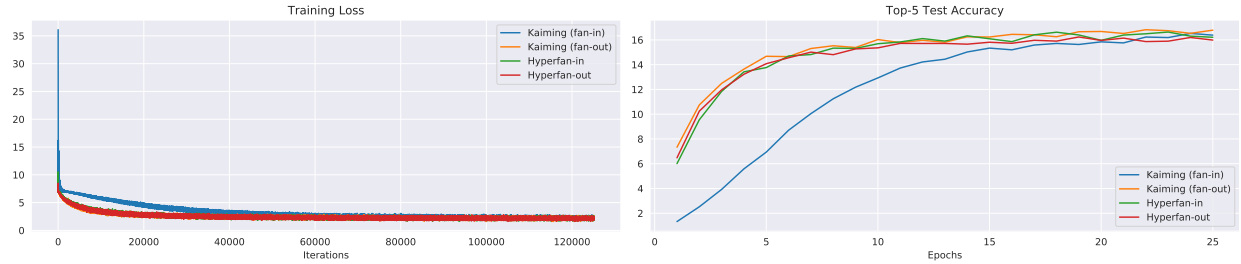

Figure 4.6: Loss and Test Accuracy Plots on ImageNet.

In all our experiments, hyperfan-in and hyperfan-out both led to successful hypernetwork training with SGD. We did not find a good reason to prefer one over the other (similar to [82]'s observation in the classical case for fan-in and fan-out init).

4.6 Conclusion

For a long time, the promise of deep nets to learn rich representations of the world was left unfulfilled due to the inability to train these models. The discovery of greedy layer-wise pre-training [157, 158] and later, Xavier and Kaiming init, as weight initialization strategies to enable such

training was a pivotal achievement that kickstarted the deep learning revolution. This underscores the importance of model initialization as a fundamental step in learning complex representations.

In this work, we developed the first principled weight initialization methods for hypernetworks, a rapidly growing branch of meta-learning. We hope our work will spur momentum towards the development of principled techniques for building and training hypernetworks, and eventually lead to significant progress in learning meta representations. Other non-hypernetwork methods of neural network generation [28, 159] can also be improved by considering whether their generated weights result in exploding activations and how to avoid that if so.

Chapter 5: Hypernetwork Optimization

5.1 Introduction

A hypernetwork is a meta neural network parametrized by ϕ that generates a main neural network with weights θ to minimize a given task loss $\mathcal{L}$. Unlike a conventional neural network, θ are not model parameters, and the gradients that backpropagate to them have to be further backpropagated to ϕ for a gradient descent update $\phi := \phi - \alpha \nabla_\phi \mathcal{L}$.

Hypernetworks were introduced by [55] for the purpose of model compression, and they have since been employed in a wide range of other applications including Bayesian deep learning, multi-task learning, continual learning, and more. The object of this chapter is to study hypernetworks in an artificial life context.

Specifically, we define and study a special class of hypernetworks called <u>hypergenerative networks</u> that act as replicators — they take a given neural network's parameters θ as input and outputs parameters equal to them $\theta_{\text{auto}} = \theta$. We make three contributions: i) A catalog of different update rules to the main network generated by gradient descent on the hypernetwork arising from simple hypernetwork architectures, ii) A proof that a generalized form for some of these update rules, when applied recursively and used to train the hypernetwork, result in the same update rule for the main network, and iii) Experimental verification that these update rules can be used to successfully train large neural networks with comparable levels of accuracy as gradient descent.

5.2 Catalog of hypergenerative Networks

We limit our study of hypergenerative networks to <u>exact</u> replicators $f = I$. A simple way to ensure exact replication is to initialize ϕ at every training step so that the hypergenerative property $\theta_{\text{auto}} = f_\phi(\theta) = \theta$ holds, where θ is considered a constant and gradient descent is used to update

ϕ. This gives rise to different update rules on the main network depending on the hypernetwork's architecture.

Rule 0: Ordinary gradient descent falls into a special case where the hypernetwork consists of just a bias term $b \in \mathbb{R}^n$. The hypergenerative property holds when $b = 0$.

$$\begin{aligned} \phi &= \{b\}, \\ \theta_{\text{auto}} &= \theta + b, \\ \theta'_{\text{auto}} &:= \theta + b - \alpha \nabla_b \mathcal{L} \\ &= \theta_{\text{auto}} - \alpha \nabla_{\theta_{\text{auto}}} \mathcal{L}. \end{aligned} \tag{5.1}$$

Rule 1: Add a multiplicative factor $w \in \mathbb{R}^n$ to Rule 0. The hypergenerative property holds when $w = 1, b = 0$.

$$\begin{aligned} \phi &= \{w, b\}, \\ \theta_{\text{auto}} &= w \odot \theta + b, \\ \theta'_{\text{auto}} &:= (w - \alpha \nabla_w \mathcal{L}) \odot \theta + b - \alpha \nabla_b \mathcal{L} \\ &= \theta_{\text{auto}} - \alpha (1 + \theta_{\text{auto}}^2) \odot \nabla_{\theta_{\text{auto}}} \mathcal{L}. \end{aligned} \tag{5.2}$$

Rule 2: Now consider a normalization of θ. The hypergenerative property holds when $w = ||\theta||_2, b = 0$.

$$\begin{aligned} \phi &= \{w, b\}, \\ \theta_{\text{auto}} &= w \odot \frac{\theta}{||\theta||_2} + b, \\ \theta'_{\text{auto}} &:= (w - \alpha \nabla_w \mathcal{L}) \odot \frac{\theta}{||\theta||_2} + b - \alpha \nabla_b \mathcal{L} \\ &= \theta_{\text{auto}} - \alpha (1 + \frac{\theta_{\text{auto}}}{||\theta_{\text{auto}}||_2} \odot \frac{\theta_{\text{auto}}}{||\theta_{\text{auto}}||_2}) \odot \nabla_{\theta_{\text{auto}}} \mathcal{L}. \end{aligned} \tag{5.3}$$

Rule 3: Consider a weight matrix $W \in \mathbb{R}^{n \text{x} n}$ instead in Rule 1. The hypergenerative property holds

when $W = I, b = 0$.

$$\begin{aligned}
\phi &= \{W, b\}, \\
\theta_{\text{auto}} &= W\theta + b, \\
\theta'_{\text{auto}} &:= (W - \alpha\nabla_W \mathcal{L})\theta + b - \alpha\nabla_b \mathcal{L} \\
&= \theta_{\text{auto}} - \alpha(1 + ||\theta_{\text{auto}}||_2)\nabla_{\theta_{\text{auto}}}\mathcal{L}.
\end{aligned} \tag{5.4}$$

Rule 4: Consider a linear autoencoder with tied weights. The hypergenerative property holds for orthogonal W (i.e. $W^T W = I$). Let $K = \theta\ \nabla_{\theta_{\text{auto}}}\mathcal{L}^T + \nabla_{\theta_{\text{auto}}}\mathcal{L}\ \theta^T$.[1]

$$\begin{aligned}
\phi &= \{W\}, \\
\theta_{\text{auto}} &= W^T W\theta, \\
\theta'_{\text{auto}} &:= (W - \alpha\nabla_W \mathcal{L})^T (W - \alpha\nabla_W \mathcal{L})\theta \\
&= \theta_{\text{auto}} - \alpha(2 - \alpha K)K\theta_{\text{auto}}.
\end{aligned} \tag{5.5}$$

Rule 5: Add bias terms to Rule 4. The hypergenerative property holds for orthogonal W, $b_1 = 0$, $b_2 = 0$.

$$\begin{aligned}
\phi &= \{W, b_1, b_2\}, \\
\theta_{\text{auto}} &= W^T(W\theta + b_1) + b_2, \\
\theta'_{\text{auto}} &:= (W - \alpha\nabla_W \mathcal{L})^T \left[(W - \alpha\nabla_W \mathcal{L})\theta + b_1 - \alpha\nabla_{b_1}\mathcal{L}\right] \\
&\quad + b_2 - \alpha\nabla_{b_2}\mathcal{L} \\
&= \theta_{\text{auto}} - \alpha(2 - \alpha K)(\nabla_{\theta_{\text{auto}}}\mathcal{L} + K\theta_{\text{auto}}).
\end{aligned} \tag{5.6}$$

[1] An implementation detail is that we have to use vector products instead of storing K explicitly, since it uses quadratic memory.

5.3 Stability under Hypergeneration

Above, we derived update rules to the main network that were a result of gradient descent updates to the hypernetwork. But these update rules can themselves be used to train the hypernetwork. We say that a hypernet is stable under hypergeneration if an update rule applied to the hypernet results in an equivalent update rule to the mainnet.

Rule 0 is trivially stable. Below, we state a generalized form for Rules 1-3 that are stable, where C is some constant.

$$
\begin{aligned}
&\textbf{Generalized Rule 1:}\\
&\theta_{\text{auto}} := \theta_{\text{auto}} - \alpha(1 + C\theta_{\text{auto}}^2) \odot \nabla_{\theta_{\text{auto}}}\mathcal{L}.\\
&\textbf{Generalized Rule 2:}\\
&\theta_{\text{auto}} := \theta_{\text{auto}} - \alpha(1 + C\frac{\theta_{\text{auto}}}{||\theta_{\text{auto}}||_2} \odot \frac{\theta_{\text{auto}}}{||\theta_{\text{auto}}||_2}) \odot \nabla_{\theta_{\text{auto}}}\mathcal{L}.\\
&\textbf{Generalized Rule 3:}\\
&\theta_{\text{auto}} := \theta_{\text{auto}} - \alpha(1 + C||\theta_{\text{auto}}||_2)\nabla_{\theta_{\text{auto}}}\mathcal{L}.
\end{aligned}
\tag{5.7}
$$

Generalized Rule 1 can be proved to be stable like so.

$$
\begin{aligned}
\theta'_{\text{auto}} &:= \big(w - \alpha(1 + Cw^2)\nabla_w\mathcal{L}\big) \odot \theta + b - \alpha(1 + Cb^2)\nabla_b\mathcal{L}\\
&= \theta_{\text{auto}} - \alpha\big(1 + (1 + C)\theta_{\text{auto}}^2\big) \odot \nabla_{\theta_{\text{auto}}}\mathcal{L}.\\
&= \theta_{\text{auto}} - \alpha(1 + C'\theta_{\text{auto}}^2) \odot \nabla_{\theta_{\text{auto}}}\mathcal{L}.
\end{aligned}
\tag{5.8}
$$

Similar derivations can be done for Generalized Rules 2 and 3. It is an open question if Rules 4 and 5 have a generalized form that is stable as well.

5.4 Experiments

We trained a ResNet18 [160] using the Adam optimizer [115] for 40 epochs on the CIFAR-10 dataset. The main finding is that all these alternative update rules except Rule 2 displayed similar

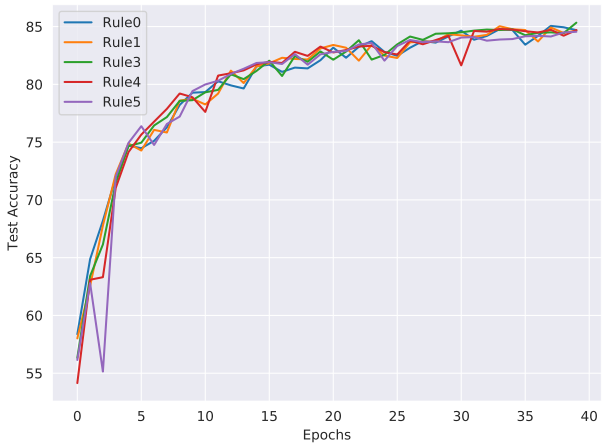

Figure 5.1: Test Accuracy of ResNet18 on CIFAR-10 for Hypergenerative Networks.

test accuracy to standard gradient descent (Rule 0), despite them not minimizing the task loss directly. This suggests that the reparametrization resulting from the use of these rules can be an efficient way to model artifical populations of neural agents created from hypernetworks, since the overhead compute used by these rules is negligible compared to maintaining full hypernetwork parameters. Rule 2 did not result in successful training because of numerical errors from division by zero.

5.5 Conclusion

We did a preliminary study of a special class of replicator hypernetworks called hypergenerative networks in this chapter. Future work involves extending our analysis to noisy replicators and more complex hypernet architectures, as well as using them in applications that involve modeling populations of neural network agents.

Chapter 6: Gradient-Based Meta-Learning

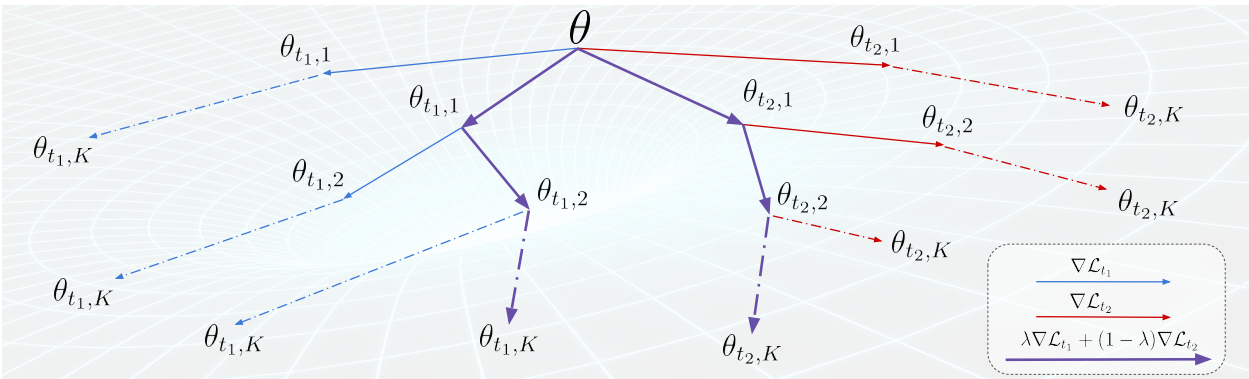

Figure 6.1: The arrows represent gradient steps taken within the inner loop of meta-learning, with $\theta_{t,k}$ denoting the version of the model after k training steps on task t. There are no across-task interactions in the inner loop, causing task-specific over-fitting. This is especially so at the beginning of meta-training, before a good initialization θ has been meta-learned. The inner loop learning process can be regularized with gradients shared from related tasks.

6.1 Introduction

Despite the recent triumphs of deep supervised learning in fields as disparate as computer vision [161], speech processing [162], and computational biology [163], much human expertise and massive amounts of data are necessary to engineer the learning algorithms involved. Devising an optimal learning algorithm for the problem at hand is usually not trivial since different domains require different inductive biases [164]. The manual search for better learning algorithms significantly increases the time needed to successfully train and deploy a machine learning model.

Meta-learning is a sub-field of machine learning that endeavors to rise to these challenges by applying machine learning itself to the (meta) task of generating better machine learning algorithms [43, 47, 165]. There are many approaches to meta-learning including but not limited to: using reinforcement learning to search for optimal neural architectures [31], learning meta networks that

generate other networks [55], augmenting neural networks with an external memory [166, 167], learning metric-based representations for different tasks [168, 169, 170], and learning parametric weight update rules [42, 45, 78].

Gradient-based meta-learning is a special case of the parametric weight update rule approach where the rule is differentiable and its parameters can be learned using a gradient-based optimizer. The weight update rule is itself a learning algorithm and is commonly referred to as the inner loop, by contrast with the optimizer which is the outer loop. Because the inner loop is differentiable, we can backpropagate gradients from the outer loop through the inner loop to update the parameters of the inner loop learning rule. In this chapter, we will refer to gradient-based meta learning as just meta-learning and the term 'meta-learn' refers to updates made to the inner loop learner by the outer loop learner.

In meta-learning, each inner loop learner learns by sampling from data points within a given task and suffering a test loss. The outer loop then meta-learns by sampling from tasks in a given meta-training distribution and combines the test losses of several inner loop learners to suffer a meta-test loss. After having seen sufficiently many tasks, the goal of meta-learning is to produce a general learning rule that can learn from a new unseen task. Intuitively, meta-learning can be seen as a way to transfer learn [171] at scale, and if the inner loop learns to quickly learn, it can be very effective at few-shot learning [75].

However, like conventional machine learning, meta-learning algorithms can be prone to the risk of over-fitting. Unlike conventional learning, over-fitting in meta-learning can occur at both the level of the outer and the inner loop. Much prior work has dealt with the outer loop over-fitting to tasks in the meta-training distribution [172, 173, 174, 175], but little attention has been paid towards the inner loop over-fitting to task-specific training data points.

During the initial phase of meta-training, the scarce number of data points in each task, especially for few-shot learning setups, inevitably causes the over-fitting of comparatively much bigger neural network models. To counter this, meta-learning methods meta-learn the initialized weights of the inner loop learner as a parameter [78]. By pooling information across different tasks in the

meta-training distribution using the outer loop, the initialization eventually picks up task invariant information and gravitates towards a good basin of attraction that reduces the tendency for the inner loop learner to over-fit [176]. This means that unlike conventional or outer loop over-fitting, inner loop over-fitting does not always pose a problem to the generalization ability of the meta-learner.

Nevertheless, limiting the interaction between tasks to take place only through the outer loop iteration significantly slows down the convergence of meta-learning. At the start of meta-training, the over-fitting of the inner loop learners causes them to suffer high test losses. Their parameters correspondingly fail to encode task specific information, reducing the signal available to the meta-test loss and thus, the outer loop. This problem is sustained until the model progresses towards more meaningful solutions in the inner loop, causing a significant number of wasteful initial model updates in the outer loop. In meta-learning, this issue is further exacerbated by the inordinate computational expense of a model update, which scales with the number of data points within each task, the number of tasks, the number of operations used by each inner loop learner, and the ultimate need to backpropagate through all of that.

Our Contribution We propose an inner loop regularization mechanism inspired by multi-task learning [177, 178] called gradient sharing. Historically, multi-task learning was a predominant approach to leveraging multiple related tasks to learn task-invariant information. Despite this common objective, the rapid development of meta-learning has occurred independently from the vast multi-task learning literature. **The surprising insight from our work is that the two fields complement each other in a synergistic way.** On one hand, sharing information across tasks in the inner loop via multi-task learning significantly reduces over-fitting. On the other hand, the outer loop can be recruited to meta-learn extra parameters so as to avoid the traditional pitfalls of multi-task learning like imbalanced task combinations. Our proposed method works by sharing gradient information obtained from both previously encountered and concurrent tasks, and scales their contribution with meta-learned parameters. Through extensive experiments on two popular few-shot image classification datasets, we show that gradient sharing accelerates the meta-training process by up to 134%, and enables meta-learning that is robust to bigger inner loop learning rates while

achieving comparable or better meta-test performance. Accelerating meta-training is a key step towards unleashing its full potential, empowering practitioners to use more complex inner loop learners that would have otherwise been intractable.

The rest of the chapter is organized as follows: Section 6.2 briefly reviews gradient-based meta-learning and related work. Section 6.3 explores the complementary relationship between the well-established field of multi-task learning and the recent body of work in gradient-based meta-learning. We introduce the gradient sharing algorithm in Section 6.4, experimentally evaluate and discuss our results in Section 6.5, and finally conclude our findings in Section 6.6.

6.2 Review of Gradient-Based Meta-Learning

Gradient-based meta-learning consists of a meta-training and a meta-testing phase, both containing batches of conventional supervised learning tasks. For such a task t drawn from task distribution $\mathcal{T}$, we denote its training loss by $\mathcal{L}_t^{\text{train}}$ and its test loss by $\mathcal{L}_t^{\text{test}}$. The goal of meta-learning is to learn to learn tasks in $\mathcal{T}_{\text{metatrain}}$ during the meta-training phase so that this learning ability generalizes to unseen tasks in $\mathcal{T}_{\text{metatest}}$ during the meta-testing phase.

Specifically, the optimization problem for gradient-based meta-learning can be written as

$$\arg\min_{\theta,\phi} \mathbb{E}_{t\sim\mathcal{T}}\left[\mathcal{L}_t^{\text{test}}\big(\text{InnerLoop}(\theta, \mathcal{L}_t^{\text{train}}; \phi)\big)\right], \tag{6.1}$$

where InnerLoop denotes a learning rule parametrized by ϕ that a model uses to update its own parameters θ based on the training loss $\mathcal{L}_t^{\text{train}}$. Each meta-training iteration consists of doing task-specific inner loop training using the learning rule, evaluating the model with the task loss, back-propagating the loss back through the inner loop, and finally, using the gradients obtained to apply a model update in the outer loop. So long as InnerLoop is differentiable, the model and the learning rule in this meta optimization problem can be trained end-to-end with gradient descent.

[74, 75] initially proposed using a recurrent neural network as the learning rule, but the dominant approach today is to use gradient descent itself as the learning rule. This was first done by the

seminal Model-Agnostic Meta-Learning (MAML) algorithm [78], whose name partially comes from the fact that there is no need to use any specific kind of external model for the inner loop updates.

We summarize below the basic MAML algorithm and a non-exhaustive list of variations that have been proposed.

6.2.1 MAML

For task t and K gradient descent steps of a fixed size α, we can write the MAML inner loop training as follows:

$$\begin{aligned} &\text{InnerLoop}(\theta, \mathcal{L}_t^{\text{train}}) = \theta_{t,K}, \\ &\theta_{t,0} := \theta, \\ &\theta_{t,k} := \theta_{t,k-1} - \alpha \nabla_{\theta_{t,k-1}} \mathcal{L}_t^{\text{train}}(\theta_{t,k-1}). \end{aligned} \tag{6.2}$$

The inner loop updates do not result in an actual model update, but are only intermediate steps used to compute it,

$$\theta' := \theta - \frac{\beta}{T} \sum_{t=1}^{T} \nabla_\theta \mathcal{L}_t^{\text{test}}(\theta_{t,K}), \tag{6.3}$$

where the final loss is a mean of the test loss of the model initialized at θ and separately trained over T sampled tasks.

Notice that before this outer loop update is computed, we have to maintain T distinct versions of the model in memory, where none of them interacts with each other in the inner loop. The special case of $K = 0$ corresponds to multi-task learning, where the inner loop is effectively collapsed and task interactions occur directly.

6.2.2 Meta-SGD

While MAML only meta-learns a good starting initialization θ, Meta-SGD [179] proposes to also meta-learn the learning rule's update direction and learning rate with a vector of learning rates $\boldsymbol{\alpha}$ to improve generalization.

$$\begin{aligned}
&\text{InnerLoop}(\theta, \mathcal{L}_t^{\text{train}}; \boldsymbol{\alpha}) = \theta_{t,K}, \\
&\theta_{t,k} := \theta_{t,k-1} - \boldsymbol{\alpha} \nabla_{\theta_{t,k-1}} \mathcal{L}_t^{\text{train}}(\theta_{t,k-1}), \\
&(\theta', \boldsymbol{\alpha}') := (\theta, \boldsymbol{\alpha}) - \frac{\beta}{T} \sum_{t=1}^{T} \nabla_{(\theta,\boldsymbol{\alpha})} \mathcal{L}_t^{\text{test}}(\theta_{t,K}).
\end{aligned} \tag{6.4}$$

While Meta-SGD preconditions the inner loop gradient with a vector of learning rates, other papers in the literature suggest preconditioning with a block diagonal matrix [180] and task-conditioned operators [181, 182].

6.2.3 MAML++

[183] observed that MAML suffers from noisy training dynamics and is very sensitive to the choice of neural network architecture despite its namesake. The authors recommended a series of fixes that they call MAML++. In addition to meta-learning the learning rate like Meta-SGD (but for each model layer not parameter), two of the most consequential fixes in MAML++ include:

Multi-Step Loss Optimization The outer loop update now consists of the test loss evaluated at all steps of the inner loop, which improves gradient propagation.

$$\theta' := \theta - \frac{\beta}{T} \sum_{t=1}^{T} \sum_{k=0}^{K} \nabla_{\theta} \mathcal{L}_t^{\text{test}}(\theta_{t,k}). \tag{6.5}$$

Per Step Batch Normalization Every batch normalization layer now has an individual copy per inner loop step, with its own weights, biases, and running statistics. This makes optimization smoother because the model can now have different activation distributions for each inner loop step.

6.2.4 Regularization Methods

There are two dominant approaches to mitigating task over-fitting in the meta-learning literature.

The first is to meta-learn parts of the model conditioned on the task. Suggested methods include meta-learning task-specific model parameters [184], loss functions [185], inner loop gradient preconditioners [181, 182], initializations in a low dimensional latent space [186], and dropout parameters for each layer of the model [181]. Task conditioning methods help with task over-fitting, but they also require substantial meta-training before the task-specific conditioning can be meta-learned.

The second is to add a regularization term to the model update (i.e. outer loop update) equation. Proposals include penalties on task entropy [172], task similarity [173], mutual information flow between the test set and parameters unrelated to the inner loop learning process [175], and $\mathcal{L}_2$ distance between the model initialization before and model parameters after the inner loop learning process [174]. Like us, [173]'s meta-learning method is inspired by multi-task learning, but their work applies specifically to first-order approximation methods like Reptile [187], whereas our work requires that the inner loop variables can be meta-learned and hence, applies to full second-order methods like MAML.

Our work is an inner loop regularization method and is not based on task conditioning (although it is complementary to it). This is a relatively under-explored area in the meta-learning literature. The only closely related attempt that we are aware of is DropGrad [188], which randomly drops task gradients during the inner loop. But as with outer loop regularization methods and unlike our work, the meta-learned information has to be fully absorbed by the initialization, leading to slow meta-training.

6.3 Insights from Multi-Task Learning

6.3.1 Multi-Task Learning Regularizes Meta-Learning

Existing meta-learning methods force most of the learned task-invariant information to reside in the high-dimensional model initialization θ. Because the interaction between different tasks happens exclusively in the outer loop and θ can only be updated by backpropagating through multiple gradient steps within the inner loop, this significantly slows down meta-learning and is a major source of training instability [183]. Furthermore, a generic starting initialization without sufficient meta-learned information tends to easily over-fit the training loss and not generalize to the test loss within each task, causing meta-training to be especially sluggish initially. Most improvements to the basic MAML algorithm (c.f. Section 6.2) can be seen as efforts to shift part of the meta-learning away from the initialization and into auxiliary parameters within the inner loop.

Fortunately, these challenges can be naturally overcome if across-task interactions occurred within the inner loop as well. So doing both regularizes the inner loop learning process by reducing task-specific over-fitting and minimizes the meta-learning burden on the outer loop by also learning task-invariant information in the inner loop.

The most straightforward way of enabling inner loop task interactions is to contemporaneously train against multiple tasks, which has a long established history rooted in the paradigm of multi-task learning [177, 178]. Before discussing our proposed algorithm in depth, we briefly review conventional challenges faced in multi-task learning, and show somewhat surprisingly that they disappear within the context of meta-learning, thus allowing meta-learning and multi-task learning to artlessly complement each other.

6.3.2 Meta-Learning Complements Multi-Task Learning

In classical supervised learning, we are training on dataset $\mathcal{A}$ and testing on dataset $\mathcal{B}$ within the same task. The paradigm of multi-task learning proposes that jointly training on both $\mathcal{A}$ and

an auxiliary set of related tasks C_i will improve generalization on $\mathcal{B}$, because the C_i regularize the training towards inductive biases common to both $\mathcal{A}$ and $\mathcal{B}$.

The effect of this regularization crucially depends on the relatedness between C_i, $\mathcal{A}$, and $\mathcal{B}$. If they are loosely related or adversarially related, multi-task learning can instead cause negative transfer and be harmful to task performance and generalization [189, 190, 191, 192]. Moreover, even when they are related, tasks should be combined in such a way that none dominates any other and all tasks have a meaningful contribution to the learned model [193].

Therefore, in practice, good results rely on tuning a set of hyperparameters λ_i to encode task relatedness and control the strength of regularization.

$$\mathcal{L}_{\text{multi-task}} = \mathcal{L}_A + \sum_i \lambda_i \mathcal{L}_{C_i}. \tag{6.6}$$

Finding an appropriate set of hyperparameters λ_i typically involves an expensive grid search or the use of heuristics [193, 194, 195]. Additionally, better performance can be obtained from using λ_i that vary over the training procedure. For example, high λ_i might be preferable at the beginning of the learning process as the model learns common aspects between the tasks. By contrast, lower λ_i might make more sense during late-stage training when the model needs to fine-tune on A. Tuning these dynamic sequences of λ_i is a challenge in the classical multi-task setting.

In a typical meta-learning setup, these challenging issues conveniently cease to be a problem. The λ_i are no longer hyperparameters but instead parameters within the inner loop that can be meta-learned. The λ_i can therefore be automatically and dynamically tuned by the outer loop.

6.3.3 Applying Multi-Task Learning Asynchronously

However, there are a couple of new problems that arise from applying multi-task learning in the inner loop of meta-learning. GPU memory capacity is already a bottleneck in meta-learning, since storing sets of tasks instead of mere data points significantly increases memory use. Hence, sharing information across all tasks at the same time is not feasible. Moreover, at meta-test time,

we are not allowed access to other tasks for training and each unseen task has to be evaluated independently. Directly applying multi-task learning during meta-testing is thus not possible.

Even when we are not able to compute new gradients from other concurrent tasks, we observe that we can sidestep this problem by reusing information that has been computed in previous iterations. Hence, by storing task information from previously encountered tasks in external memory, we can solve both the problem of small task batches and also enable multi-task learning during meta-test time. In fact, memory-based approaches to meta-learning have been very successful, but they generally require substantial amounts of computational resources and violate the model-agnostic nature of MAML [167, 196].

Therefore, instead of storing information from related tasks directly, we propose to store information from related task gradients. This can be done in a lightweight and model-agnostic nature by simply maintaining a running mean of task gradients, similar to how batch normalization maintains a running mean of layer activations [197]. We present the gradient sharing algorithm in the next section that explains our proposal in detail.

6.4 Gradient Sharing

Gradient sharing augments the standard MAML inner loop with a meta-learned regularizer that shares gradient information from related tasks and is parametrized by $\boldsymbol{m} \in \mathbb{R}^K, \boldsymbol{\lambda} \in \mathbb{R}^K$. σ denotes the sigmoid function.

$$\text{InnerLoop}(\theta, \mathcal{L}_t^{\text{train}}; \boldsymbol{m}, \boldsymbol{\lambda}) = \theta_{t,K}. \tag{6.7}$$

At the k-th step of the inner loop, we first compute the normalized average gradient across the task batch (Equation 6.8), and use it to update a running mean of task gradients $\hat{g_k}$ with an exponential moving average factor $\sigma(\boldsymbol{m}_k)$ (Equation 6.9, 6.10). $\boldsymbol{m}_k$ can be seen as a momentum variable that controls the weight of recent gradient information relative to past gradients. While the model is largely malleable in the early stages of meta-training, it makes sense for $\boldsymbol{m}_k$ to be large so as to keep

pace with quickly changing task gradients. By contrast, near the end of meta-training, variations in task gradients can mostly be attributed to sampling noise and thus, a small $\boldsymbol{m}_k$ is needed for stable training. Meta-learning $\boldsymbol{m}_k$ gives the outer loop flexibility to adapt to both scenarios.

Next, the inner loop update is performed with $\Delta_{t,k}$ which is a $\sigma(\boldsymbol{\lambda}_k)$-weighted linear interpolation between the current task gradient $\nabla_{\theta_{t,k-1}}\mathcal{L}_t^{\text{train}}(\theta_{t,k-1})$ and the running mean task gradient $\hat{g_k}$ (Equation 6.11). $\boldsymbol{\lambda}_k$ is a gating variable that decides the strength of the multi-task learning regularization coming from related task gradients encountered in the current task batch and previously seen tasks. It is also meta-learned, thus allowing both the task distribution and the size of the task batches to determine the appropriate amount of regularization. For simplicity and storage efficiency, we choose to have a single parameter $\boldsymbol{\lambda}_k$ model the relatedness of each task to all other tasks, although it is straightforward to extend the proposed version of gradient sharing to use task-conditioned parameters $\boldsymbol{\lambda}_{t,k}$ to yield a direct equivalent of our discussion in Equation 6.6.

Finally, we combine the inner loop task losses to arrive at the outer loop update (Equation 6.12). At meta-test time, the inner loop is regularized using the $\hat{g_k}$ stored during meta-training.

We write the full gradient sharing algorithm in pseudo-code for vanilla MAML during the meta-training and meta-testing phase in Algorithms 4 and 5 respectively. Notice that applying regularization in an inner loop gradient step changes subsequent gradient steps taken. While MAML is usually written by looping over the task batch first and then the inner loop gradient steps, we have to exchange the order of the two for loops in our algorithm. This does not affect the efficiency of meta-learning, since the outer loop update can only be applied only after the completion of the inner loop. Both orderings use memory and compute scaling in $O(TK)$ per model update.

While the pseudo-code is written for vanilla MAML, gradient sharing can be applied in general to any second-order gradient-based meta-learning method (i.e. the inner loop has to be differentiable) by using the regularized $\Delta_{t,k}$ in the place of an inner loop task gradient $\nabla_{\theta_{t,k-1}}\mathcal{L}_t^{\text{train}}(\theta_{t,k-1})$.

6.5 Experimental Results and Discussions

We study the effects of gradient sharing using two popular few-shot image classification datasets, the Caltech-UCSD Birds-200-2011 (CUB) dataset [198] and the MiniImagenet dataset [75]. The former consists of 100 meta-training classes, 50 meta-validation classes, and 50 meta-test classes. The latter consists of 64 meta-training classes, 16 meta-validation classes, and 20 meta-test classes. Each task involves 5-way and $1/5$-shot classification on randomly sampled classes using the cross-entropy loss. The goal of our experiments is to answer the following questions:

1. Does gradient sharing accelerate meta-training?
2. Does gradient sharing enable higher learning rates in the inner loop?
3. How does gradient sharing affect the eventual meta-test performance compared to the baseline?
4. How do $\boldsymbol{m}$ and $\boldsymbol{\lambda}$ change as meta-training proceeds?

The size of the task batch affects the variance of the task gradients and in the case of size 1, there is an absence of gradients from related tasks in the same batch. Different meta-learning methods also induce different meta-training dynamics: for example, MAML uses static learning rates, while Meta-SGD meta-learns them. These factors affect the degree of task over-fitting and the rate at which the outer loop corrects the inner loop over-fitting. Hence, in the interest of a comprehensive experimental setup, we study answers to the above questions in two distinct regimes — task batches of size 1 and 5 — and across three distinct meta-learning methods — MAML, Meta-SGD, and MAML++.

For task batch size 5, we meta-train on CUB for 150 epochs and MiniImagenet for 250 epochs using outer loop Adam [115] with default hyperparameters and inner loop gradient descent with learning rate 0.1 and $K = 5$ steps. For task batch size 1, we do 5x as many epochs. Each epoch consists of 1000 iterations. More experimental details can be found in Appendix Section B.1.

We present our main findings with diagrams containing select subsets of all the experiments. The interested reader is encouraged to verify that they hold generally and learn about particular experimental nuances by viewing the full spectrum of our experiments in Appendix Section B.2.

6.5.1 Acceleration of Meta-Training

We observe in Figure 6.2 that in the initial phase of meta-training, prior to the initialization meta-learning sufficient task-invariant information, gradient sharing results in higher meta-validation performance. This effect is significantly more pronounced when there are other concurrent tasks in the task batch, due to stronger regularization and smaller variance in task gradients, as we can see by comparing the plots for task batch size 5 versus 1. This is clear evidence that gradient sharing is indeed reducing inner loop over-fitting, because it consistently results in higher inner loop test performance early on (Recall that the outer loop loss is a mean of the inner loop test losses).

Achieving superior meta-training performance early on accelerates the overall meta-training process. To quantify the amount of meta-training acceleration, we use the rate at which the highest meta-validation accuracy is achieved as a proxy. Specifically, we calculate Speed-up = $\frac{\text{Epoch}_{\text{OG}}-\text{Epoch}_{\text{GS}}}{\text{Epoch}_{\text{GS}}}$ where Epoch_{OG} and Epoch_{GS} is the earliest epoch when the highest meta-validation accuracy is achieved for the original baseline and gradient sharing respectively. We see in Table 6.1 that gradient sharing results in a non-trivial amount of meta-training acceleration, potentially a speed-up of up to 134% at comparable levels of meta-test accuracy.

6.5.2 Bigger Inner Loop Learning Rates

Another advantage to reducing inner loop over-fitting is the ability to use higher learning rates. Figure 6.3 shows that gradient sharing achieves successful meta-training even when the inner loop learners have been initialized with 10x their learning rate. Even though in theory, methods like Meta-SGD and MAML++ allow the outer loop to adjust the inner loop learning rate to enable training, we see that the baselines often fail to train at all or experience very sluggish meta-training.

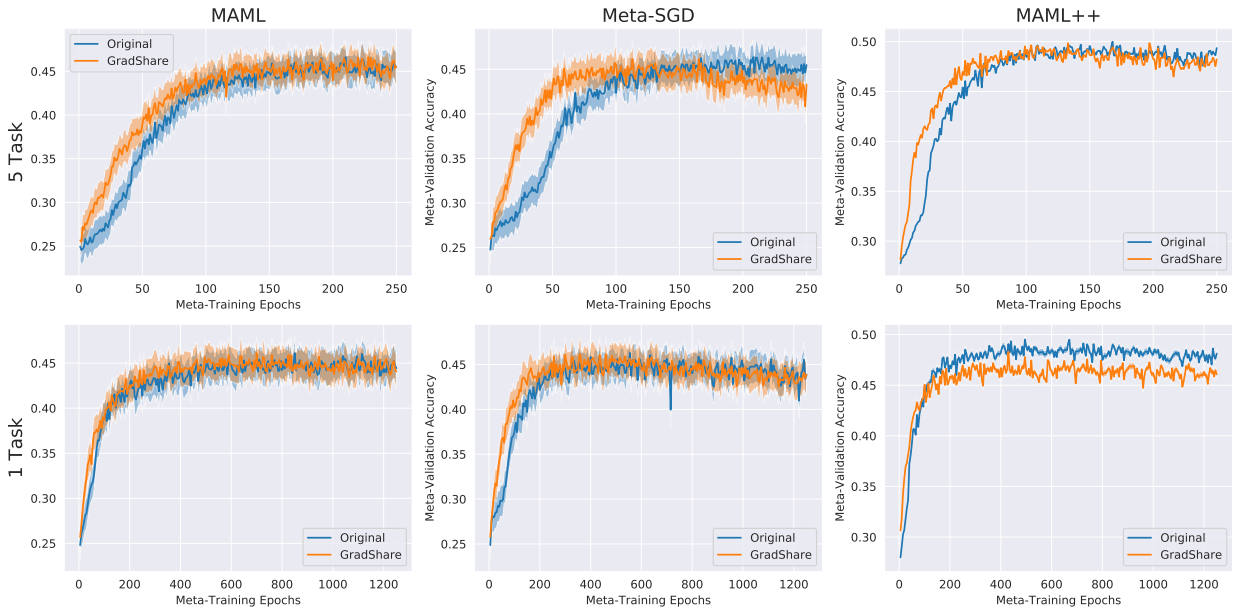

Figure 6.2: Meta-validation accuracy plots on 5-way 1-shot classification on the MiniImagenet dataset. Gradient sharing accelerates meta-learning by reducing inner loop over-fitting in early stage meta-training. The acceleration is more pronounced when there are other concurrent tasks in the inner loop.

Higher inner loop learning rates produces superior meta-test generalization under certain circumstances [179], and additional robustness to meta learning hyperparameters is generally very desirable.

6.5.3 Comparable Meta-Test Performance

We did meta-testing using an ensemble of the top 5 meta-validation accuracy models following the methodology of the MAML++ paper [183]. From Table 6.1, we see that gradient sharing achieves comparable or better meta-test performance than the respective baselines. It is important to note that optimization acceleration schemes in the conventional machine learning literature are often prone to introducing biases in the model that adversely impact generalization [147, 148]. Despite the complexities of the inner and outer loop interactions, gradient sharing achieves significant acceleration without compromising on meta-test performance.

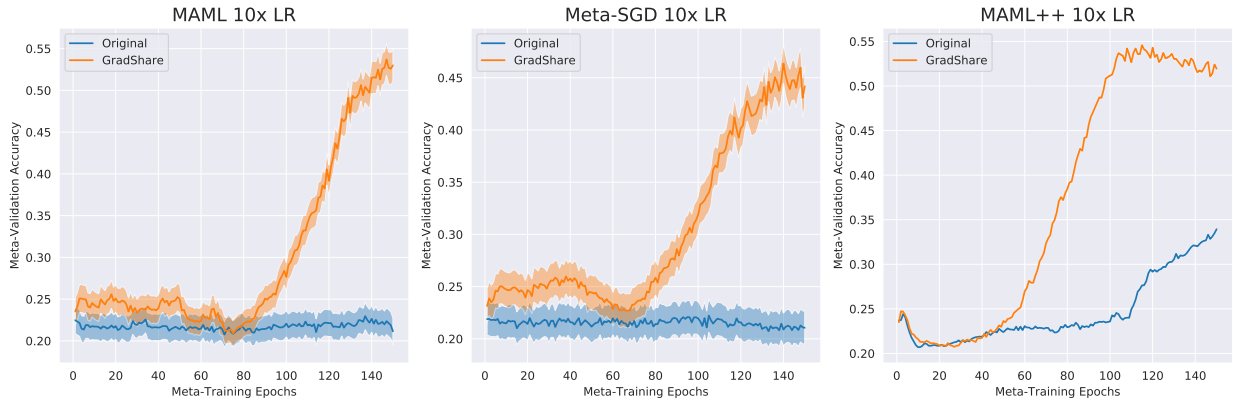

Figure 6.3: Meta-validation accuracy plots on 5-way 1-shot classification on the CUB dataset with task batch size 5 and 10x the inner loop learning rate. Gradient sharing successfully enables meta-training on baseline meta-learning methods that either do not meta-train at all or experience sluggish meta-training.

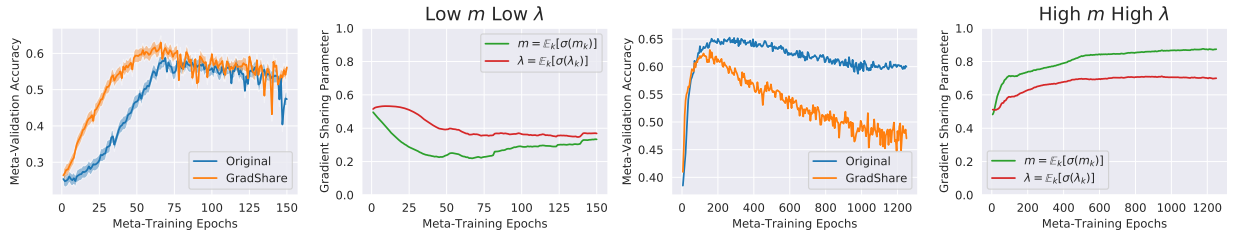

Figure 6.4: The left two plots show the results of meta-training using gradient sharing on 5-way 1-shot classification on CUB using MAML with task batch size 5. They represent a successful example of gradient sharing with the outer loop meta-learning low values for both $\boldsymbol{m}$ and $\boldsymbol{\lambda}$. The right two plots show meta-training results for 5-way 5-shot classification on MiniImagenet using MAML++ with task batch size 1. They represent a pathological example of gradient sharing with the outer loop meta-learning high values for both $\boldsymbol{m}$ and $\boldsymbol{\lambda}$.

6.5.4 Evolution of $\boldsymbol{m}$ and $\boldsymbol{\lambda}$ through Meta-Training

In the previous sections, we had argued that meta-learning compliments multi-task learning by allowing us to meta-learn the task combination coefficients. In gradient sharing, this amounts to tuning $\boldsymbol{m}_k$ and $\boldsymbol{\lambda}_k$ to their appropriate values. On the left two sub-figures of Figure 6.4, we observe that accelerated meta-training goes hand in hand with reduced values of the averages of $\boldsymbol{m}_k$ and $\boldsymbol{\lambda}_k$ as meta-training proceeds, which agrees with what we had discussed in Sections 6.3 and 6.4.

The right two sub-figures of Figure 6.4 show a characteristically different pattern. We observe that the outer loop meta-learns high values of $\boldsymbol{m}_k$ and $\boldsymbol{\lambda}_k$ as meta-training proceeds. High $\boldsymbol{m}_k$

indicates that the store of task gradients $\hat{g_k}$ has not stabilized and recent task gradients are continually overwriting it. High $\boldsymbol{\lambda}_k$ suggests an excessive amount of regularization is being applied; in fact, at the limit of $\sigma(\boldsymbol{\lambda}_k) = 1.0$, the true task gradient $\nabla_{\theta_{t,k-1}} \mathcal{L}_t^{\text{train}}(\theta_{t,k-1})$ is completely masked out, effectively making it zero-shot instead of few-shot learning. This pathological phenomenon of high $\boldsymbol{m}_k$ and $\boldsymbol{\lambda}_k$ is congruent with the observed result of gradient sharing exacerbating the original MAML++ baseline's outer loop over-fitting in this case.

The over-fitting of the outer loop has not proven to be a serious issue in our work due to the use of early stopping (since we select the meta-test model using the meta-validation set). However, looking into combinations of outer and inner loop regularization, for example task-conditioned $\boldsymbol{m}_{t,k}$ and $\boldsymbol{\lambda}_{t,k}$, is an important topic for future work.

6.6 Conclusion

In this work, we developed a technique inspired by multi-task learning to mitigate over-fitting within the inner loop of meta-learning. Our proposed method accelerates meta-training under comparable meta-test performance and makes it robust to inner loop learners with higher learning rates. Given that meta-learning is significantly more computationally expensive than conventional machine learning, we hope that our work will inspire more research into inner loop regularization methods for meta-learning that will accelerate meta-training. Alternative methods would be especially helpful for the case of meta-training under task batch size 1, since the lack of concurrent tasks limits the utility of our multi-task learning inspired solution. Finally, further such research can also be expected to robustify meta-learning so that it works with a wider range of inner loop learners.

Algorithm 4: Gradient Sharing for MAML Meta-Training.

Initialize θ, T, K, $\boldsymbol{m} = \mathbf{0}$, $\boldsymbol{\lambda} = \mathbf{0}$.
for $i = 1$ **to** $numMetatrainIters$ **do**
  Sample batch $\mathcal{B}$ with T tasks from meta-training set.
  Initialize $\theta_{t,0} = \theta$ for all tasks t in $\mathcal{B}$.
  // K is the number of inner loop gradient steps.
  for $k = 1$ **to** K **do**
    // Calculate normalized mean of task gradients in $\mathcal{B}$.

$$\boxed{g_k = \frac{\sum_{t=1}^{T} \nabla_{\theta_{t,k-1}} \mathcal{L}_t^{\text{train}}(\theta_{t,k-1})}{||\sum_{t=1}^{T} \nabla_{\theta_{t,k-1}} \mathcal{L}_t^{\text{train}}(\theta_{t,k-1})||_2}.} \tag{6.8}$$

    // Calculate running mean gradient statistics $\hat{g_k}$.
    if $i = 1$ **then**

$$\boxed{\hat{g_k} = g_k.} \tag{6.9}$$

    else

$$\boxed{\hat{g_k} = \sigma(\boldsymbol{m}_k) g_k + (1 - \sigma(\boldsymbol{m}_k)) \hat{g_k}.} \tag{6.10}$$

    end if
    for task t **in** batch $\mathcal{B}$ **do**
      // Apply inner loop update.

$$\boxed{\begin{aligned} \Delta_{t,k} &= \sigma(\boldsymbol{\lambda}_k) \hat{g_k} + (1 - \sigma(\boldsymbol{\lambda}_k)) \nabla_{\theta_{t,k-1}} \mathcal{L}_t^{\text{train}}(\theta_{t,k-1}). \\ \theta_{t,k} &= \theta_{t,k-1} - \alpha \Delta_{t,k}. \end{aligned}} \tag{6.11}$$

    end for
  end for
  // Apply outer loop update.

$$\boxed{(\theta', \boldsymbol{m}', \boldsymbol{\lambda}') = (\theta, \boldsymbol{m}, \boldsymbol{\lambda}) - \frac{\beta}{T} \sum_{t=1}^{T} \nabla_{(\theta, \boldsymbol{m}, \boldsymbol{\lambda})} \mathcal{L}_t^{\text{test}}(\theta_{t,K}).} \tag{6.12}$$

end for

Algorithm 5: Gradient Sharing for MAML Meta-Testing.

for task t **in** meta-testing set **do**
  Initialize $\theta_{t,0} = \theta$.
  for $k = 1$ **to** K **do**
    $\Delta_{t,k} = \sigma(\boldsymbol{\lambda}_k)\hat{g}_k \; +$
      $(1 - \sigma(\boldsymbol{\lambda}_k))\nabla_{\theta_{t,k-1}}\mathcal{L}_t^{\text{train}}(\theta_{t,k-1})$.
    $\theta_{t,k} = \theta_{t,k-1} - \alpha\Delta_{t,k}$.
  end for
  Evaluate task t's test performance with $\mathcal{L}_t^{\text{test}}(\theta_{t,K})$.
end for

Table 6.1: Meta-test accuracy (with 95% confidence intervals) and speed-up for 5-way 5-shot classification for the CUB and MiniImagenet datasets. Gradient sharing achieves comparable meta-test accuracy, but often in a fraction of the number of meta-training epochs.

		CUB			MINIIMAGENET		
METHOD	TASKS	ORIGINAL	GRADSHARE	SPEED-UP	ORIGINAL	GRADSHARE	SPEED-UP
MAML	5	83.2 ± 1.4%	83.4 ± 1.4%	66%	67.7 ± 1.8%	67.0 ± 1.8%	1%
MAML	1	82.6 ± 1.5%	82.7 ± 1.5%	44%	66.4 ± 1.8%	68.3 ± 1.8%	134%
META-SGD	5	80.7 ± 1.5%	80.3 ± 1.5%	100%	67.0 ± 1.8%	67.4 ± 1.8%	61%
META-SGD	1	81.6 ± 1.5%	79.6 ± 1.6%	54%	64.8 ± 1.9%	64.9 ± 1.9%	37%
MAML++	5	72.7 ± 1.7%	73.8 ± 1.7%	42%	68.9 ± 1.8%	69.4 ± 1.8%	26%
MAML++	1	76.1 ± 1.7%	76.5 ± 1.6%	100%	69.1 ± 1.8%	66.8 ± 1.8%	71%

Chapter 7: Logical Networks

7.1 Introduction

Machine learning systems have become increasingly capable at a wide range of tasks, with neural network based models outperforming humans at tasks like object recognition [8], speech recognition [9, 199], the game of Go [200, 11], Atari videogames [201, 202], and more. Nonetheless, the success of deep learning comes with significant caveats: neural networks require immense amounts of labeled data for training, can be easily tricked by tiny input perturbations or spurious correlations, and succumb to brittle generalization when tested on data that deviate ever so modestly from the training distribution. Critics point to these caveats as evidence that deep learning, in its current incarnation, is really just performing a sophisticated type of pattern matching, the likes of which can only ever constitute intelligence in narrow, circumscribed domains [203, 204].

By comparison, human intelligence can be applied more generally. This has been argued to be a result of two distinct modes of cognition: System 1 and System 2 [205, 206]. System 1 happens quickly and without conscious effort, for example comparing the size of objects or locating the general source of a sound. On the other hand, System 2 involves slow and deliberate attention, for example solving for a complicated arithmetic equation or checking that an argument is logical. Current machine learning systems have been likened to System 1 [207], because System 1 mostly involves the use of associative memory, and is highly susceptible to cognitive biases and sensory illusions. Symbolic AI algorithms that are based on logic and search more closely resemble System 2.

To achieve robust human-level AI that can solve non-trivial cognitive tasks, it is crucial to combine both System 1 like pattern recognition and System 2 like logical reasoning capabilities in a seamless end-to-end learning fashion. This is because in many practical problems of inter-

est, it is difficult and expensive to collect intermediate labels to train specific machine learning sub-components. For example, it appears infeasible to build a 'danger' classifier for a self-driving car, where every possible dangerous scenario is pre-determined and categorized beforehand. Researchers are thus far able to combine both capabilities in a single AI system, but not train them end-to-end. Famously, OpenAI's very impressive achievement of controlling a robotic hand to solve a Rubik's cube required the separate use of a machine learning system to perform the dexterous manipulation and a discrete solver to decide the side of the cube that should be turned [208].

Attempts to bridge the two capabilities seamlessly belong to one of three approaches. The first involves augmenting deep learning models with soft logic operators [209, 210, 211, 212, 213, 214, 215] or combinatorial solving modules [216, 217, 218, 219, 220]. However, this approach typically requires the programmer to pre-specify intricate logical structures according to the problem domain. Moreover, these logical components are fixed and not amenable to learning. The second approach uses sub-symbolic reasoning techniques like Recurrent Relation Networks to implicitly pick up on logical structures within the problem [170, 221, 222]. This approach improves on the first by learning the logical structure implicitly by optimization, but nevertheless also necessitates careful feature engineering. The third approach is the field of inductive logic programming (ILP), which starts from a traditional symbolic AI model like a knowledge base, and adds learning capabilities to it [223, 224, 225, 226]. Unfortunately, ILP is limited to symbolic inputs and outputs, unlike deep neural networks.

Against the backdrop of such approaches, SATNet [83] promised to integrate "logical structures within deep learning" with a differentiable MAXSAT solver that can infer logical rules and be used as a neural network layer. SATNet claimed to have solved problems that were "impossible for traditional deep learning methods and existing logical learning methods to reliably learn without any prior knowledge," most notably solving a Sudoku puzzle visually from images of puzzle digits, and was awarded with a Best Paper Honorable Mention at 2019's *International Conference on Machine Learning*.

Based on SATNet's success, one might think that enabling end-to-end gradient-based optimiza-

tion (i.e. making every component in a system differentiable) is sufficient for end-to-end learning (i.e. learning without intermediate supervision signals). However, defining gradients for an objective does not, on its own, result in successful learning outcomes, as exemplified by the history of deep learning. Successful training of architectures with hundreds of layers, where gradients are trivially well defined, is highly non-trivial and requires careful initialization, batch normalization, adaptive learning rates, etc. Additionally, without an appropriate inductive bias (like the rules of the game), learning to solve complex problems like visual Sudoku from relatively few samples is extraordinarily challenging. It is unlikely that end-to-end gradient-based optimization by itself will, in general, result in models that generalize well.

Thus, SATNet's claim to have solved the end-to-end learning problem of visual Sudoku "in a minimally supervised fashion" should be revisited. **Can SATNet learn to assign logical variables (symbols) to images of digits (perceptual phenomena) without explicit supervision of this mapping?** This is also known as the symbol grounding problem [84], which has long been thought to be a prerequisite for intelligent agents to perform real-world logical reasoning. If answered in the affirmative, SATNet would have marked a revolutionary leap forward for the whole field of AI, by virtue of the difficulty of the symbol grounding problem in visual Sudoku.

The general complexity of the symbol grounding problem embedded in end-to-end learning should not be underestimated. Figure 7.1 directly exemplifies the difficulty of the symbol grounding problem for both human and artificial intelligence. Common measures of abstract reasoning in artificial intelligence such as DeepMind's PGM work similarly to Raven's Progressive Matrices (a test for human intelligence), where predicting what comes next involves determining the hidden attributes (symbols) in what has been presented (perceptual phenomena), and inferring the pattern from them [204, 227, 228, 229]. Once given the hidden attributes, it is trivial for a human or a combinatorial solver to infer the pattern [227]. However, jointly inferring the hidden attributes together with the pattern proves to be a challenging cognitive task in general.

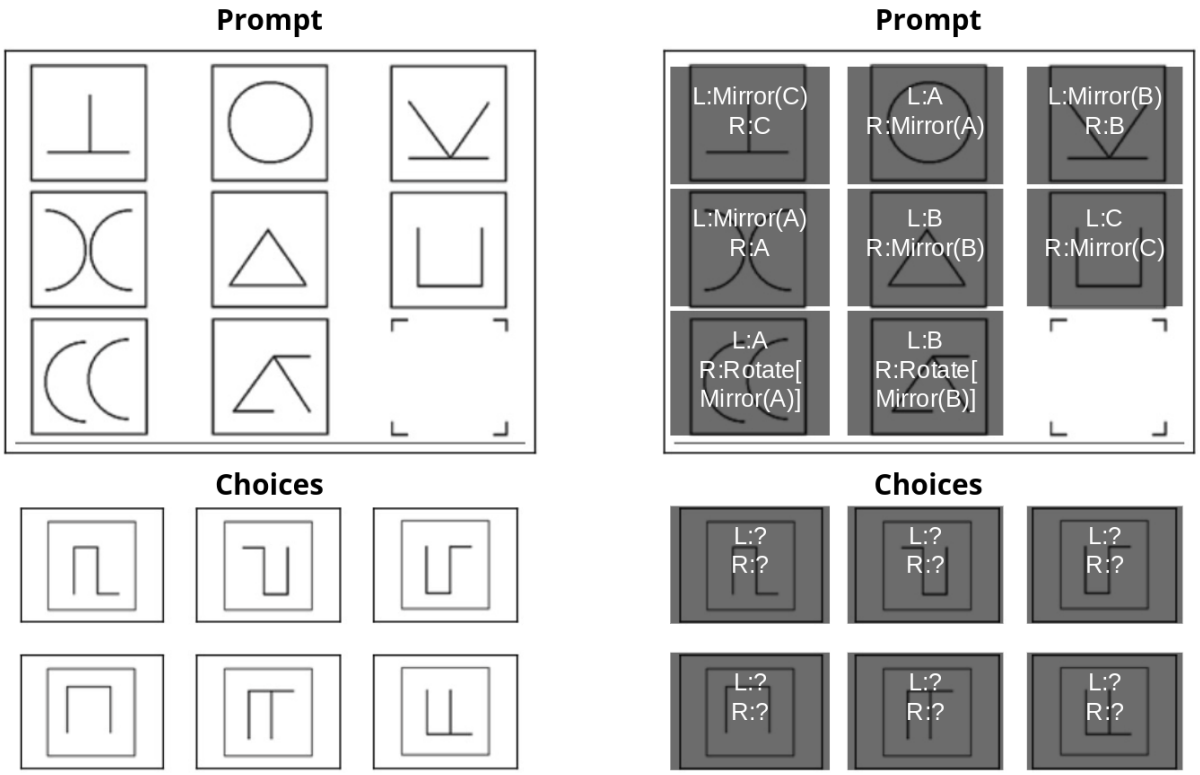

Figure 7.1: A challenging Raven's Matrix puzzle that exemplifies a difficult instance of the symbol grounding problem. We invite the reader to attempt the puzzle for themselves on the left hand side of the figure first, before looking at the annotations on the right hand side. Once the given images have been decoded to an appropriate symbolic representation, it is straightforward for a discrete solver or a human to solve it. For a full explanation of the solution, please see Appendix Section C.1.

7.1.1 Our Contribution

In this chapter, our principal contribution is a re-assessment of SATNet that clarifies the extent of its capabilities and a discussion of practical solutions that will help future researchers train SATNet layers in deep networks.

First, we observed from the SATNet authors' open-source code that intermediate labels are leaked in the SATNet training process for visual Sudoku. The leaked labels essentially result in a two-step training process for SATNet, where it first uses the leaked labels to train a digit classifier, and then uses the symbolic representations of the digits to solve for the Sudoku puzzle. After removing the intermediate labels, SATNet was observed to completely fail at visual Sudoku (0%

test accuracy). If intermediate labels are available, it is possible to separately pre-train a digit classifier and then use SATNet, independent of a deep network, to solve for the puzzle. This might even be preferable, given our finding that SATNet fails in 8 out of 10 random seeds despite access to the labels, which is evidence that SATNet struggles to learn to ground the Sudoku digits into their symbolic representation. To be clear, the label leakage did not affect SATNet in the non-visual case, and its success on purely symbolic inputs and outputs nonetheless marks progress in ILP, but does not fix the field's persisting deficiency in dealing with perceptual input.

While solving difficult instances of the symbol grounding problem like visual Sudoku or PGM might be beyond the reach of SATNet, we found that SATNet also cannot solve easy instances, unless properly configured. We devised a test called the MNIST mapping problem, whose solution requires merely digit classification (a simple problem for neural networks) and learning a bijective mapping between logical variables (a simple problem for discrete solvers). This test serves as an easy instance of the symbol grounding problem, and is suitable as a sanity test not just for SATNet, but other prospective differentiable symbolic solvers. Even on a simple test like this, a naive application of SATNet can cause it to perform worse than models without logical reasoning capabilities.

Our work identifies several factors that affect the learning dynamics of SATNet and provides practical suggestions for configuring SATNet to enable successful training. We reveal surprising complexities that are unique to SATNet and break standard deep learning norms. For example, using different learning rates for different layers in neural networks is not a common practice, since the use of Adam usually suffices. But for the case of SATNet, even when Adam is used, the backbone layer has to learn at a slower rate than the SATNet layer for successful training to occur. Surprisingly, we found that unconditionally increasing the number of auxiliary variables does not increase the expressivity of the model, but instead leads to a complete failure in learning. Further adjusting the choice of optimizer and neural architecture led to statistically significant improvements, culminating in near perfect test accuracy (99%).

The rest of the chapter is organized as follows: Section 7.2 reviews the relevant technical

background for SATNet and visual Sudoku. Section 7.3 examines the subtle nature of the label leakage in the original SATNet paper and its ramifications. Section 7.4 describes the MNIST mapping problem, and investigates optimal SATNet configurations for this simple MNIST-based test. Finally, we conclude in Section 7.5.

7.2 Background

7.2.1 SATNet

SATNet is a neural network layer that solves a semidefinite programming (SDP) relaxation of the following MAXSAT problem,

$$\max_{\tilde{v}\in\{-1,1\}^n} \sum_{j=1}^{m} \bigvee_{i=1}^{n} \mathbf{1}\left\{\tilde{s}_{ij}\tilde{v}_i > 0\right\}, \tag{7.1}$$

where $\tilde{v} \in \{-1, 1\}^n$ denotes assignments to n binary variables, and $\tilde{s}_i \in \{-1, 0, 1\}^m$ denotes the sign of variable $\tilde{v}_i$ in m clauses. The set of $\tilde{s}_{ij}$, denoted by S, forms the SATNet layer's learnable parameters. $\tilde{v}$ can be partitioned into two disjoint sets $\mathcal{I}$ and $\mathcal{O}$, which are represented in SATNet by layer inputs $Z_\mathcal{I}$ and outputs $Z_\mathcal{O}$ (which can be either probabilistic or strictly binary), and their respective continuous relaxations $V_\mathcal{I}$ and $V_\mathcal{O}$. Gradients from the layer output $\nabla_{Z_\mathcal{O}}\mathcal{L}$ are backpropagated to both the layer's weights in the form of $\nabla_S\mathcal{L}$ and to the layer input in the form of $\nabla_{Z_\mathcal{I}}\mathcal{L}$. The two main tunable hyperparameters in a SATNet layer are the number of clauses m and the number of auxiliary variables aux (which "play a role akin to register memory that is useful for inference"). Auxiliary variables are also input variables, but unlike $Z_\mathcal{I}$, they are not the output of preceding layers.

7.2.2 Visual Sudoku

Sudoku is a number puzzle played out on a 9-by-9 grid. Each of the 9x9=81 cells has to contain a digit from 1 to 9. The game starts out from a partially filled grid, and the object of the game is to complete the rest of the cells on the grid. Each of the digits from 1 to 9 has to appear

exactly once in every row, column, and each of the nine 3-by-3 subgrids. In the non-visual case, the state of the Sudoku grid can be encoded using 9x81=729 binary variables, and SATNet can learn to map from the binary encoding of the initial grid to the binary encoding of the completed grid without the programmer having to explicitly encode for the rules of the game. Given 9000 training and 1000 test examples (with 36.2 pre-filled cells on average), where each example is a pair consisting of the initial and completed grid, SATNet achieves 99.7% training and 98.3% test accuracy. By comparison, a symbolic solver that knows the rules of the game can provably solve the game perfectly [230], while a purely deep learning based approach, trained on a million examples, scores 70.0% on a test set of thirty games [231]. We report on other related work on non-visual Sudoku in Appendix Section C.2.

In visual Sudoku, the inputs are now 81 images of digits (taken from the MNIST dataset), with '0' standing in for empty cells. They are processed by a convolutional neural network (CNN) backbone with a SATNet layer, which performs at 93.6% training and 63.2% test accuracy using the same number of training and test examples. The SATNet authors contextualized their findings by claiming that the "theoretical best" test accuracy is capped at 74.8% ($\approx 0.992^{36.2}$), which is the probability that the LeNet[1] CNN backbone, which has 99.2% test accuracy on MNIST, has correctly classified all the pre-filled cells.

7.3 SATNet Fails at Symbol Grounding

7.3.1 The Absence of Output Masking

While every Sudoku puzzle corresponds to 729 logical variables in the MAXSAT problem (excluding the auxiliary variables for now), the number of pre-filled cells and their positions differ depending on the puzzle. Thus, $\mathcal{I}$ and $\mathcal{O}$ are different for each example, even though the sizes of $Z_{\mathcal{I}}$ and $Z_{\mathcal{O}}$ are fixed beforehand and not example-dependent. A straightforward way to solve this is to apply an appropriate bit mask depending on the example.

Consider a toy example with 5 variables $v_1 = 1, v_2 = 0, v_3 = 0, v_4 = 1, v_5 = 0$ where $\mathcal{I} =$

[1] To be precise, the SATNet authors used a bigger version with ~10x more parameters than the original.

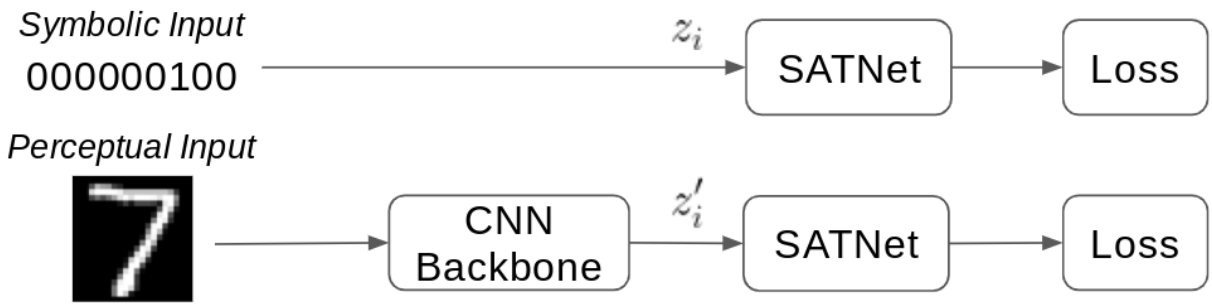

Figure 7.2: A visualization of the difference between symbolic and perceptual inputs.

$\{1, 2, 3\}$ and $\mathcal{O} = \{4, 5\}$. Then, the input to SATNet should be 10000 with the bit mask 11100, and the output should be 00010 with the bit mask 00011. The problem with the original SATNet implementation is that the bits that correspond to the inputs are not masked in the output.

Not masking the output might not seem problematic, given that SATNet does not modify input variables $Z_{\mathcal{I}}$ nor their relaxations $V_{\mathcal{I}}$. But consider the decomposition of the loss function $\mathcal{L}$ into a sum of binary cross entropies (BCE) between the SATNet variables z and the training label l.

$$\mathcal{L} = \sum_{i=1}^{n} \mathrm{BCE}(z_i, l_i) = \sum_{i \in \mathcal{I}} \mathrm{BCE}(z_i, l_i) + \sum_{o \in \mathcal{O}} \mathrm{BCE}(z_o, l_o). \tag{7.2}$$

Since the $z_{\mathcal{I}}$ are not modified by SATNet, $z_i = l_i$ for $i \in \mathcal{I}$, effectively zero-ing out any loss contributed by terms in $z_{\mathcal{I}}$. This is true when SATNet is applied to purely symbolic problems like non-visual Sudoku.

However, once perceptual input is introduced, z_i is not directly accessible by SATNet. Instead, the input to the SATNet layer is a symbolic representation z'_i of features extracted from the data (see Figure 7.2). Thus, the loss from $z_{\mathcal{I}}$ in Equation 7.2 is non-zero before the neural network has learned to ground the symbols appropriately, i.e. $z'_i = z_i = l_i$. Not masking the output to SATNet thus leaks label information to the layers before the SATNet layer, effectively training a classifier that learns to map from the perceptual data to the appropriate symbol representation, i.e. symbol grounding.

7.3.2 Visual Sudoku

Table 7.1: Effects of Output Masking

	Non-Visual Sudoku		Visual Sudoku	
Accuracy	Original	Masked Outputs	Original	Masked Outputs
Train	99.7±0.0%	99.7±0.0%	18.5±12.3%	0.0±0.0%
Test	97.6±0.1%	97.6±0.1%	11.9±7.9%	0.0±0.0%

We re-ran the Sudoku experiments using the SATNet authors' open-sourced implementation with identical experimental settings, but over 10 different random seeds to get standard error confidence intervals. Table 7.1 shows clearly that output masking does not affect the results in the non-visual case, but causes SATNet to fail completely for visual Sudoku, which is what we expect from the discussion in the previous section. Once the intermediate labels are gone, the CNN does not ever learn to classify the digits better than chance. SATNet's failure at symbol grounding directly leads to its failure at the overall visual Sudoku task.

Interestingly, we also found that SATNet's performance in visual Sudoku in the absence of output masking is highly dependent on the random initialization, with 8/10 random seeds leading to complete failure as well. This explains why SATNet's performance over 10 runs (18.5% training accuracy) is dramatically lower than what was originally reported (93.6% training accuracy). Therefore, even for problems where we have access to intermediate labels, leaking them indirectly via the absence of output masking is strictly less desirable than directly pre-training a neural network classifier with those labels. In Section 7.4.1, we note important strategies for mitigating complete failure.

Of the 2 runs that succeeded (i.e. had non-zero training accuracy, specifically 93.2% and 91.7% respectively), we found that the label leakage basically results in a two-step training process for SATNet, where the CNN first learns to do MNIST digit classification, and then the SATNet layer learns to solve the actual Sudoku problem. We show in Figure 7.3 training accuracy plots of two example runs, one successful and the other not. They are annotated with corresponding plots (at the bottom for comparison) of the CNN's classification accuracy on the MNIST test set. For the

successful runs, we observe that the training accuracy for visual Sudoku stays at zero for a small number of epochs, during which time the leaked labels help train the CNN to be an MNIST digit classifier. Only after the digit classifier works to some degree, does the training accuracy for visual Sudoku actually become non-zero. By contrast, in most of the unsuccessful runs, the CNN takes a very long time to become somewhat proficient at digit classification, and even after it does so, the SATNet layer seems unable to adapt to it, resulting in a permanent plateau at 0% training accuracy.

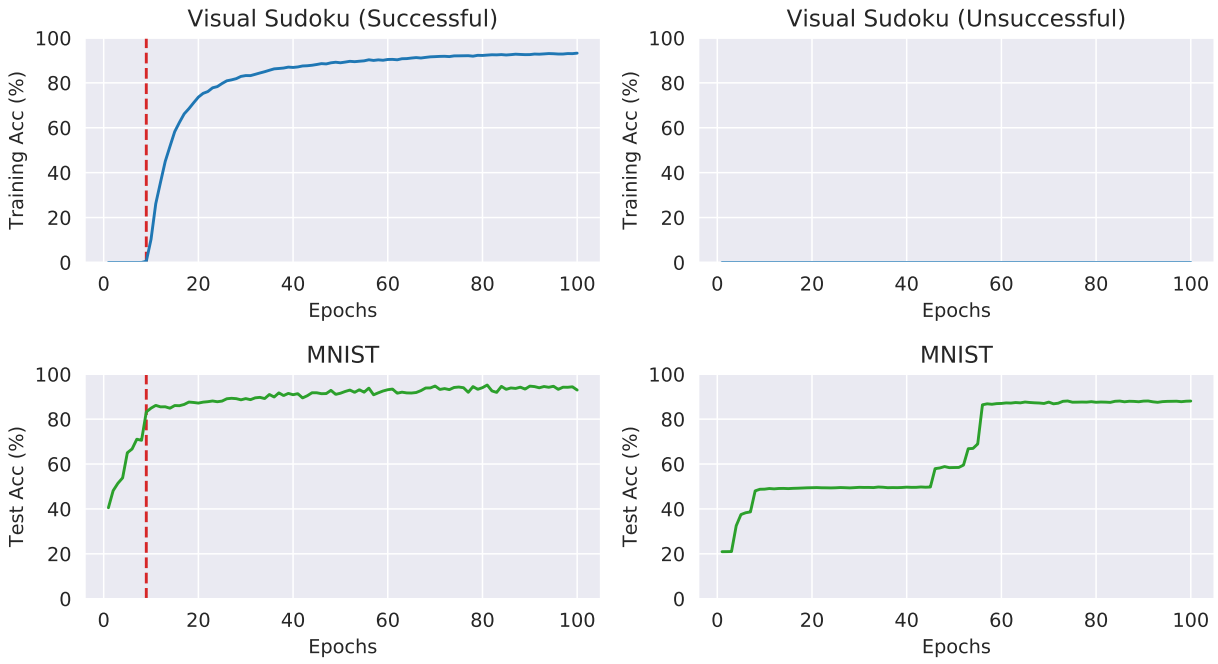

Figure 7.3: The graphs on the left show a successful run of SATNet on visual Sudoku, while the graphs on the right show an unsuccessful run. The successful run in the absence of output masking leads to a two-step training process, where the CNN first rapidly learns to classify digits, and then the SATNet layer learns to solve for Sudoku. The red vertical dotted line demarcates the point at which the training accuracy for visual Sudoku becomes non-zero. Unsuccessful runs typically take a long time for the CNN to classify digits, and never does better than 0% training accuracy at the overall visual Sudoku task.

7.4 MNIST Mapping Problem

The MNIST mapping problem involves a symbolic problem with 20 variables v_i, where the first ten variables are input (i.e. $\mathcal{I} = \{1, \ldots, 10\}$), and the next ten are output (i.e. $\mathcal{O} = \{11, \ldots, 20\}$). But the $v_{\mathcal{I}}$ are not provided directly; instead the input is given as perceptual data in the form of an

MNIST digit image, and the challenge is to map an image of digit i to the variable v_{11+i}. We assume that these variables are boolean (or the probabilistic equivalent, i.e. random variables taking real values in $[0, 1]$), but this should be adapted accordingly to the symbolic representation of a given solver.

There are two distinct sub-problems. The first sub-problem involves classifying an MNIST digit image into $v_1, \ldots, v_{10}$ (using a neural network). The second sub-problem involves learning a bijection (or an equivalent permutation) to $v_{11}, \ldots, v_{20}$ (using a symbolic solver), from which the class of the input image has to be identified. Both sub-problems taken on their own are considered to be easy problems. MNIST digits can be easily classified to 99% test accuracy [232], while permutation groups under equivalence queries are known to be exactly learnable in polynomial time [233]. Hence, we propose that a suitable sanity test for a differentiable symbolic solver is to solve the MNIST mapping problem to an accuracy of 99%. Note that a model that does not have to learn the bijection can circumvent the symbol grounding problem entirely by simply learning the output labels directly. Therefore, the test is strictly intended to be a check for symbol grounding, rather than a grand AI challenge that necessitates the combination of pattern recognition and logical reasoning as in visual Sudoku or PGM.

7.4.1 Configuring SATNet Properly

Surprisingly, some SATNet configurations fail the test, not by a slight margin, but completely (i.e. test accuracy no better than chance; we count them using 12% as a threshold to account for variance). In general, we found that the successful training of SATNet can be very sensitive to specific combinations of hyperparameters, optimizers, and neural architectures. We present four empirical findings using experiments on the MNIST mapping problem. All experiments were ran for 50 training epochs over 10 random seeds to get standard error confidence intervals. The Sudoku CNN, which was the backbone architecture used in the SATNet author's visual Sudoku implementation, is used throughout unless stated otherwise. We evaluate the results by presenting test accuracies with their confidence intervals and the number of complete failures in parenthe-

ses. For comparison, a non-SATNet baseline, which consists of the Sudoku CNN but with the SATNet layer replaced by two fully connected layers (1000 hidden units and ReLU), performs at 72.1±13.3% (3). At a minimum, SATNet should perform better than that, since its raison d'être disappears if it can be bested by equivalent models without logical reasoning capabilities.

Finding 1 *Too little "logic" (i.e. low m) or too much "slack" (i.e. high aux) can cause failure.*

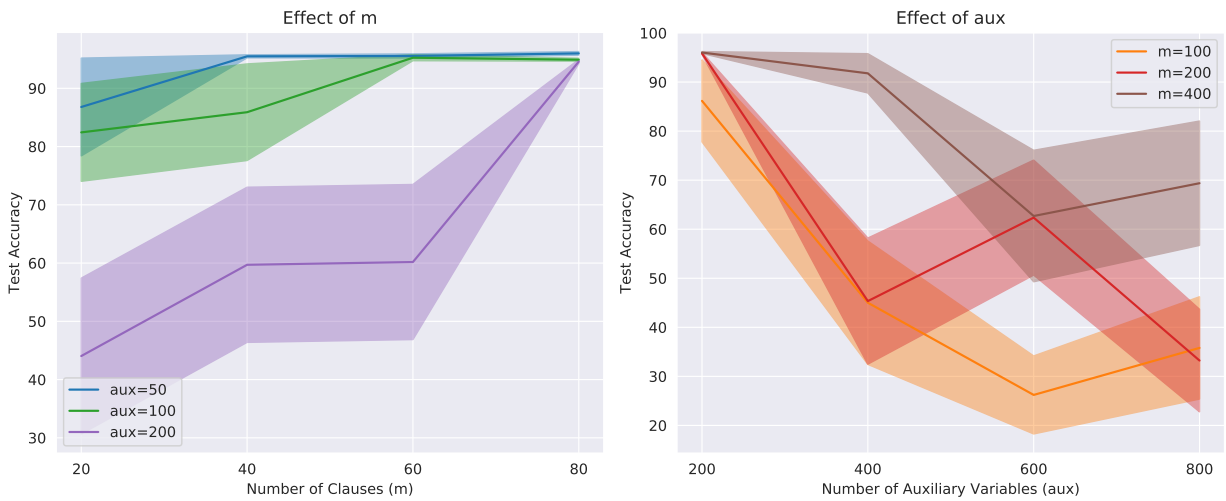

Figure 7.4: Both graphs show test accuracy on the MNIST mapping problem with the shaded interval representing the standard error.

The number of clauses m controls the capacity of SATNet (rank of clause matrix), and we found that it can cause failure or result in terrible test accuracy when it is too low relative to what is needed for the problem. The number of auxiliary variables aux also controls model capacity, but we observed that if it is too high for a given m, it can also cause failure (because most of the clauses end up being filled with meaningless input-independent auxiliary variables). High m or low aux do not affect test accuracy on the MNIST mapping problem, but they affect the amount of compute the SATNet layer uses.

Finding 2 *The backbone layer has to learn at a slower rate than the SATNet layer.*

Table 7.2 shows the effect of differential learning rates between the SATNet and CNN backbone layers on test accuracy and number of failures, using Adam [115] for both layers. If the

Table 7.2: Effects of Different Learning Rates on the SATNet and Backbone Layer on Test Accuracy

SATNet Layer	Backbone Layer Learning Rate		
Learning Rate	$1x10^{-3}$	$1x10^{-4}$	$1x10^{-5}$
$1x10^{-3}$	19.9±8.6% (9)	90.0±8.7% (1)	96.3±0.2% (0)
$1x10^{-4}$	17.4±4.3% (8)	74.6±8.6% (0)	96.1±0.2% (0)
$1x10^{-5}$	14.8±3.6% (9)	31.7±7.1% (5)	72.4±5.3% (0)

backbone layer has a higher learning rate than the SATNet layer, this often leads to failure. Optimal performance is observed when the backbone layer has a lower learning rate than the SATNet layer. Note that this might be counter-intuitive, given that in the label leakage scenario, the backbone CNN had to learn digit recognition before the SATNet layer could learn to solve Sudoku. But without label leakage, having a higher learning rate for the backbone does not make sense because it cannot learn anything useful without the help of the SATNet layer.

Finding 3 *Optimizing the backbone layer with SGD and the SATNet layer with Adam improves both training and test accuracy.*

Instead of simply using different learning rates, swapping the optimizer for the backbone layer with SGD raises test accuracy from 96.3 ±0.2% (0) to 98.6±0.1% (0) (similarly so for training accuracy).

Finding 4 *A sigmoid output layer for the backbone is preferable to softmax.*

Table 7.3: Effects of Different Neural Architectures on Test Accuracy

		Backbone Output Layer	
Architectures	Parameters	Softmax	Sigmoid
LeNet [232]	68,626	63.3±14.1% (4)	98.8±0.0% (0)
Sudoku CNN	860,780	98.6±0.1% (0)	**99.1±0.0% (0)**
ResNet18 [160]	11,723,722	67.6±6.3% (0)	97.2±0.9% (0)

The output of the CNN backbone has to take real values in $[0, 1]$; the SATNet authors' implementation used a softmax output layer to achieve this. We found that a sigmoid output layer strictly

outperforms a softmax layer in all three architectures tested. When softmax is used, we observed that the size of the architecture can result in poor performance if it is too small or too big. In the case where it is too big, it is possible for accuracy to degrade rapidly after reaching its peak (we don't use early stopping). Of the three, the Sudoku CNN appears to be the optimal size.

Every model we tested failed at visual Sudoku, demonstrating the non-triviality of visual Sudoku's grounding problem (since getting even one puzzle in the test set correct necessitates the accurate classification of 36.2 digits on average). However, even for a seemingly easy instance of the symbol grounding problem in the form of MNIST mapping, it was highly non-trivial to find the correct SATNet configuration that would lead to 99% test accuracy. This shows that the current state of SATNet falls significantly short of its promise to integrate logical reasoning in deep learning.

7.5 Conclusion

In this chapter, we presented a detailed analysis of SATNet's capabilities, and provided practical solutions that will help future researchers train SATNet layers in their deep neural networks more effectively. Specifically, we noted that the original experimental setup for visual Sudoku resulted in intermediate label leakage. After removing the intermediate labels, SATNet is found to completely fail at the task of visual Sudoku due to its inability to ground the images of the puzzle digits into the appropriate symbolic representation. We further introduced the MNIST mapping problem as an easier instance of the symbol grounding problem compared to visual Sudoku, and found that SATNet needs to be delicately configured for training to be successful. In particular, the number of auxiliary variables cannot be increased unconditionally with respect to the number of clauses, and the backbone layer has to learn at a slower rate than the SATNet layer.

We can apply what we have learned about SATNet and its failure to solve visual Sudoku's symbol grounding problem more generally to other attempts to integrate logical reasoning into deep learning. Given that logical reasoning modules act at a symbolic level, while generic deep learning modules act at a sub-symbolic level, the interface between these two levels has to involve a

symbol grounding problem. Hence, even if the intermediate label leakage identified in this chapter might be SATNet-specific, we think that explicit tests against simple, interpretable instances of the symbol grounding problem will be fruitful for future researchers in discerning their claims about end-to-end learning (versus end-to-end gradient-based optimization).

In general, we think that the differences between deep learning and logic mirror the ones between continuous and discrete optimization. These differences go far deeper than the superficial lack of derivatives in discrete optimization, and we believe true progress has to come from significantly tighter integrations between deep learning and logic. We are excited that our work brings these differences to the forefront and encourages the community to think more critically about how to go about integrating logical reasoning into deep learning.

Directions for Future Work

Here are some promising directions for future work:

1. **Scalability** Meta-learning methods like hypernetworks or MAML involve an extra level of backpropagation, thus making them significantly more expensive in terms of compute and memory. Improving the efficiency of these methods will help them scale to novel use cases like mobile or edge computing, which might be prohibitive currently.

2. **Optimization** Little is known about the optimization properties of many of these meta-learning methods, and why exactly they are so successful. A deeper understanding of how information is shared across different tasks and settings will yield better methods for extracting invariant and equivariant representations.

3. **Logical Reasoning** A lot of what we consider meta information can be expressed in logical form. For example, grammar is the meta rule that seems to emerge from the arbitrary statistical correlations expressed by a language model. Figuring out how to use auto-generative networks and meta-learning to extract out logical rules from statistical patterns will go a long way towards remedying the current deficits of deep learning systems.

References

[1] Bloomberg, Tesla's elon musk: We're 'summoning the demon' with artificial intelligence, Youtube, 2014.

[2] R. Cellan-Jones, Stephen hawking warns artificial intelligence could end mankind, BBC, 2014.

[3] S. Russell, D. Dewey, and M. Tegmark, "Research priorities for robust and beneficial artificial intelligence," Ai Magazine, vol. 36, no. 4, pp. 105–114, 2015.

[4] R. Kurzweil, The singularity is near: When humans transcend biology. Penguin, 2005.

[5] K. Grace, J. Salvatier, A. Dafoe, B. Zhang, and O. Evans, "When will ai exceed human performance," Evidence from AI Experts. Disponível em:< https://arxiv. org/abs/1705.08807> Acesso em, vol. 24, no. 08, 2017.

[6] C. F. Bolz-Tereick, The first 15 years of pypy — a personal retrospective, PyPy Status Blog, 2018.

[7] K. Thompson, "Reflections on trusting trust," Communications of the ACM, vol. 27, no. 8, pp. 761–763, 1984.

[8] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al., "Imagenet large scale visual recognition challenge," International Journal of Computer Vision, vol. 115, no. 3, pp. 211–252, 2015.

[9] W Xiong, L Wu, F Alleva, J Droppo, X Huang, and A Stolcke, "The microsoft 2017 conversational speech recognition system," arXiv preprint arXiv:1708.06073, 2017.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," arXiv preprint arXiv:1312.5602, 2013.

[11] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., "Mastering the game of go without human knowledge," Nature, vol. 550, no. 7676, p. 354, 2017.

[12] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerry-Ryan, et al., "Natural tts synthesis by conditioning wavenet on mel spectrogram predictions," arXiv preprint arXiv:1712.05884, 2017.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in Advances in neural information processing systems, 2017, pp. 5998–6008.

[14] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," arXiv preprint arXiv:1511.06434, 2015.

[15] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio, "Generating sentences from a continuous space," arXiv preprint arXiv:1511.06349, 2015.

[16] C. Chan, S. Ginosar, T. Zhou, and A. A. Efros, "Everybody dance now," arXiv preprint arXiv:1808.07371, 2018.

[17] N. Anand and P. Huang, Generative modeling for protein structures, 2018.

[18] M. Kudlek, "On the existence of universal finite or pushdown automata," arXiv preprint arXiv:1207.7149, 2012.

[19] G. Cybenko, "Approximation by superpositions of a sigmoidal function," Mathematics of control, signals and systems, vol. 2, no. 4, pp. 303–314, 1989.

[20] K. Hornik, "Approximation capabilities of multilayer feedforward networks," Neural networks, vol. 4, no. 2, pp. 251–257, 1991.

[21] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," Journal of computer and system sciences, vol. 50, no. 1, pp. 132–150, 1995.

[22] E. Jonas and K. P. Kording, "Could a neuroscientist understand a microprocessor?" PLoS computational biology, vol. 13, no. 1, e1005268, 2017.

[23] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in Advances in neural information processing systems, 2013, pp. 3111–3119.

[24] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," arXiv preprint arXiv:1603.01360, 2016.

[25] B. Plank, A. Søgaard, and Y. Goldberg, "Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss," arXiv preprint arXiv:1604.05529, 2016.

[26] X. Yu and N. T. Vu, "Character composition model with convolutional neural networks for dependency parsing on morphologically rich languages," arXiv preprint arXiv:1705.10814, 2017.

[27] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," Evolutionary computation, vol. 10, no. 2, pp. 99–127, 2002.

[28] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," Artificial life, vol. 15, no. 2, pp. 185–212, 2009.

[29] C. Fernando, D. Banarse, M. Reynolds, F. Besse, D. Pfau, M. Jaderberg, M. Lanctot, and D. Wierstra, "Convolution by evolution: Differentiable pattern producing networks," in Proceedings of the Genetic and Evolutionary Computation Conference 2016, ACM, 2016, pp. 109–116.

[30] R Miikkulainen, J Liang, E Meyerson, A Rawal, D Fink, O Francon, B Raju, H Shahrzad, A Navruzyan, N Duffy, et al., "Evolving deep neural networks (2017)," arXiv preprint arXiv:1703.00548, 2017.

[31] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," arXiv preprint arXiv:1611.01578, 2016.

[32] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," arXiv preprint arXiv:1510.00149, 2015.

[33] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," arXiv preprint arXiv:1503.02531, 2015.

[34] C. Bucilua, R. Caruana, and A. Niculescu-Mizil, "Model compression," in Proceedings of the 12th ACM SIGKDD, ACM, 2006, pp. 535–541.

[35] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra, "Pathnet: Evolution channels gradient descent in super neural networks," arXiv preprint arXiv:1701.08734, 2017.

[36] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," arXiv preprint arXiv:1701.06538, 2017.

[37] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," in International Conference on Machine Learning, 2018, pp. 549–558.

[38] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in European Conference on Computer Vision, Springer, 2016, pp. 646–661.

[39] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Training pruned neural networks," arXiv preprint arXiv:1803.03635, 2018.

[40] E. Meyerson and R. Miikkulainen, "Beyond shared hierarchies: Deep multitask learning through soft layer ordering," arXiv preprint arXiv:1711.00108, 2017.

[41] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Smash: One-shot model architecture search through hypernetworks," arXiv preprint arXiv:1708.05344, 2017.

[42] J. Schmidhuber, "Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook," PhD thesis, Technische Universität München, 1987.

[43] J. Schmidhuber, J. Zhao, and N. N. Schraudolph, "Reinforcement learning with self-modifying policies," in Learning to learn, Springer, 1998, pp. 293–309.

[44] J Schmidhuber, "An'introspective'network that can learn to run its own weight change algorithm," in 1993 Third International Conference on Artificial Neural Networks, IET, 1993, pp. 191–194.

[45] Y. Bengio, S. Bengio, and J. Cloutier, Learning a synaptic learning rule. Citeseer, 1990.

[46] S. Bengio, Y. Bengio, and J. Cloutier, "On the search for new learning rules for anns," Neural Processing Letters, vol. 2, no. 4, pp. 26–30, 1995.

[47] S. Thrun and L. Pratt, "Learning to learn: Introduction and overview," in Learning to learn, Springer, 1998, pp. 3–17.

[48] J. Kaiser, "Richard hamming-you and your research," in Simula Research Laboratory, Springer, 2010, pp. 37–60.

[49] S. C. Smithson, G. Yang, W. J. Gross, and B. H. Meyer, "Neural networks designing neural networks: Multi-objective hyper-parameter optimization," in Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on, IEEE, 2016, pp. 1–8.

[50] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," arXiv preprint arXiv:1712.00559, 2017.

[51] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," arXiv preprint arXiv:1802.03268, 2018.

[52] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," arXiv preprint arXiv:1806.09055, 2018.

[53] Y. He and S. Han, "Adc: Automated deep compression and acceleration with reinforcement learning," arXiv preprint arXiv:1802.03494, 2018.

[54] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," arXiv preprint arXiv:1808.05377, 2018.

[55] D. Ha, A. Dai, and Q. V. Le, "Hypernetworks," arXiv preprint arXiv:1609.09106, 2016.

[56] 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 preprint arXiv:1903.10258, 2019.

[57] C. Zhang, M. Ren, and R. Urtasun, "Graph hypernetworks for neural architecture search," arXiv preprint arXiv:1810.05749, 2018.

[58] D. Krueger, C.-W. Huang, R. Islam, R. Turner, A. Lacoste, and A. Courville, "Bayesian hypernetworks," arXiv preprint arXiv:1710.04759, 2017.

[59] K. Ukai, T. Matsubara, and K. Uehara, "Hypernetwork-based implicit posterior estimation and model averaging of cnn," in Asian Conference on Machine Learning, 2018, pp. 176–191.

[60] N. Pawlowski, A. Brock, M. C. Lee, M. Rajchl, and B. Glocker, "Implicit weight uncertainty in neural networks," arXiv preprint arXiv:1711.01297, 2017.

[61] 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 Bayesian Deep Learning Workshop, NeurIPS (Spotlight), vol. 2018, 2018.

[62] L. Deutsch, E. Nijkamp, and Y. Yang, "A generative model for sampling high-performance and diverse weights for neural networks," arXiv preprint arXiv:1905.02898, 2019.

[63] Z. Pan, Y. Liang, J. Zhang, X. Yi, Y. Yu, and Y. Zheng, "Hyperst-net: Hypernetworks for spatio-temporal forecasting," arXiv preprint arXiv:1809.10889, 2018.

[64] F. Shen, S. Yan, and G. Zeng, "Meta networks for neural style transfer," arXiv preprint arXiv:1709.04111, 2017.

[65] S. Klocek, Ł. Maziarka, M. Wołczyk, J. Tabor, M. Śmieja, and J. Nowak, "Hypernetwork functional image representation," arXiv preprint arXiv:1902.10404, 2019.

[66] J. Serrà, S. Pascual, and C. Segura, "Blow: A single-scale hyperconditioned flow for non-parallel raw-audio voice conversion," arXiv preprint arXiv:1906.00794, 2019.

[67] E. Meyerson and R. Miikkulainen, "Modular universal reparameterization: Deep multi-task learning across diverse domains," arXiv preprint arXiv:1906.00097, 2019.

[68] J. von Oswald, C. Henning, J. Sacramento, and B. F. Grewe, "Continual learning with hypernetworks," arXiv preprint arXiv:1906.00695, 2019.

[69] J. Suarez, "Language modeling with recurrent highway hypernetworks," in Advances in neural information processing systems, 2017, pp. 3267–3276.

[70] N. Ratzlaff and L. Fuxin, "Hypergan: A generative model for diverse, performant neural networks," arXiv preprint arXiv:1901.11058, 2019.

[71] A. Kristiadi and A. Fischer, "Predictive uncertainty quantification with compound density networks," arXiv preprint arXiv:1902.01080, 2019.

[72] J. Lorraine and D. Duvenaud, "Stochastic hyperparameter optimization through hypernetworks," arXiv preprint arXiv:1802.09419, 2018.

[73] Z. Sun, M. Ozay, and T. Okatani, "Hypernetworks with statistical filtering for defending adversarial examples," arXiv preprint arXiv:1711.01791, 2017.

[74] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas, "Learning to learn by gradient descent by gradient descent," in Advances in Neural Information Processing Systems, 2016, pp. 3981–3989.

[75] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning," in International Conference on Learning Representations, 2018.

[76] P. Ramachandran, B. Zoph, and Q. V. Le, "Swish: A self-gated activation function," arXiv preprint arXiv:1710.05941, 2017.

[77] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in Advances in neural information processing systems, 2014, pp. 2672–2680.

[78] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," arXiv preprint arXiv:1703.03400, 2017.

[79] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al., "Overcoming catastrophic forgetting in neural networks," Proceedings of the national academy of sciences, vol. 114, no. 13, pp. 3521–3526, 2017.

[80] T. Gabor, S. Illium, A. Mattausch, L. Belzner, and C. Linnhoff-Popien, "Self-replication in neural networks," in Artificial Life Conference Proceedings, MIT Press, 2019, pp. 424–431.

[81] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in Proceedings of the thirteenth international conference on artificial intelligence and statistics, 2010, pp. 249–256.

[82] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in Proceedings of the IEEE international conference on computer vision, 2015, pp. 1026–1034.

[83] P.-W. Wang, P. L. Donti, B. Wilder, and Z. Kolter, "Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver," arXiv preprint arXiv:1905.12149, 2019.

[84] S. Harnad, "The symbol grounding problem," Physica D: Nonlinear Phenomena, vol. 42, no. 1-3, pp. 335–346, 1990.

[85] J. Von Neumann and A. W. Burks, "Theory of self-reproducing automata," in, Urbana: University of Illinois Press, 1966, p. 8.

[86] D. Hofstadter, Gödel, Escher, Bach: an Eternal Golden Braid. New York: Vintage Books, 1980.

[87] W. contributors, Quine (computing) — wikipedia, the free encyclopedia, [Online; accessed 5-February-2018], 2018.

[88] Y. LeCun and C. Cortes, "The mnist database of handwritten digits," 1998.

[89] N. Drake, "Why male dark fishing spiders die spontaneously after sex," Wired, 2013.

[90] D. Held, S. Thrun, and S. Savarese, "Learning to track at 100 fps with deep regression networks," in Computer Vision – ECCV 2016, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., Cham: Springer International Publishing, 2016, pp. 749–765, ISBN: 978-3-319-46448-0.

[91] Gulshan, Varun and Peng, Lily and Coram, Marc and Stumpe, Martin C. and Wu, Derek and Narayanaswamy, Arunachalam and Venugopalan, Subhashini and Widner, Kasumi and Madams, Tom and Cuadros, Jorge, and Kim, Ramasamy and Raman, Rajiv and Nelson, Philip C. and Mega, Jessica L. and Webster, Dale R., "Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs," JAMA, vol. 316, no. 22, pp. 2402–2410, 2016.

[92] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," arXiv preprint arXiv:1609.03499, 2016.

[93] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, et al., "Starcraft ii: A new challenge for reinforcement learning," arXiv preprint arXiv:1708.04782, 2017.

[94] M. Marshall, "First life: The search for the first replicator," New Scientist, vol. Issue 2825, 2011.

[95] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," IEEE transactions on pattern analysis and machine intelligence, vol. 35, no. 8, pp. 1798–1828, 2013.

[96] V. Zykov, E. Mytilinaios, B. Adams, and H. Lipson, "Robotics: Self-reproducing machines," Nature, vol. 435, pp. 163–164, 2005.

[97] G. P. Thompson. (1999). The quine page, (visited on 03/07/2018).

[98] Y. Endoh. (2017). Quine relay, (visited on 03/07/2018).

[99] T. Wang, R. Sha, R. Dreyfus, M. E. Leunissen, C. Maass, D. J. Pine, P. M. Chaikin, and N. C. Seeman, "Self-replication of information-bearing nanoscale patterns," Nature, vol. 478, no. 7368, p. 225, 2011.

[100] J. Breivik, "Self-organization of template-replicating polymers and the spontaneous rise of genetic information," Entropy, vol. 3, no. 4, pp. 273–279, 2001.

[101] M. Denil, B. Shakibi, L. Dinh, N. De Freitas, et al., "Predicting parameters in deep learning," in Advances in neural information processing systems, 2013, pp. 2148–2156.

[102] J. Schmidhuber, "Learning to control fast-weight memories: An alternative to dynamic recurrent networks," Neural Computation, vol. 4, no. 1, pp. 131–139, 1992.

[103] D. Ha, A. M. Dai, and Q. V. Le, "Hypernetworks," International Conference on Learning Representations, 2017.

[104] J. Schmidhuber, "A 'self-referential'weight matrix," in ICANN'93, Springer, 1993, pp. 446–450.

[105] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," International Conference on Learning Representations, 2016.

[106] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra, "Draw: A recurrent neural network for image generation," Proceedings of the 32nd International Conference on Machine Learning, 2015.

[107] K. O. Stanley, D. D'Ambrosio, and J. Gauci, "A hypercube-based indirect encoding for evolving large-scale neural networks," Artificial Life, vol. 15(2), pp. 185–212, 2009.

[108] W. contributors, Self-replication — wikipedia, the free encyclopedia, [Online; accessed 5-March-2018], 2017.

[109] W. B. Johnson and J. Lindenstrauss, "Extensions of lipschitz mappings into a hilbert space," Contemporary mathematics, vol. 26, no. 189-206, p. 1, 1984.

[110] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in Advances in neural information processing systems, 2008, pp. 1177–1184.

[111] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," Neurocomputing, vol. 70, no. 1-3, pp. 489–501, 2006.

[112] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in Advances in Neural Information Processing Systems, 2017, pp. 972–981.

[113] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," arXiv preprint arXiv:1712.06567, 2017.

[114] T. Salimans, J. Ho, X. Chen, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," arXiv preprint arXiv:1703.03864, 2017.

[115] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[116] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," Journal of Machine Learning Research, vol. 12, no. Jul, pp. 2121–2159, 2011.

[117] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural networks for machine learning, vol. 4, no. 2, pp. 26–31, 2012.

[118] B. Adams and H. Lipson, "A universal framework for analysis of self-replication phenomena," Entropy, vol. 11, pp. 295–325, 2009.

[119] L. S. Penrose, "Self-reproducing machines," Scientific American, vol. 200, pp. 105–112, 1959.

[120] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," arXiv preprint arXiv:1602.01783, 2016.

[121] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in Advances in neural information processing systems, 2016, pp. 3675–3683.

[122] Q. Zhang and S.-C. Zhu, "Visual interpretability for deep learning: A survey," arXiv preprint arXiv:1802.00614, 2018.

[123] S. Chakraborty, R. Tomsett, R. Raghavendra, D. Harborne, M. Alzantot, F. Cerutti, M. Srivastava, A. Preece, S. Julier, R. M. Rao, et al., "Interpretability of deep learning models: A survey of results," in IEEE Smart World Congress 2017 Workshop: DAIS, 2017.

[124] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 815–823.

[125] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in Proceedings of the 2013 Conference of the Association for Computational Linguistics, 2013, pp. 746–751.

[126] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in Proceedings of the 2014 EMNLP, 2014, pp. 1532–1543.

[127] Y. Li, J. Yosinski, J. Clune, H. Lipson, and J. Hopcroft, "Convergent learning: Do different neural networks learn the same representations?" In Feature Extraction: Modern Questions and Challenges, 2015, pp. 196–212.

[128] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, "Programmatically interpretable reinforcement learning," arXiv preprint arXiv:1804.02477, 2018.

[129] C. Dann, L. Li, W. Wei, and E. Brunskill, "Policy certificates: Towards accountable reinforcement learning," arXiv preprint arXiv:1811.03056, 2018.

[130] Y. Zha, Y. Li, S. Gopalakrishnan, B. Li, and S. Kambhampati, "Recognizing plans by learning embeddings from observed action distributions," in Proceedings of the 17th AAMAS, International Foundation for Autonomous Agents and Multiagent Systems, 2018, pp. 2153–2155.

[131] D. Ashlock and C. Lee, "Agent-case embeddings for the analysis of evolved systems.,"

[132] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," arXiv preprint arXiv:1609.03499, 2016.

[133] C. Vondrick, H. Pirsiavash, and A. Torralba, "Generating videos with scene dynamics," in Advances In Neural Information Processing Systems, 2016, pp. 613–621.

[134] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," Journal of machine learning research, vol. 3, no. Feb, pp. 1137–1155, 2003.

[135] T. Xiao, J. Hong, and J. Ma, "Dna-gan: Learning disentangled representations from multi-attribute images," arXiv preprint arXiv:1711.05415, 2017.

[136] O. Chang and H. Lipson, "Neural network quine," Artificial life, vol. 30, pp. 234–241, 2018.

[137] C. M. Bishop, "Bayesian neural networks," Journal of the Brazilian Computer Society, vol. 4, no. 1, 1997.

[138] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," arXiv preprint arXiv:1312.6114, 2013.

[139] C. Louizos and M. Welling, "Multiplicative normalizing flows for variational bayesian neural networks," arXiv preprint arXiv:1703.01961, 2017.

[140] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," IEEE transactions on systems, man, and cybernetics, vol. SMC-13, no. 5, pp. 834–846, 1983.

[141] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," arXiv preprint arXiv:1606.01540, 2016.

[142] C. Li, H. Farkhoor, R. Liu, and J. Yosinski, "Measuring the intrinsic dimension of objective landscapes," arXiv preprint arXiv:1804.08838, 2018.

[143] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," arXiv preprint arXiv:1511.07289, 2015.

[144] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, p. 529, 2015.

[145] K. Sohn, H. Lee, and X. Yan, "Learning structured output representation using deep conditional generative models," in Advances in Neural Information Processing Systems, 2015, pp. 3483–3491.

[146] J. E. Hopcroft and R. M. Karp, "An n$\hat{5}$/2 algorithm for maximum matchings in bipartite graphs," SIAM Journal on computing, vol. 2, no. 4, pp. 225–231, 1973.

[147] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning," in Advances in Neural Information Processing Systems, 2017, pp. 4148–4158.

[148] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," in International Conference on Learning Representations, 2018.

[149] D. E. Rumelhart, G. E. Hintont, and R. J. Williams, "Learning representations by back-propagating errors," NATURE, vol. 323, p. 9, 1986.

[150] H. Zhang, Y. N. Dauphin, and T. Ma, "Fixup initialization: Residual learning without normalization," arXiv preprint arXiv:1901.09321, 2019.

[151] S. Laue, M. Mitterreiter, and J. Giesen, "Computing higher order derivatives of matrix and tensor expressions," in Advances in Neural Information Processing Systems, 2018, pp. 2755–2764.

[152] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," arXiv preprint arXiv:1312.6120, 2013.

[153] I. Balazevic, C. Allen, and T. M. Hospedales, "Hypernetwork knowledge graph embeddings," arXiv preprint arXiv:1808.07018, 2018.

[154] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," arXiv preprint arXiv:1412.6806, 2014.

[155] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," arXiv preprint arXiv:1704.04861, 2017.

[156] Y. Wu and K. He, "Group normalization," in Proceedings of ECCV, 2018, pp. 3–19.

[157] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," Neural computation, vol. 18, no. 7, pp. 1527–1554, 2006.

[158] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in Advances in neural information processing systems, 2007, pp. 153–160.

[159] J. Koutnik, F. Gomez, and J. Schmidhuber, "Evolving neural networks in compressed weight space," in Proceedings of the 12th annual conference on Genetic and evolutionary computation, ACM, 2010, pp. 619–626.

[160] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[161] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in neural information processing systems, 2012, pp. 1097–1105.

[162] Y. Wang, R. Skerry-Ryan, D. Stanton, Y. Wu, R. J. Weiss, N. Jaitly, Z. Yang, Y. Xiao, Z. Chen, S. Bengio, et al., "Tacotron: Towards end-to-end speech synthesis," arXiv preprint arXiv:1703.10135, 2017.

[163] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland, et al., "Improved protein structure prediction using potentials from deep learning," Nature, pp. 1–5, 2020.

[164] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," IEEE transactions on evolutionary computation, vol. 1, no. 1, pp. 67–82, 1997.

[165] J. Clune, "Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence," arXiv preprint arXiv:1905.10985, 2019.

[166] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," arXiv preprint arXiv:1410.5401, 2014.

[167] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap, "Meta-learning with memory-augmented neural networks," in International conference on machine learning, 2016, pp. 1842–1850.

[168] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra, et al., "Matching networks for one shot learning," in Advances in Neural Information Processing Systems, 2016, pp. 3630–3638.

[169] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in Advances in neural information processing systems, 2017, pp. 4077–4087.

[170] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap, "A simple neural network module for relational reasoning," in Advances in neural information processing systems, 2017, pp. 4967–4976.

[171] R. Caruana, "Learning many related tasks at the same time with backpropagation," in Advances in neural information processing systems, 1995, pp. 657–664.

[172] M. A. Jamal and G.-J. Qi, "Task agnostic meta-learning for few-shot learning," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 11 719–11 727.

[173] M. Khodak, M.-F. Balcan, and A. Talwalkar, "Provable guarantees for gradient-based meta-learning," arXiv preprint arXiv:1902.10644, 2019.

[174] A. Rajeswaran, C. Finn, S. M. Kakade, and S. Levine, "Meta-learning with implicit gradients," in Advances in Neural Information Processing Systems, 2019, pp. 113–124.

[175] M. Yin, G. Tucker, M. Zhou, S. Levine, and C. Finn, "Meta-learning without memorization," in International Conference on Learning Representations, 2020.

[176] S. Guiroy, V. Verma, and C. Pal, "Towards understanding generalization in gradient-based meta-learning," arXiv preprint arXiv:1907.07287, 2019.

[177] R. Caruana, "Multitask learning," Machine learning, vol. 28, no. 1, pp. 41–75, 1997.

[178] J. Baxter, "A bayesian/information theoretic model of learning to learn via multiple task sampling," Machine learning, vol. 28, no. 1, pp. 7–39, 1997.

[179] Z. Li, F. Zhou, F. Chen, and H. Li, "Meta-sgd: Learning to learn quickly for few-shot learning," arXiv preprint arXiv:1707.09835, 2017.

[180] E. Park and J. B. Oliva, "Meta-curvature," in Advances in NeurIPS, 2019, pp. 3309–3319.

[181] H. B. Lee, T. Nam, E. Yang, and S. J. Hwang, "Meta dropout: Learning to perturb latent features for generalization," in International Conference on Learning Representations, 2020.

[182] S. Flennerhag, A. A. Rusu, R. Pascanu, F. Visin, H. Yin, and R. Hadsell, "Meta-learning with warped gradient descent," in International Conference on Learning Representations, 2020.

[183] A. Antoniou, H. Edwards, and A. Storkey, "How to train your maml," arXiv preprint arXiv:1810.09502, 2018.

[184] L. M. Zintgraf, K. Shiarlis, V. Kurin, K. Hofmann, and S. Whiteson, "Fast context adaptation via meta-learning," arXiv preprint arXiv:1810.03642, 2018.

[185] A. Antoniou and A. J. Storkey, "Learning to learn by self-critique," in Advances in Neural Information Processing Systems, 2019, pp. 9936–9946.

[186] A. A. Rusu, D. Rao, J. Sygnowski, O. Vinyals, R. Pascanu, S. Osindero, and R. Hadsell, "Meta-learning with latent embedding optimization," arXiv preprint arXiv:1807.05960, 2018.

[187] A. Nichol, J. Achiam, and J. Schulman, "On first-order meta-learning algorithms," arXiv preprint arXiv:1803.02999, 2018.

[188] H.-Y. Tseng, Y.-W. Chen, Y.-H. Tsai, S. Liu, Y.-Y. Lin, and M.-H. Yang, Dropgrad: Gradient dropout regularization for meta-learning, 2020.

[189] Z. Kang, K. Grauman, and F. Sha, "Learning with whom to share in multi-task feature learning," in Proceedings of the 28th International Conference on Machine Learning, ser. ICML'11, Madison, WI, USA: Omnipress, 2011, 521–528, ISBN: 9781450306195.

[190] S. Ruder, "An overview of multi-task learning in deep neural networks," arXiv preprint arXiv:1706.05098, 2017.

[191] H. B. Lee, E. Yang, and S. J. Hwang, "Deep asymmetric multi-task feature learning," arXiv preprint arXiv:1708.00260, 2017.

[192] S. Liu, Y. Liang, and A. Gitter, "Loss-balanced task weighting to reduce negative transfer in multi-task learning," in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, 2019, pp. 9977–9978.

[193] Z. Chen, V. Badrinarayanan, C.-Y. Lee, and A. Rabinovich, "Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks," arXiv preprint arXiv:1711.02257, 2017.

[194] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 7482–7491.

[195] O. Sener and V. Koltun, "Multi-task learning as multi-objective optimization," in Advances in Neural Information Processing Systems, 2018, pp. 527–538.

[196] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al., "Hybrid computing using a neural network with dynamic external memory," Nature, vol. 538, no. 7626, pp. 471–476, 2016.

[197] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," arXiv preprint arXiv:1502.03167, 2015.

[198] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, "The Caltech-UCSD Birds-200-2011 Dataset," California Institute of Technology, Tech. Rep. CNS-TR-2011-001, 2011.

[199] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al., "Deep speech 2: End-to-end speech recognition in english and mandarin," in International conference on machine learning, 2016, pp. 173–182.

[200] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., "Mastering the game of go with deep neural networks and tree search," nature, vol. 529, no. 7587, p. 484, 2016.

[201] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell, "Agent57: Outperforming the atari human benchmark," arXiv preprint arXiv:2003.13350, 2020.

[202] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "First return then explore," arXiv preprint arXiv:2004.12919, 2020.

[203] G. Marcus, "The next decade in ai: Four steps towards robust artificial intelligence," arXiv preprint arXiv:2002.06177, 2020.

[204] F. Chollet, "The measure of intelligence," arXiv preprint arXiv:1911.01547, 2019.

[205] J. S. B. Evans, "Heuristic and analytic processes in reasoning," British Journal of Psychology, vol. 75, no. 4, pp. 451–468, 1984.

[206] D. Kahneman, Thinking, fast and slow. Macmillan, 2011.

[207] Y. Bengio, From system 1 deep learning to system 2 deep learning, Conference on Neural Information Processing Systems, 2019.

[208] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, et al., "Solving rubik's cube with a robot hand," arXiv preprint arXiv:1910.07113, 2019.

[209] Z. Hu, X. Ma, Z. Liu, E. Hovy, and E. Xing, "Harnessing deep neural networks with logic rules," arXiv preprint arXiv:1603.06318, 2016.

[210] T. Rocktäschel and S. Riedel, "End-to-end differentiable proving," in Advances in Neural Information Processing Systems, 2017, pp. 3788–3800.

[211] N. Cingillioglu and A. Russo, "Deeplogic: Towards end-to-end differentiable logical reasoning," arXiv preprint arXiv:1805.07433, 2018.

[212] R. Evans and E. Grefenstette, "Learning explanatory rules from noisy data," Journal of Artificial Intelligence Research, vol. 61, pp. 1–64, 2018.

[213] L. Serafini and A. d. Garcez, "Logic tensor networks: Deep learning and logical reasoning from data and knowledge," arXiv preprint arXiv:1606.04422, 2016.

[214] G. Sourek, V. Aschenbrenner, F. Zelezny, and O. Kuzelka, "Lifted relational neural networks," arXiv preprint arXiv:1508.05128, 2015.

[215] E. van Krieken, E. Acar, and F. van Harmelen, "Analyzing differentiable fuzzy logic operators," arXiv preprint arXiv:2002.06100, 2020.

[216] M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rolínek, "Differentiation of blackbox combinatorial solvers," arXiv preprint arXiv:1912.02175, 2019.

[217] M. Rolínek, P. Swoboda, D. Zietlow, A. Paulus, V. Musil, and G. Martius, "Deep graph matching via blackbox differentiation of combinatorial solvers," arXiv preprint arXiv:2003.11657, 2020.

[218] S. Tschiatschek, A. Sahin, and A. Krause, "Differentiable submodular maximization," arXiv preprint arXiv:1803.01785, 2018.

[219] M. Blondel, O. Teboul, Q. Berthet, and J. Djolonga, "Fast differentiable sorting and ranking," arXiv preprint arXiv:2002.08871, 2020.

[220] B. Amos and J. Z. Kolter, "Optnet: Differentiable optimization as a layer in neural networks," in Proceedings of the 34th International Conference on Machine Learning-Volume 70, JMLR. org, 2017, pp. 136–145.

[221] R. Palm, U. Paquet, and O. Winther, "Recurrent relational networks," in Advances in Neural Information Processing Systems, 2018, pp. 3368–3378.

[222] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a sat solver from single-bit supervision," arXiv preprint arXiv:1802.03685, 2018.

[223] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt, "Deepproblog: Neural probabilistic logic programming," in Advances in Neural Information Processing Systems, 2018, pp. 3749–3759.

[224] F. Yang, Z. Yang, and W. W. Cohen, "Differentiable learning of logical rules for knowledge base reasoning," in Advances in Neural Information Processing Systems, 2017, pp. 2319–2328.

[225] A. Cropper, S. Dumančić, and S. H. Muggleton, "Turning 30: New ideas in inductive logic programming," arXiv preprint arXiv:2002.11002, 2020.

[226] W. W. Cohen, "Tensorlog: A differentiable deductive database," arXiv preprint arXiv:1605.06523, 2016.

[227] C. Zhang, F. Gao, B. Jia, Y. Zhu, and S.-C. Zhu, "Raven: A dataset for relational and analogical visual reasoning," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 5317–5327.

[228] D. G. Barrett, F. Hill, A. Santoro, A. S. Morcos, and T. Lillicrap, "Measuring abstract reasoning in neural networks," arXiv preprint arXiv:1807.04225, 2018.

[229] S. Hu, Y. Ma, X. Liu, Y. Wei, and S. Bai, "Hierarchical rule induction network for abstract visual reasoning," arXiv preprint arXiv:2002.06838, 2020.

[230] P. Norvig, Solving every sudoku puzzle, http://norvig.com/sudoku.html, 2006.

[231] K. Park, Can convolutional neural networks crack sudoku puzzles? https://github.com/Kyubyong/sudoku, 2018.

[232] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

[233] V. Arvind and N. Vinodchandran, "The complexity of exactly learning algebraic concepts," in International Workshop on Algorithmic Learning Theory, Springer, 1996, pp. 100–112.

[234] T. Deleu, T. Würfl, M. Samiei, J. P. Cohen, and Y. Bengio, Torchmeta: A Meta-Learning library for PyTorch, Available at: https://github.com/tristandeleu/pytorch-meta, 2019.

[235] user265554 (https://puzzling.stackexchange.com/users/16477/user265554), Complete the sequence, Puzzling Stack Exchange, URL:https://puzzling.stackexchange.com/questions/22 the-sequence (version: 2020-05-08), 2015. eprint: https://puzzling.stackexchange.com/questions/22495/complete-the-sequence.

Appendix A: Supplementary Information for Chapter 4

A.1 Re-using Hypernet Weights

A.1.1 For Mainnet Weights of the Same Size

For model compression or weight-sharing purposes, different parts of the mainnet might be generated by the same hypernet function. This will cause some assumptions of independence in our analysis to be invalid. Consider the example of the same hypernet being used to generate multiple different mainnet weight layers of the same size, i.e. $H[t]^{i_{t+1}}_{i_t k} = H[t+1]^{i_{t+2}}_{i_{t+1} k}, \mathrm{d}_{i_{t+1}} = \mathrm{d}_{i_{t+2}} = \mathrm{d}_{i_t}$. Then, $x[t+1]^{i_{t+1}} = H[t]^{i_{t+1}}_{i_t k_t} e[t]^{k_t} x[t]^{i_t} \not\perp W[t+1]^{i_{t+2}}_{i_{t+1}} = H[t+1]^{i_{t+2}}_{i_{t+1} k} e[t+1]^{k_{t+1}}$.

The relaxation of some of these independence assumptions does not always prove to be a big problem in practice, because the correlations introduced by repeated use of H can be minimized with the use of flat distributions like the uniform distribution. It can even be helpful, since the re-use of the same hypernet for different layers causes the gradient flowing through the hypernet output layer to be the sum of the gradients from the weights of these layers: $\frac{\partial L}{\partial h(e)^k} = \sum_t \frac{\partial L}{\partial W[t]^{i_{t+1}}_{i_t}} H^{i_{t+1}}_{i_t k}$, thus combating the shrinking effect.

A.1.2 For Mainnet Weights of Different Sizes

Similar reasoning applies if the same hypernet was used to generate differently sized subsets of weights in the mainnet. However, we encourage avoiding this kind of hypernet architecture design if not otherwise essential, since it will complicate the initialization formulae listed in Table 4.1.

Consider [55]'s hypernetwork architecture. Their two-layer hypernet generated weight chunks of size (K, n, n) for a main convolutional network where $K = 16$ was found to be the highest common factor among the size of mainnet layers, and $n^2 = 9$ was the size of the receptive field.

We simplify the presentation by writing i for i_t, j for j_t, k for $k_{t,m}$, and l for $l_{t,m}$.

$$W[t]^i_j = \begin{cases} H^{i(\text{mod } K)}_k \alpha[t][j + \lfloor \frac{i}{K} \rfloor \mathrm{d}_j]^k + \beta^{i(\text{mod } K)} & \text{if } i \text{ is divisible by } K \\ \delta_{j(\text{mod } K) j(\text{mod } K)} \left[H^{j(\text{mod } K)}_k \alpha[t][i + \lfloor \frac{j}{K} \rfloor \mathrm{d}_i]^k + \beta^{j(\text{mod } K)} \right] & \text{if } j \text{ is divisible by } K \end{cases}$$

$$\alpha[t][m_t]^k = G[t][m_t]^k_l e[t][m_t]^l + \gamma[t][m_t]^k \tag{A.1}$$

Because the output layer (H, β) in the hypernet was re-used to generate mainnet weight matrices of different sizes (i.e. in general, $i_t \neq i_{t+1}, j_t \neq j_{t+1}$), G effectively becomes the output layer that we want to be considering for hyperfan-in and hyperfan-out initialization.

Hence, to achieve fan-in in the mainnet $\mathrm{Var}(W[t]^i_j) = \frac{1}{\mathrm{d}_j}$, we have to use fan-in init for H (i.e. $\mathrm{Var}(H^{i(\text{mod } K)}_k) = \frac{1}{\mathrm{d}_k} \neq \frac{1}{\mathrm{d}_j \mathrm{d}_k \mathrm{Var}(e[t][m_t]^l)}$), and hyperfan-in init for G (i.e. $\mathrm{Var}(G[t][m_t]^k_l) = \frac{1}{\mathrm{d}_j \mathrm{d}_l \mathrm{Var}(e[t][m_t]^l)}$).

Analogously, to achieve fan-out in the mainnet $\mathrm{Var}(W[t]^i_j) = \frac{1}{\mathrm{d}_i}$, we have to use fan-in init for H (i.e. $\mathrm{Var}(H^{i(\text{mod } K)}_k) = \frac{1}{\mathrm{d}_k} \neq \frac{1}{\mathrm{d}_i \mathrm{d}_k \mathrm{Var}(e[t][m_t]^l)}$), and hyperfan-out init for G (i.e. $\mathrm{Var}(G[t][m_t]^k_l) = \frac{1}{\mathrm{d}_i \mathrm{d}_l \mathrm{Var}(e[t][m_t]^l)}$).

A.2 More Experimental Details

A.2.1 Feedforward Networks on MNIST

The networks were trained on MNIST for 30 epochs with batch size 10 using a learning rate of 0.0005 for the hypernets and 0.01 for the classical network. The hypernets had one linear layer with embeddings of size 50 and different hidden layers in the mainnet were all generated by the same hypernet output layer with a different embedding, which was randomly sampled from $\mathcal{U}(-\sqrt{3}, \sqrt{3})$ and fixed. We use the mean cross entropy loss for training, but the summed cross entropy loss for testing.

We show activation and gradient plots for two cases: (i) the hypernet generates only the weights of the mainnet, and (ii) the hypernet generates both the weights and biases of the mainnet. (i) covers Figures 4.3, 4.1, A.1, A.2, A.3, A.4, A.5, A.6, 4.2, A.7, A.8, A.9, and A.10. (ii) covers Figures A.11, A.12, A.13, A.14, A.15, A.16, A.17, A.18, A.19, A.20, A.21, A.22, and A.23.

The activations and gradients in our plots were calculated by averaging across a fixed held-out set of 300 examples drawn randomly from the test set.

In Figures 4.1, A.2, A.3, A.5, A.6, A.7, A.8, A.10, A.12, A.14, A.15, A.17, A.18, A.20, A.21, and A.23, the y axis shows the number of activations/gradients, while the x axis shows the value of the activations/gradients. The value of activations/gradients from the hypernet output layer correspond to the value of mainnet weights.

In Figures 4.2, A.1, A.4, A.9, A.13, A.16, A.19, and A.22, the y axis shows the mean value of the activations/gradients, while each increment on the x axis corresponds to a measurement that was taken every 1000 training batches, with the bars denoting one standard deviation away from the mean.

Hypernet Generates Only the Mainnet Weights

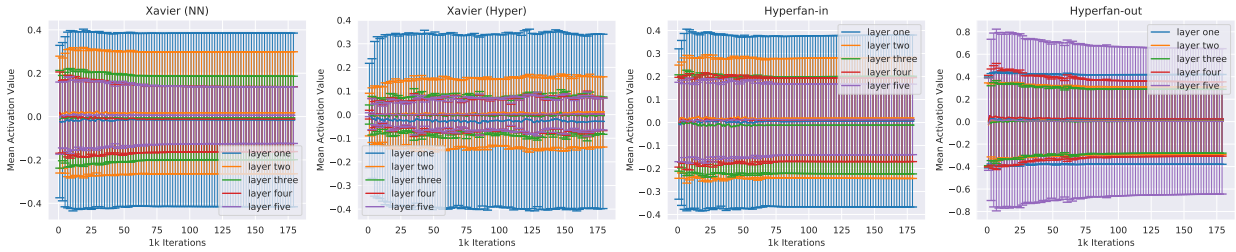

Figure A.1: Evolution of Mainnet Activations during Training on MNIST.

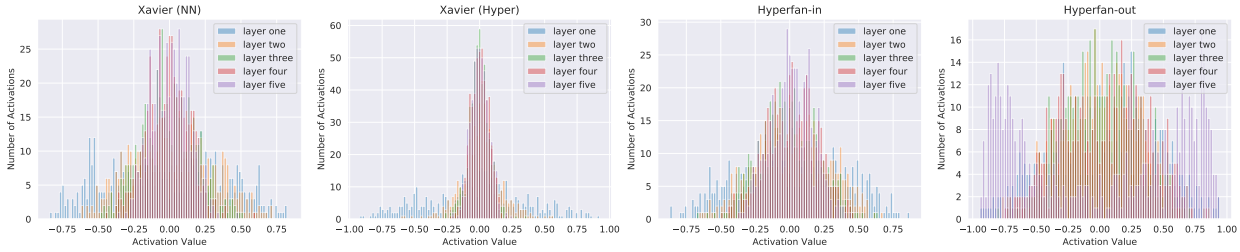

Figure A.2: Mainnet Activations at the End of Training on MNIST.

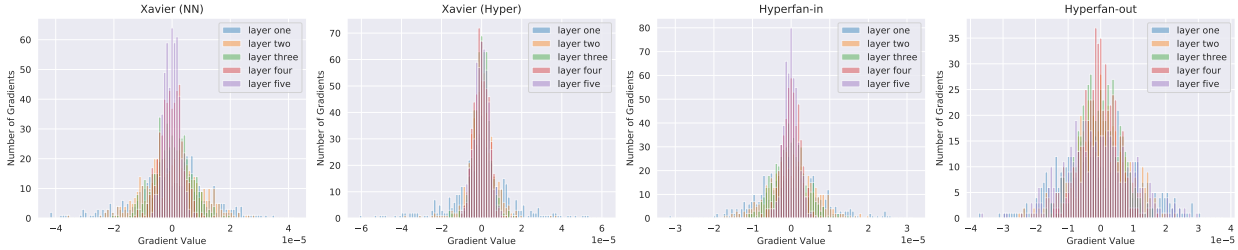

Figure A.3: Mainnet Gradients before the Start of Training on MNIST.

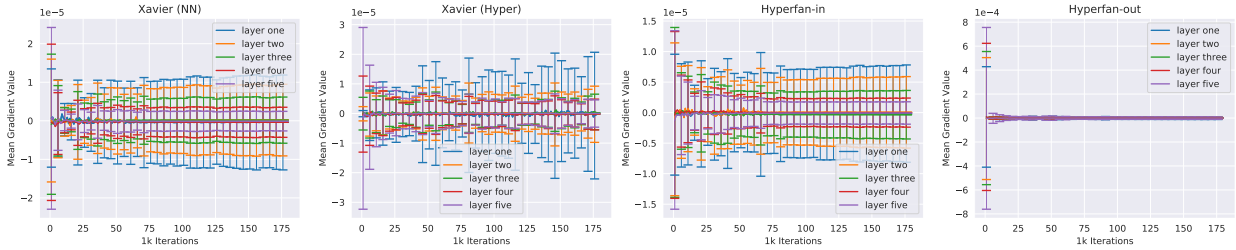

Figure A.4: Evolution of Mainnet Gradients during Training on MNIST.

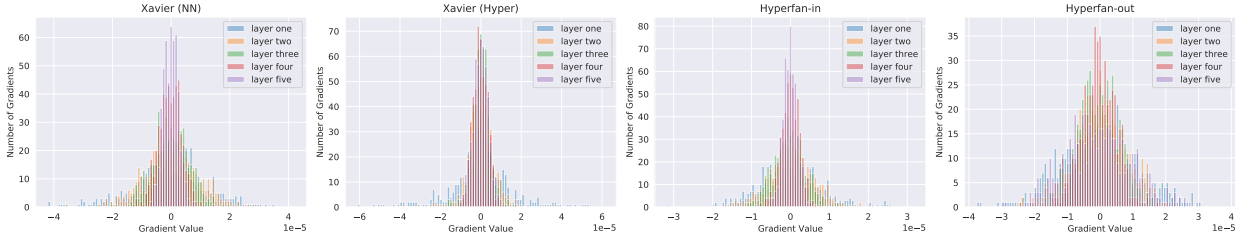

Figure A.5: Mainnet Gradients at the End of Training on MNIST.

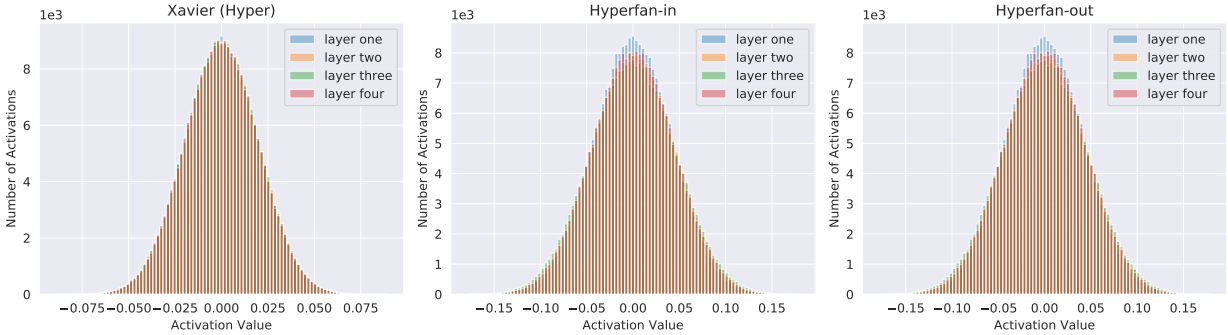

Figure A.6: Hypernet Output Layer Activations before the Start of Training on MNIST.

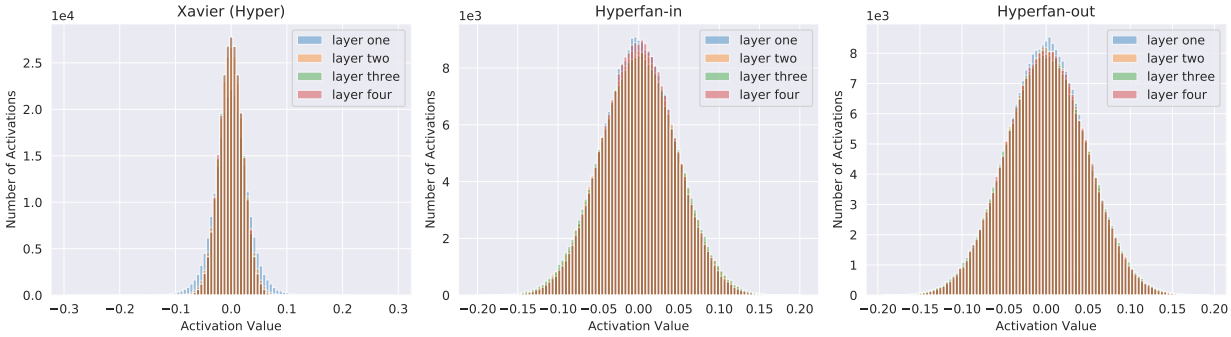

Figure A.7: Hypernet Output Layer Activations at the End of Training on MNIST.

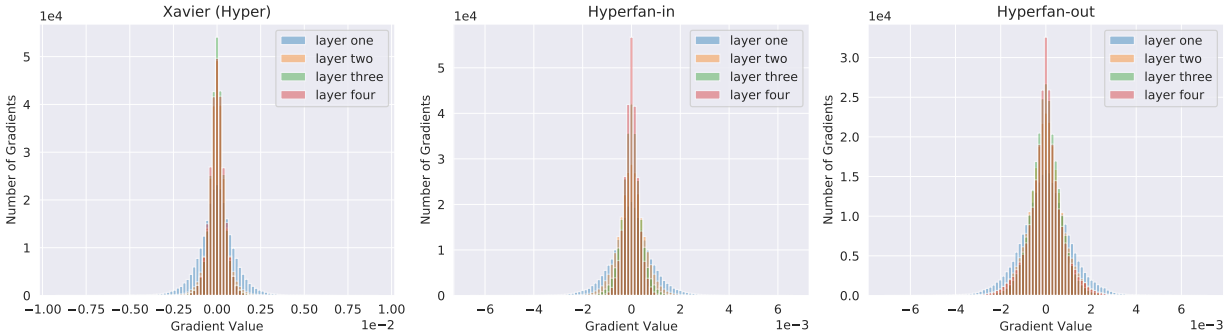

Figure A.8: Hypernet Output Layer Gradients before the Start of Training on MNIST.

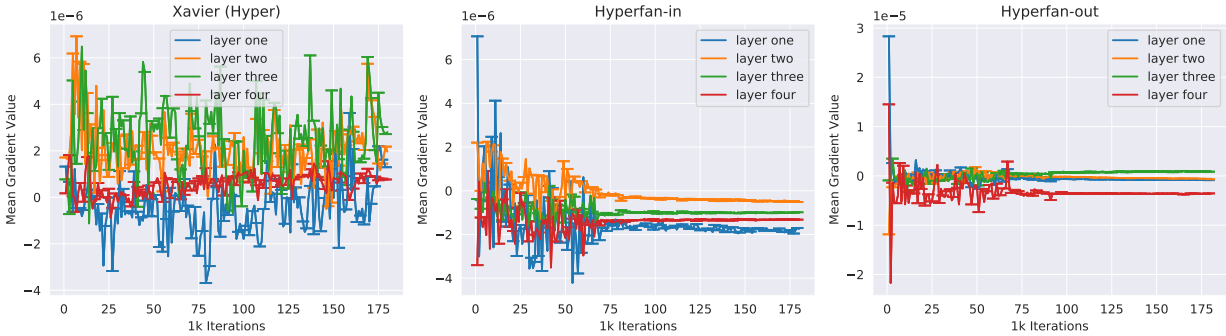

Figure A.9: Evolution of Hypernet Output Layer Gradients during Training on MNIST.

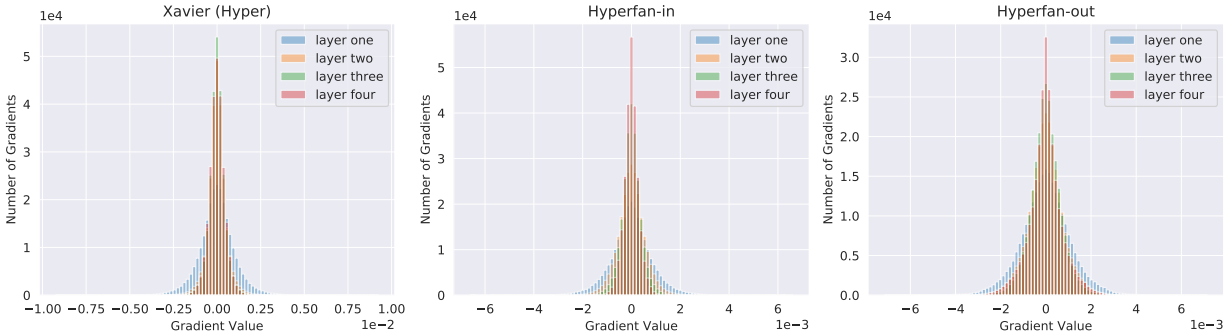

Figure A.10: Hypernet Output Layer Gradients at the End of Training on MNIST.

Hypernet Generates Both Mainnet Weights and Biases

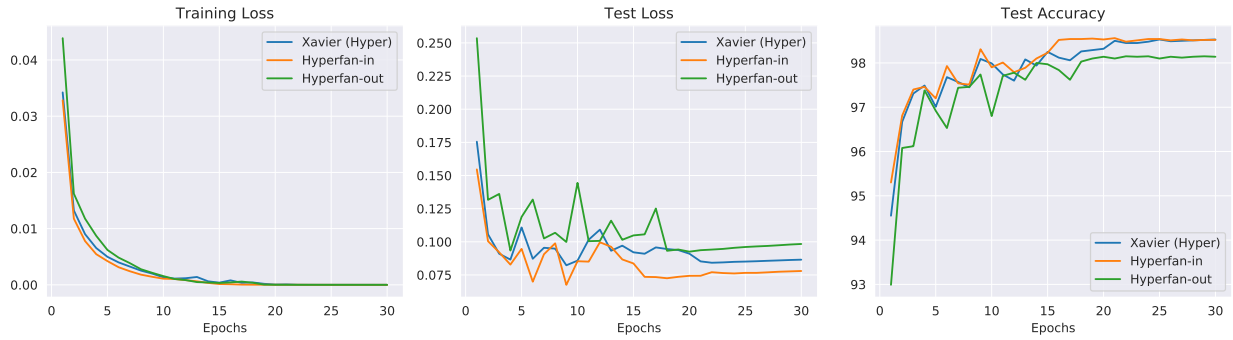

Figure A.11: Loss and Test Accuracy Plots on MNIST.

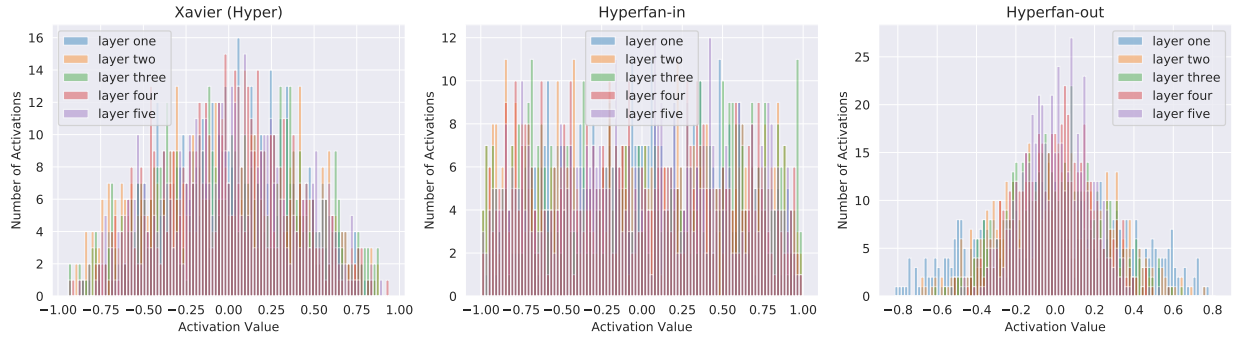

Figure A.12: Mainnet Activations before the Start of Training on MNIST.

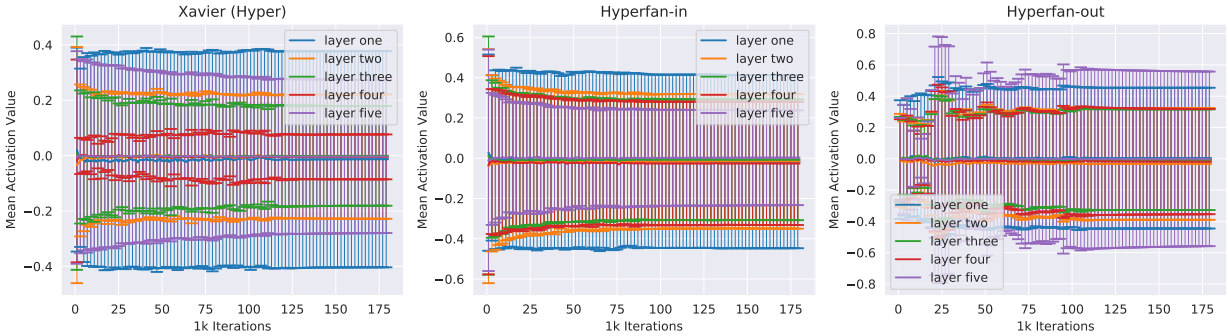

Figure A.13: Evolution of Mainnet Activations during Training on MNIST.

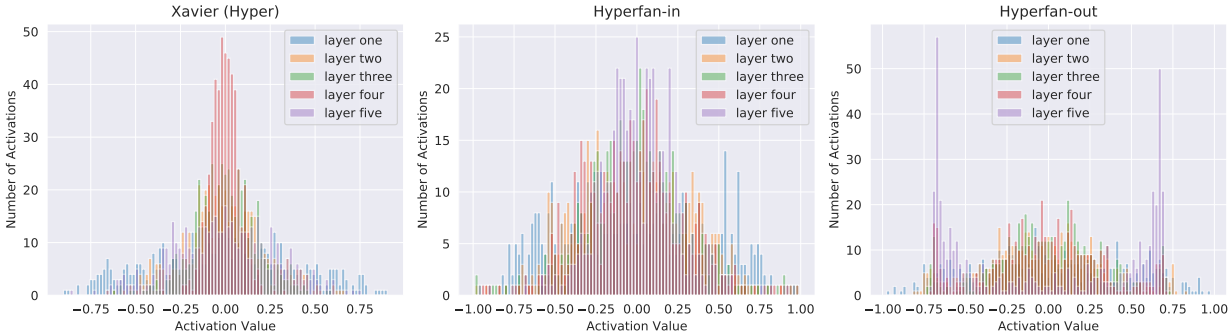

Figure A.14: Mainnet Activations at the End of Training on MNIST.

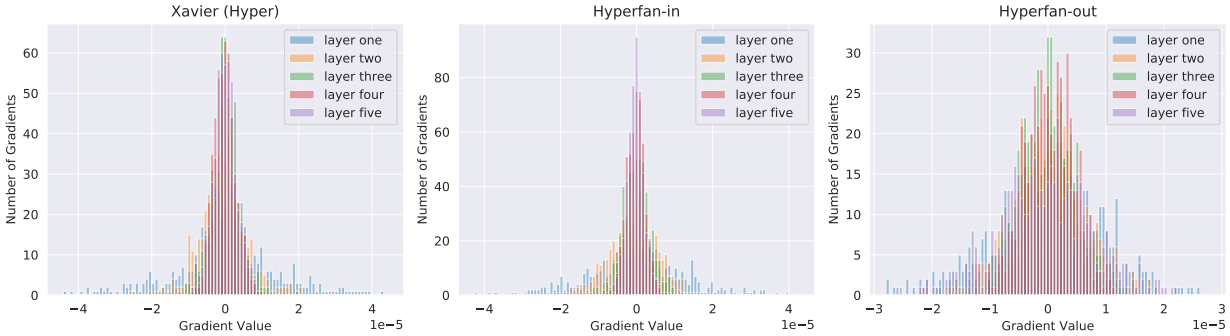

Figure A.15: Mainnet Gradients before the Start of Training on MNIST.

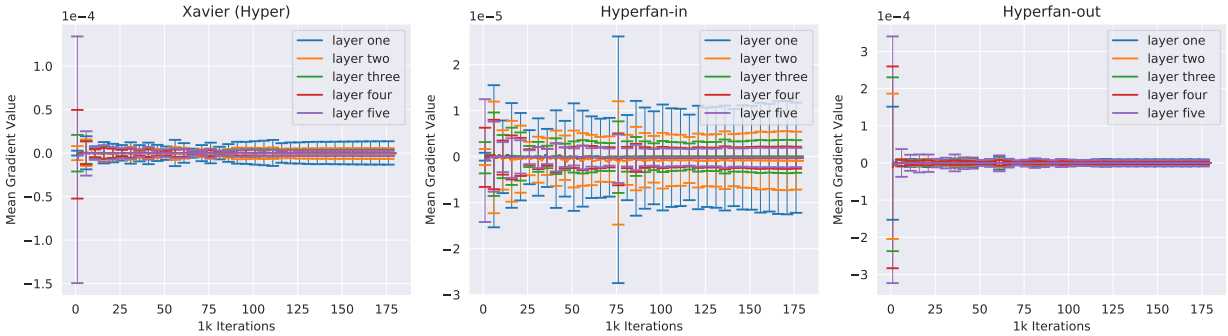

Figure A.16: Evolution of Mainnet Gradients during Training on MNIST.

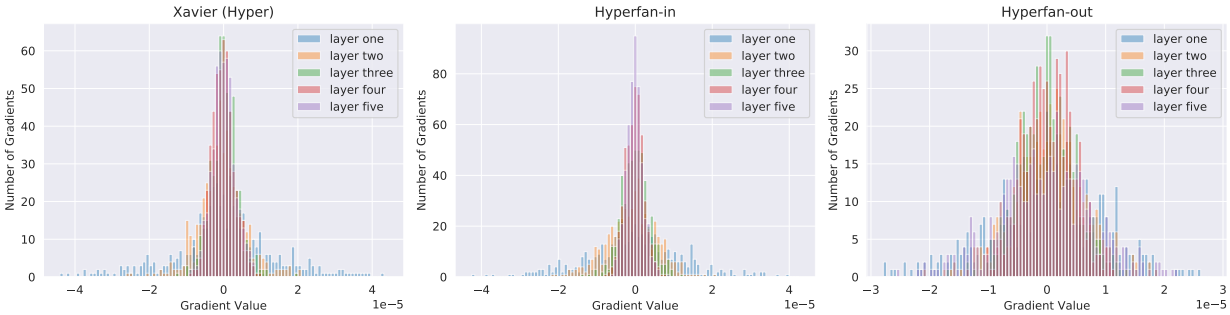

Figure A.17: Mainnet Gradients at the End of Training on MNIST.

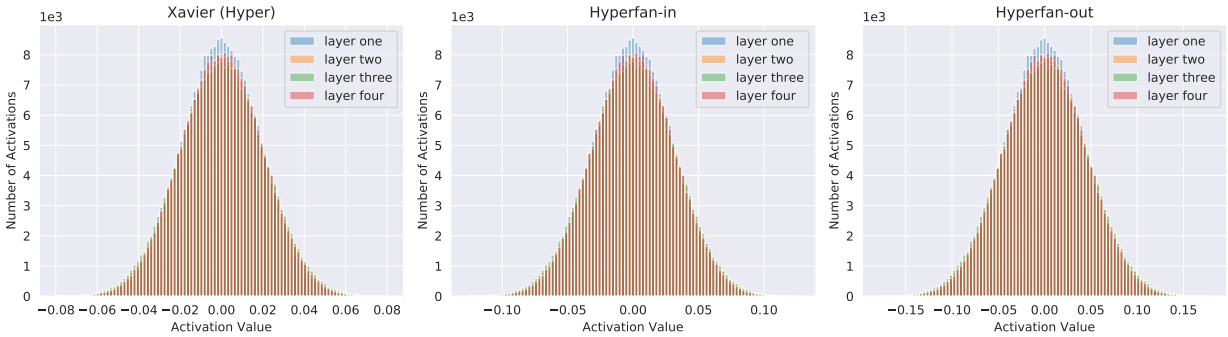

Figure A.18: Hypernet Output Layer Activations before the Start of Training on MNIST.

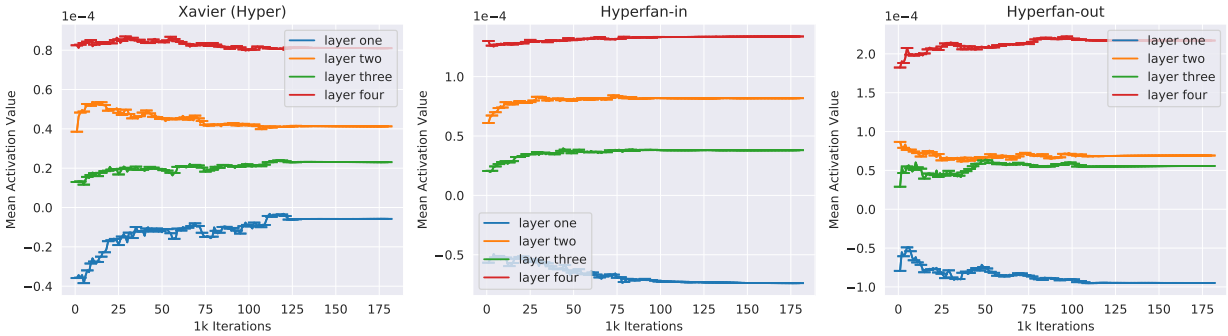

Figure A.19: Evolution of Hypernet Output Layer Activations during Training on MNIST.

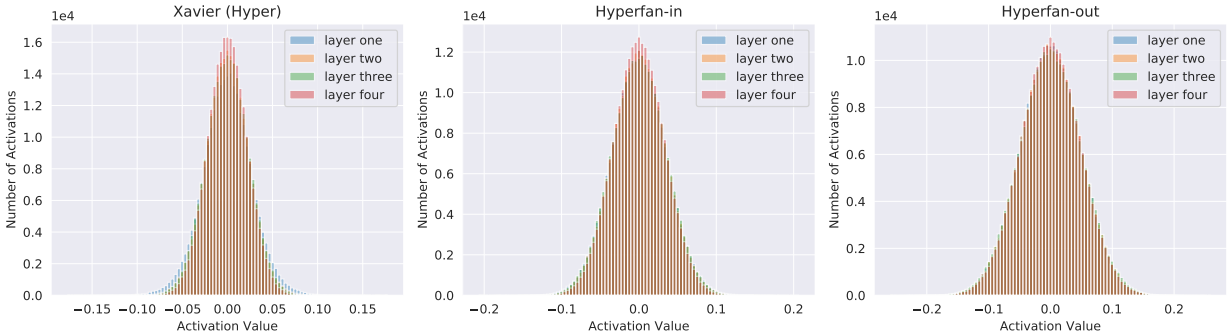

Figure A.20: Hypernet Output Layer Activations at the End of Training on MNIST.

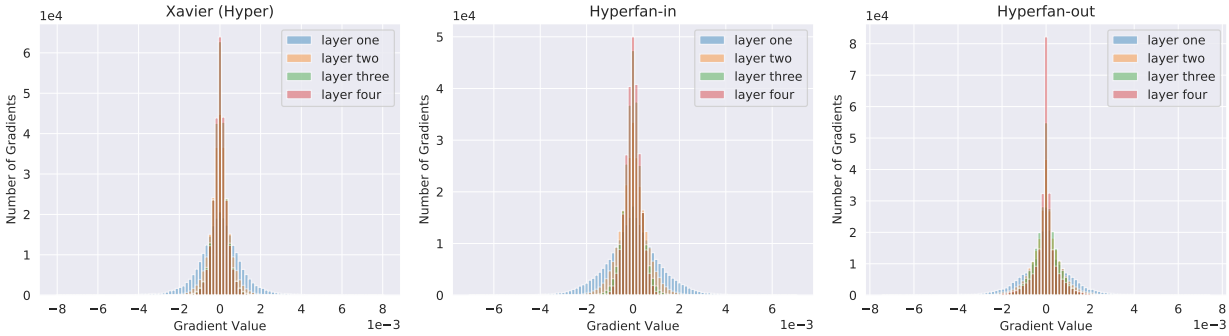

Figure A.21: Hypernet Output Layer Gradients before the Start of Training on MNIST.

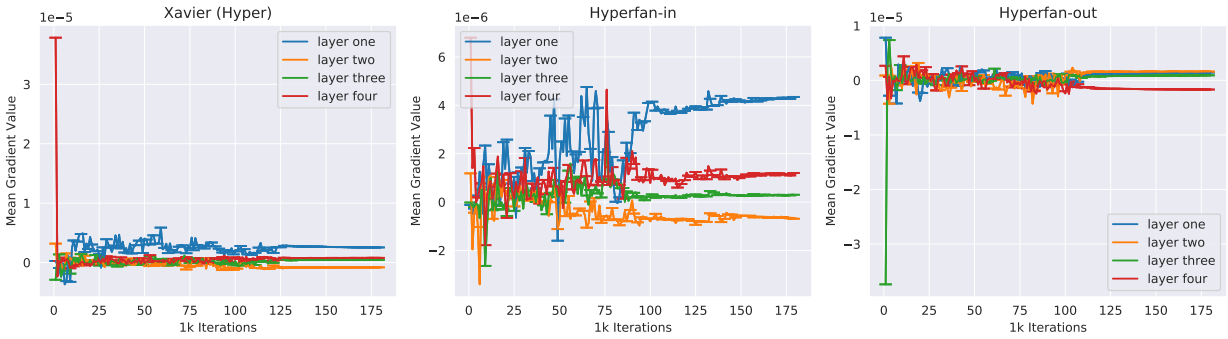

Figure A.22: Evolution of Hypernet Output Layer Gradients during Training on MNIST.

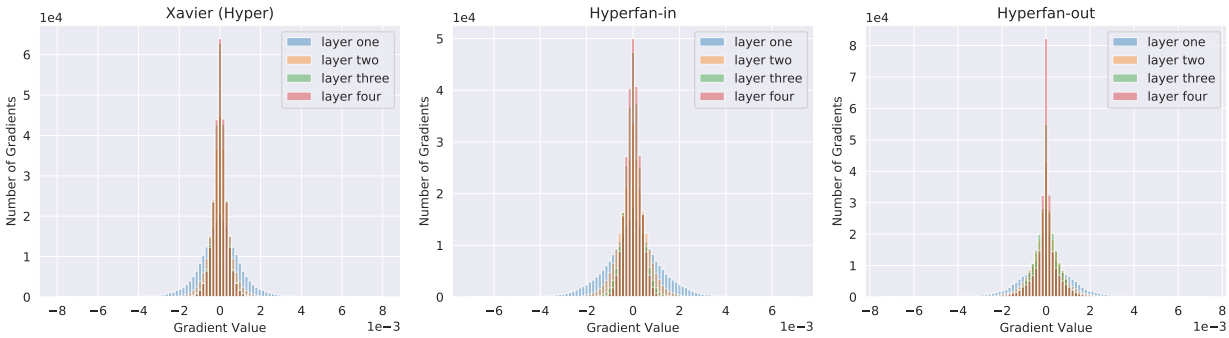

Figure A.23: Hypernet Output Layer Gradients at the End of Training on MNIST.

Remark on the Combination of Fan-in and Fan-out Init

[81] proposed to use the harmonic mean of the two different initialization formulae derived from the forward and backward pass. [82] commented that either version suffices for convergence, and that it does not really matter given that the difference between the two will be a depth-independent factor.

We experimented with the harmonic, geometric, and arithmetic means of the two different formulae in both the classical and the hypernet case. There was no indication of any significant benefit from taking any of the three different means in both cases. Thus, we confirm and concur with [82]'s original observation that either the fan-in or the fan-out version suffices.

A.2.2 Continual Learning on Regression Tasks

The mainnet is a feedforward network with two hidden layers (10 hidden units) and the ReLU activation function. The weights and biases of the mainnet are generated from a hypernet with two hidden layers (10 hidden units) and trainable embeddings of size 2 sampled from $\mathcal{U}(-\sqrt{3}, \sqrt{3})$. We keep the same continual learning hyperparameter β_{output} value of 0.005 and pick the best learning rate for each initialization method from $\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. Notably, Kaiming (fan-in) could only be trained from learning rate 10^{-5}, with losses diverging soon after initialization using the other learning rates. Each task was trained for 6000 training iterations using batch size 32, with Figure 4.4 plotted from losses measured at every 100 iterations.

A.2.3 Convolutional Networks on CIFAR-10

The networks were trained on CIFAR-10 for 500 epochs starting with an initial learning rate of 0.0005 using batch size 100, and decaying with $\gamma = 0.1$ at epochs 350 and 450. The hypernet is composed of two layers (50 hidden units) with separate embeddings and separate input layers but shared output layers. The weight generation happens in blocks of $(96, 3, 3)$ where $K = 96$ is the highest common factor between the different sizes of the convolutional layers in the mainnet and $n = 3$ is the size of the convolutional filters (see Appendix Section A.1.2 for a more detailed

explanation on the hypernet architecture). The embeddings are size 50 and fixed after random sampling from $\mathcal{U}(-\sqrt{3}, \sqrt{3})$. We use the mean cross entropy loss for training, but the summed cross entropy loss for testing.

A.2.4 Bayesian Neural Network on ImageNet

[59] showed that a Bayesian neural network can be developed by using a hypernetwork to express a prior distribution without substantial changes to the vanilla hypernetwork setting. Their methods simply require putting $\mathcal{L}_2$-regularization on the model parameters and sampling from stochastic embeddings. We trained a linear hypernet to generate the weights of a MobileNet mainnet architecture (excluding the batch normalization layers), using the block-wise sampling strategy described in [59], with a factor of 0.0005 for the $\mathcal{L}_2$-regularization. We initialize fixed embeddings of size 32 sampled from $\mathcal{U}(-\sqrt{3}, \sqrt{3})$, and sample additive stochastic noise coming from $\mathcal{U}(-0.1, 0.1)$ at the beginning of every mini-batch training. The training was done on ImageNet with batch size 256 and learning rate 0.1 for 25 epochs, or equivalently, 125125 iterations. The testing was done with 10 Monte Carlo samples. We omit the test loss plots due to the computational expense of doing 10 forward passes after every mini-batch instead of every epoch.

Appendix B: Supplementary Information for Chapter 6

B.1 More Experimental Details

B.1.1 Loading the CUB and MiniImagenet Data

We use [234]'s Torchmeta dataloader implementation to load the CUB and MiniImagenet datasets for the MAML and Meta-SGD experiments. For the MAML++ experiments, we use [183]'s dataloader implementation.

We use 15 test examples within each task and 600 evaluation tasks for meta-validation and meta-testing. Following common practice, every image is resized to 84 by 84 before being inputted into the model.

B.1.2 Model Backbone

The model used in our experiments is a standard 4-layer convolutional neural network backbone that is commonly used in the meta-learning literature. We rely on [183]'s implementation, which has 48 filters, batch normalization and ReLU activations for each convolutional layer, as well as a max pooling and linear layer before the final softmax. The appropriate flags are set such that the standard MAML backbone is used for the MAML and Meta-SGD experiments, while the version with Per Step Batch Normalization is used for MAML++.

B.1.3 Meta-Training

In addition to the pseudo-code provided in Algorithms 4 and 5, we also attach example PyTorch code in the Supplementary materials demonstrating an implementation of gradient sharing on MAML and Meta-SGD. We adapt [183]'s meta-training implementation accordingly to enable gradient sharing for MAML++.

B.2 More Plots

B.2.1 Meta-Validation Plots

We document meta-validation accuracy plots for the CUB dataset in Figure B.1 and the Mini-Imagenet dataset in Figure B.3. Respective plots but for the versions with 10x higher inner loop learning rates can be found in Figures B.5 and B.7. It can be quickly seen that in all plots except one (third row third column in Figure B.7) gradient sharing accelerates meta-training compared to the baseline. The acceleration effect is more pronounced in the 5-task setting and less so in the 1-task setting, which is not surprising because gradient sharing is a multi-task learning based inner loop regularizer. The 1-task setting also occasionally results in a lower meta-validation accuracy peak compared to the baseline. This prompts important future work into inner loop regularizers that can strongly accelerate meta-learning while not sacrificing meta-test performance even in the absence of other tasks in the task batch.

B.2.2 Meta-Test Accuracy

The legend in each of these meta-validation plots also indicates the maximum validation accuracy achieved, the meta-training epoch at which it was achieved, as well as the final meta-test accuracy. We note that the meta-test accuracies established for the MAML and MAML++ baselines generally reproduce or surpass what was reported in [78], [183], and [185], even though specific hyperparameters might be slightly different. However, it seems that the baseline Meta-SGD meta-test accuracy often falls short of that of MAML, which is contrary to what was reported in [179]. Like [179], we initialize all the entries of the vector learning rate $\boldsymbol{\alpha}$ to the same value. While we chose 0.1 for fair comparison to MAML and MAML++, they mentioned that they randomly chose from $[0.005, 0.1]$. It is possible that a hyperparameter search will enable Meta-SGD to outperform MAML, but we note that the meta-validation graphs indicate declining performance beyond a certain point, indicating the presence of task over-fitting. This happens more often than vanilla MAML, which makes sense because there are more inner loop parameters that can be over-fit, and

thus, like our method, it would benefit from outer loop regularization.

B.2.3 Momentum $\boldsymbol{m}$ and Lambda $\boldsymbol{\lambda}$ Variables

Plots for $\boldsymbol{m}$ and $\boldsymbol{\lambda}$ are also documented in Figure B.2 for CUB and Figure B.4 for MiniImagenet, respectively Figures B.6 and B.8 for the versions with 10x inner loop learning rate. If we compare them side-by-side with the meta-validation plots, it is easy to confirm our observation in the main chapter that exemplary outcomes of gradient sharing correspond to low meta-learned $\boldsymbol{m}$ and $\boldsymbol{\lambda}$, while pathological outcomes correspond to high meta-learned $\boldsymbol{m}$ and $\boldsymbol{\lambda}$. In general, even for the pathological experiments, we can use the meta-validation set to perform early stopping and pick models that have yet to over-fit, so this is not a major issue.

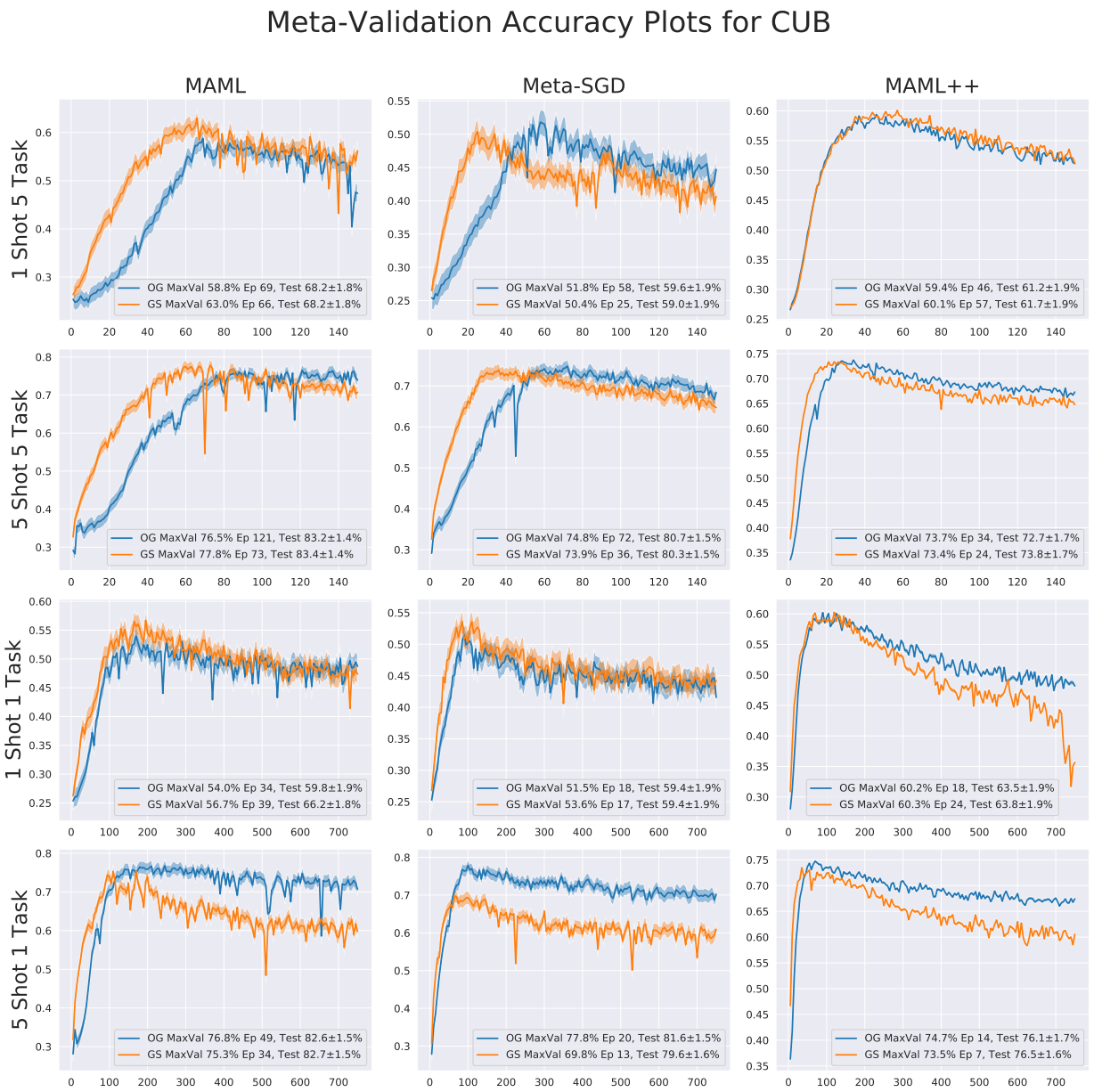

Figure B.1: Meta-Validation Accuracy Plots for the CUB dataset. The x axes denote the number of meta-training epochs, the y axes denote the accuracy on the meta-validation set, and the shaded areas denote the 95% standard error confidence interval. In the legend, OG denotes the original baseline meta-learning method, and GS denotes the version with Gradient Sharing. MaxVal [A] Ep [B] denotes that the maximum meta-validation accuracy of [A] was achieved at epoch [B]. Test [C]±[D] denotes that the meta-test accuracy of [C] was achieved within a 95% confidence interval of [D]. The column headers denote the meta-learning method, while the row headers denote the number of shots and number of tasks in the task batch. All experiments are done in the 5-way few-shot classification setting, with the meta-test accuracy reported using an ensemble composed of the top 5 meta-validation accuracy models.

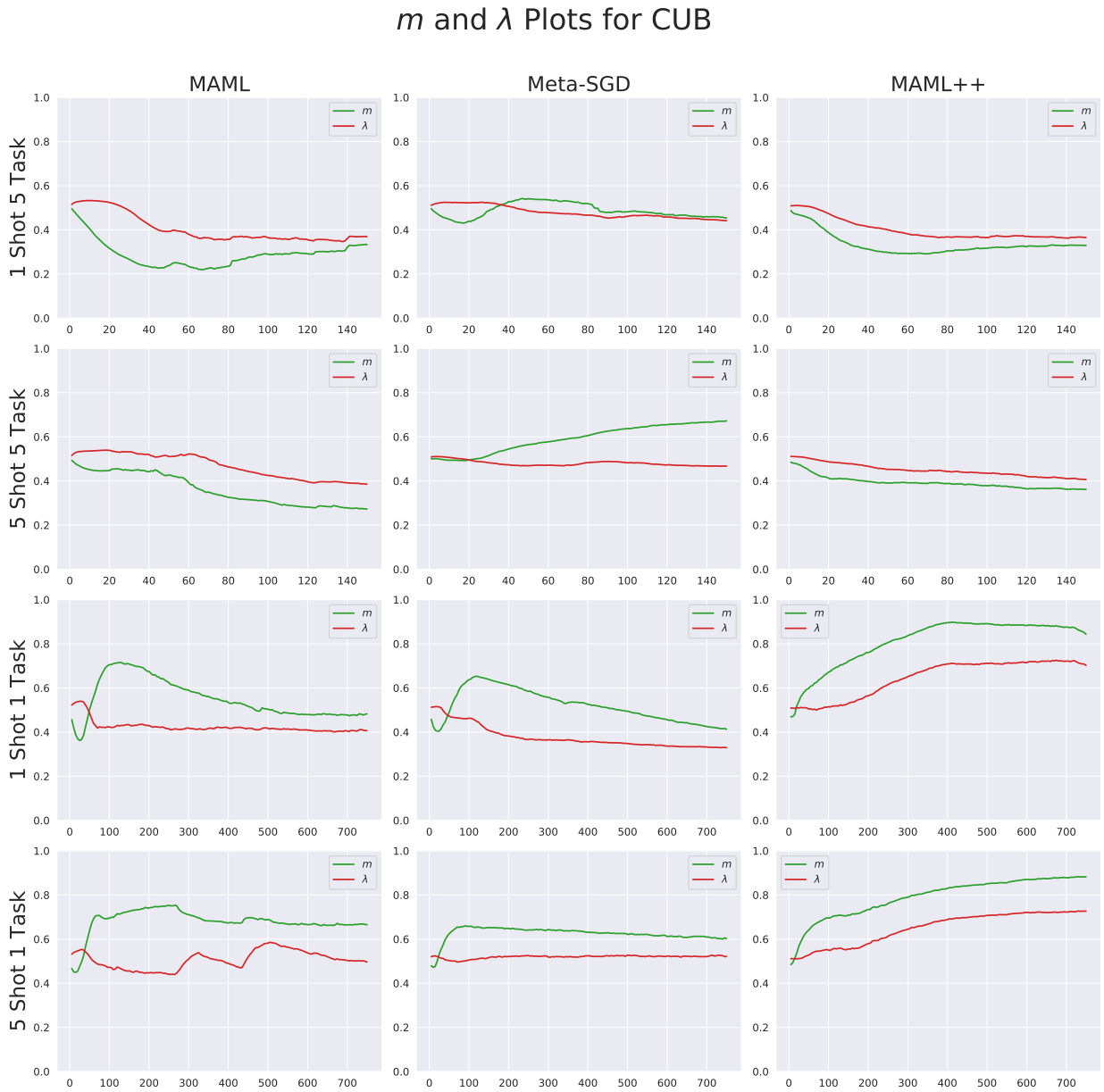

Figure B.2: Evolution of Gradient Sharing Parameters throughout Meta-Training for the CUB dataset. The x axes denote the number of meta-training epochs, while the y axes denote the mean sigmoided value of the gradient sharing parameter. Specifically, m denotes the average value of $\sigma(m_k)$ and λ denotes the average value of $\sigma(\lambda_k)$ across $k \in [1, K]$. $K = 5$ was set for all our experiments.

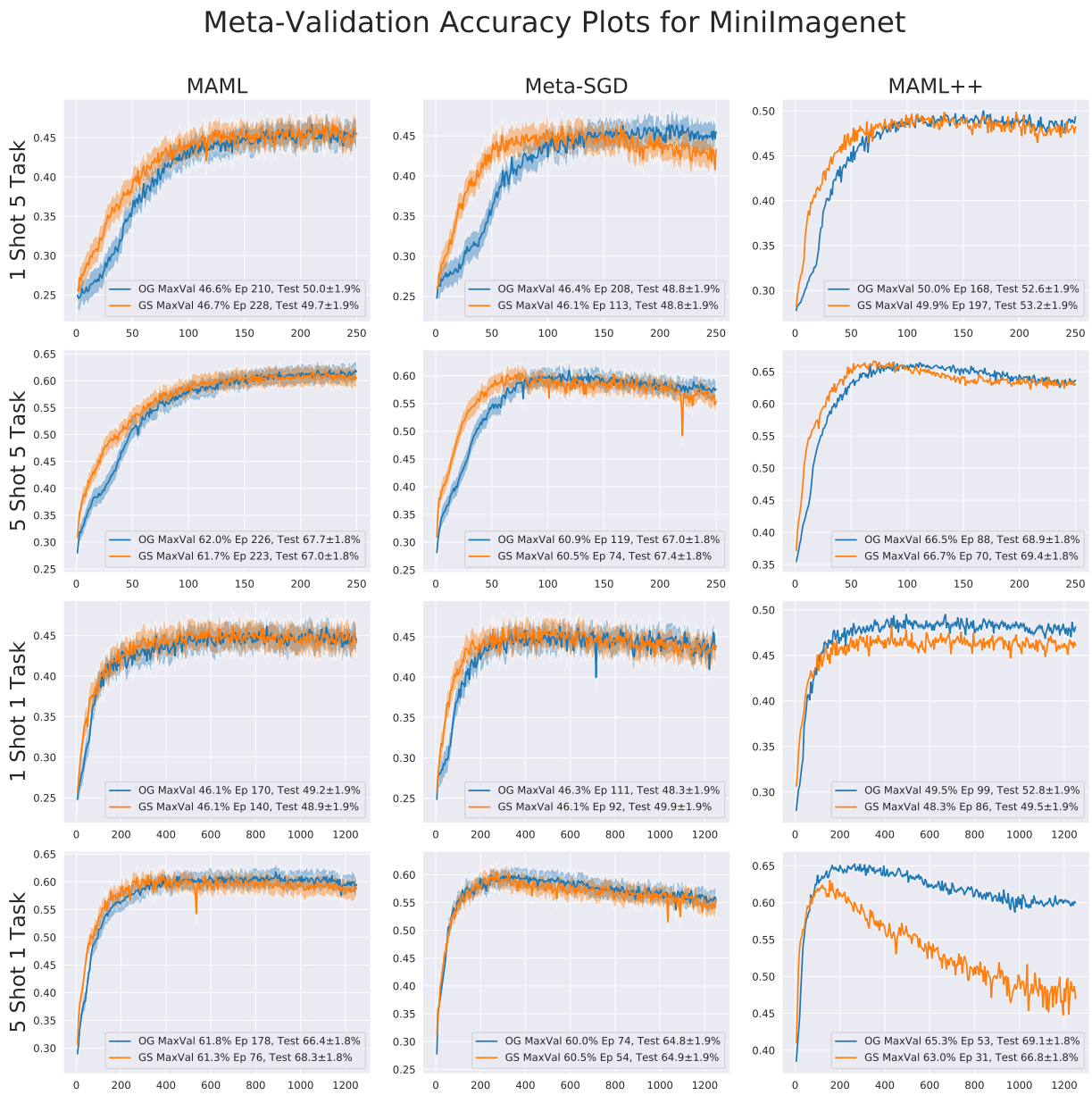

Figure B.3: Meta-Validation Accuracy Plots for the MiniImagenet dataset. The x axes denote the number of meta-training epochs, the y axes denote the accuracy on the meta-validation set, and the shaded areas denote the 95% standard error confidence interval. In the legend, OG denotes the original baseline meta-learning method, and GS denotes the version with Gradient Sharing. MaxVal [A] Ep [B] denotes that the maximum meta-validation accuracy of [A] was achieved at epoch [B]. Test [C]±[D] denotes that the meta-test accuracy of [C] was achieved within a 95% confidence interval of [D]. The column headers denote the meta-learning method, while the row headers denote the number of shots and number of tasks in the task batch. All experiments are done in the 5-way few-shot classification setting, with the meta-test accuracy reported using an ensemble composed of the top 5 meta-validation accuracy models.

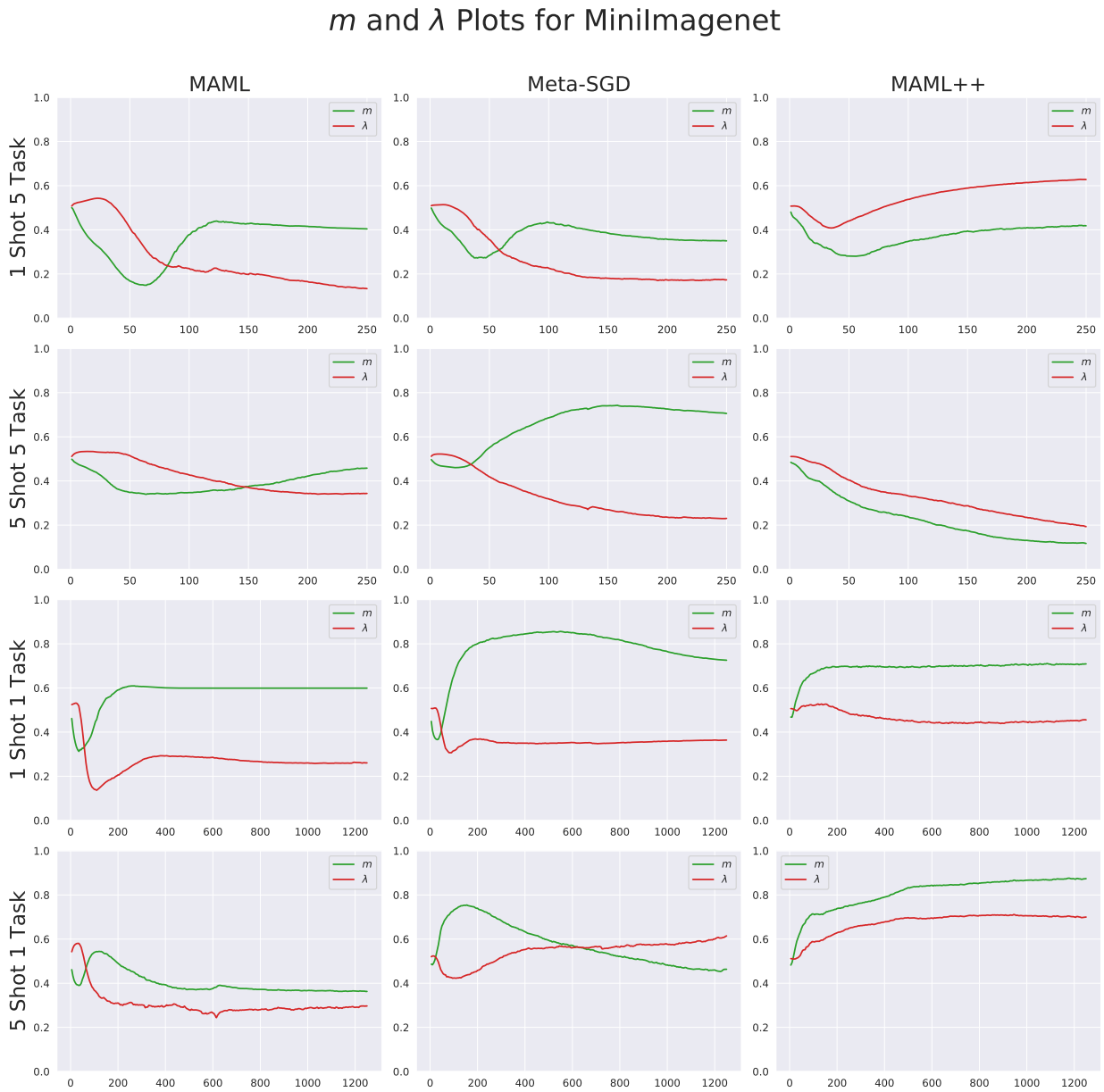

Figure B.4: Evolution of Gradient Sharing Parameters throughout Meta-Training for the MiniImagenet dataset. The x axes denote the number of meta-training epochs, while the y axes denote the mean sigmoided value of the gradient sharing parameter. Specifically, m denotes the average value of $\sigma(m_k)$ and λ denotes the average value of $\sigma(\lambda_k)$ across $k \in [1, K]$. $K = 5$ was set for all our experiments.

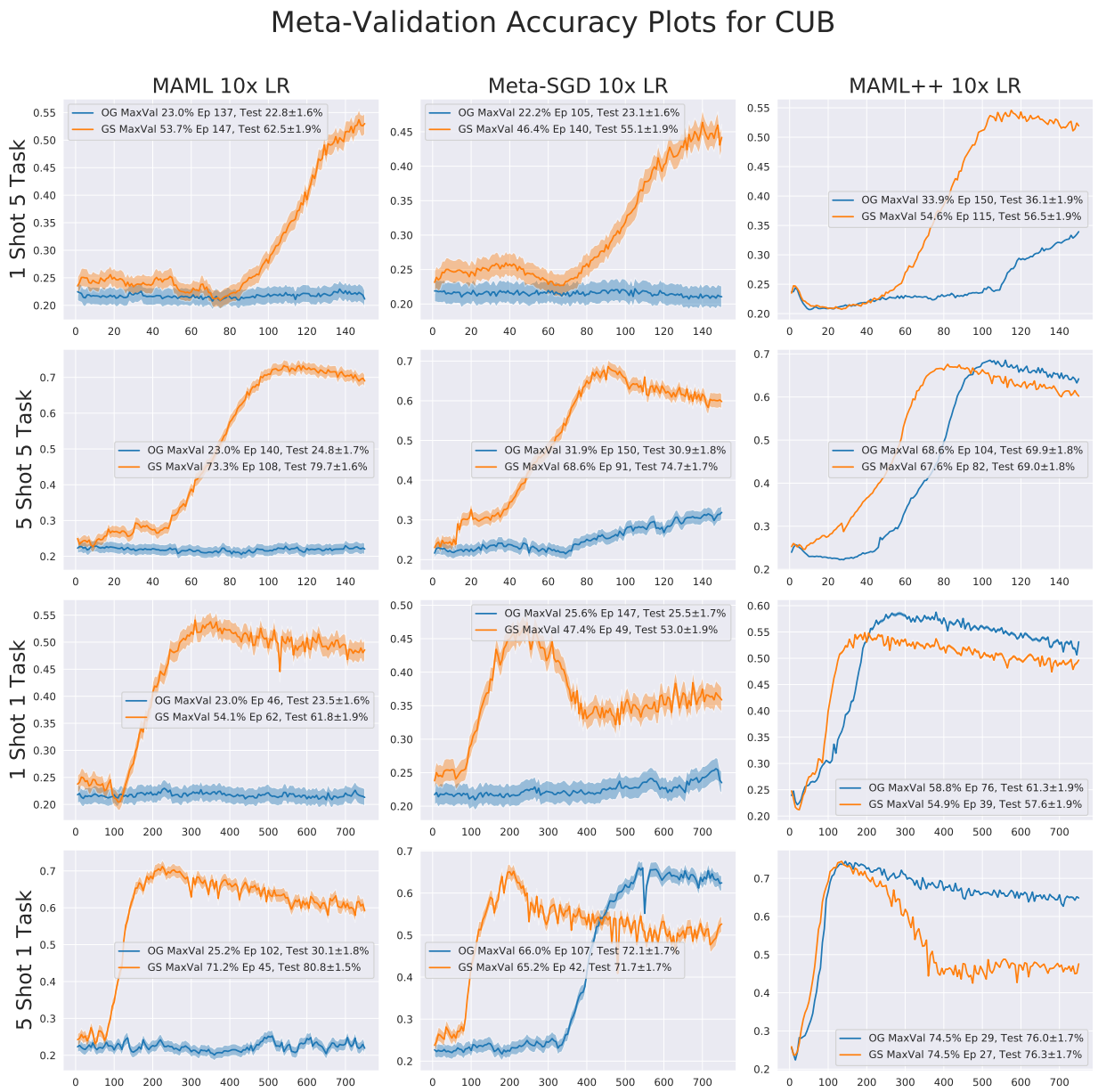

Figure B.5: Meta-Validation Accuracy Plots for the CUB dataset with 10x the Inner Loop Learning Rate. The x axes denote the number of meta-training epochs, the y axes denote the accuracy on the meta-validation set, and the shaded areas denote the 95% standard error confidence interval. In the legend, OG denotes the original baseline meta-learning method, and GS denotes the version with Gradient Sharing. MaxVal [A] Ep [B] denotes that the maximum meta-validation accuracy of [A] was achieved at epoch [B]. Test [C]±[D] denotes that the meta-test accuracy of [C] was achieved within a 95% confidence interval of [D]. The column headers denote the meta-learning method, while the row headers denote the number of shots and number of tasks in the task batch. All experiments are done in the 5-way few-shot classification setting, with the meta-test accuracy reported using an ensemble composed of the top 5 meta-validation accuracy models.

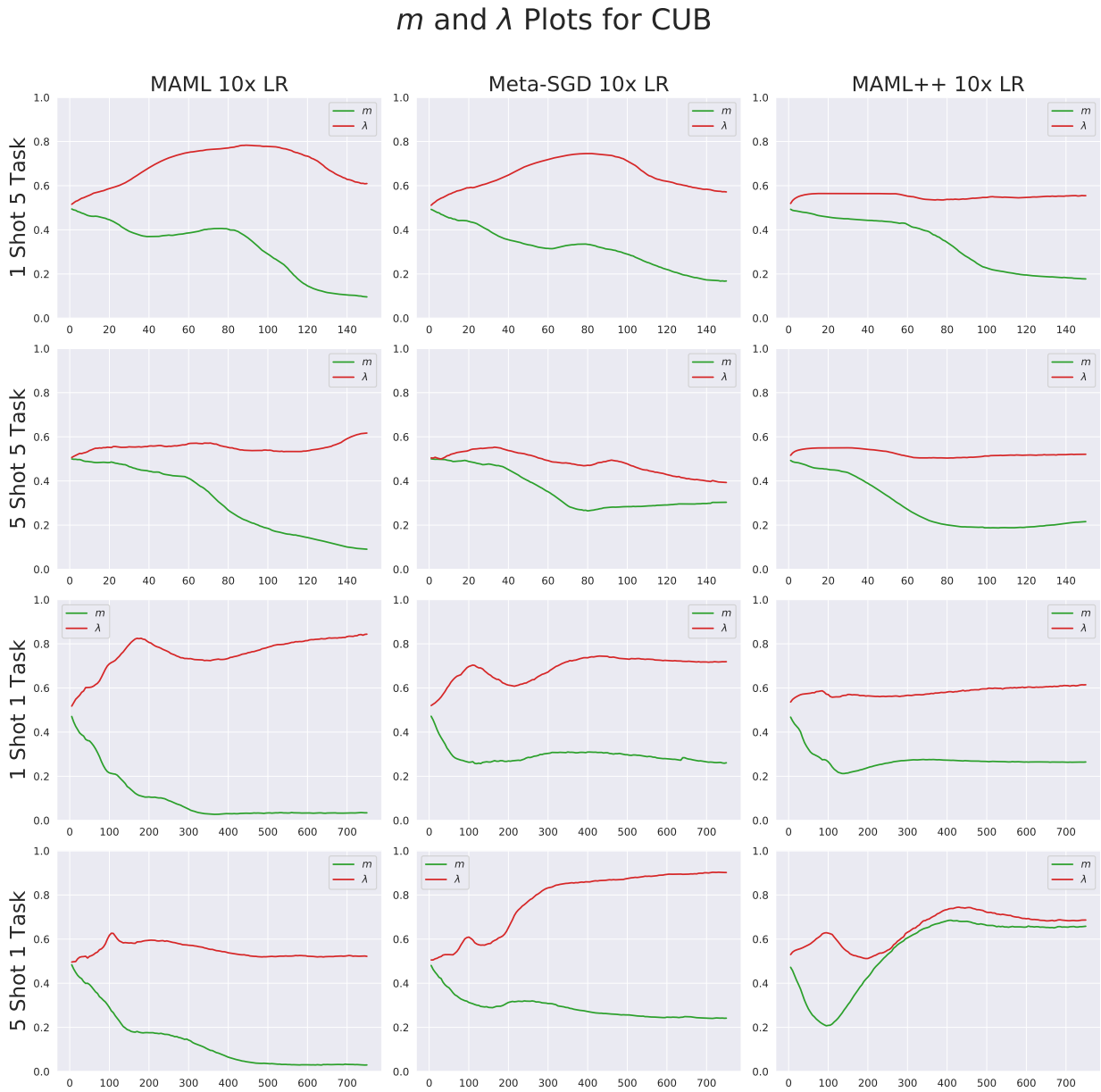

Figure B.6: Evolution of Gradient Sharing Parameters throughout Meta-Training for the CUB dataset with 10x the Inner Loop Learning Rate. The x axes denote the number of meta-training epochs, while the y axes denote the mean sigmoided value of the gradient sharing parameter. Specifically, m denotes the average value of $\sigma(m_k)$ and λ denotes the average value of $\sigma(\lambda_k)$ across $k \in [1, K]$. $K = 5$ was set for all our experiments.

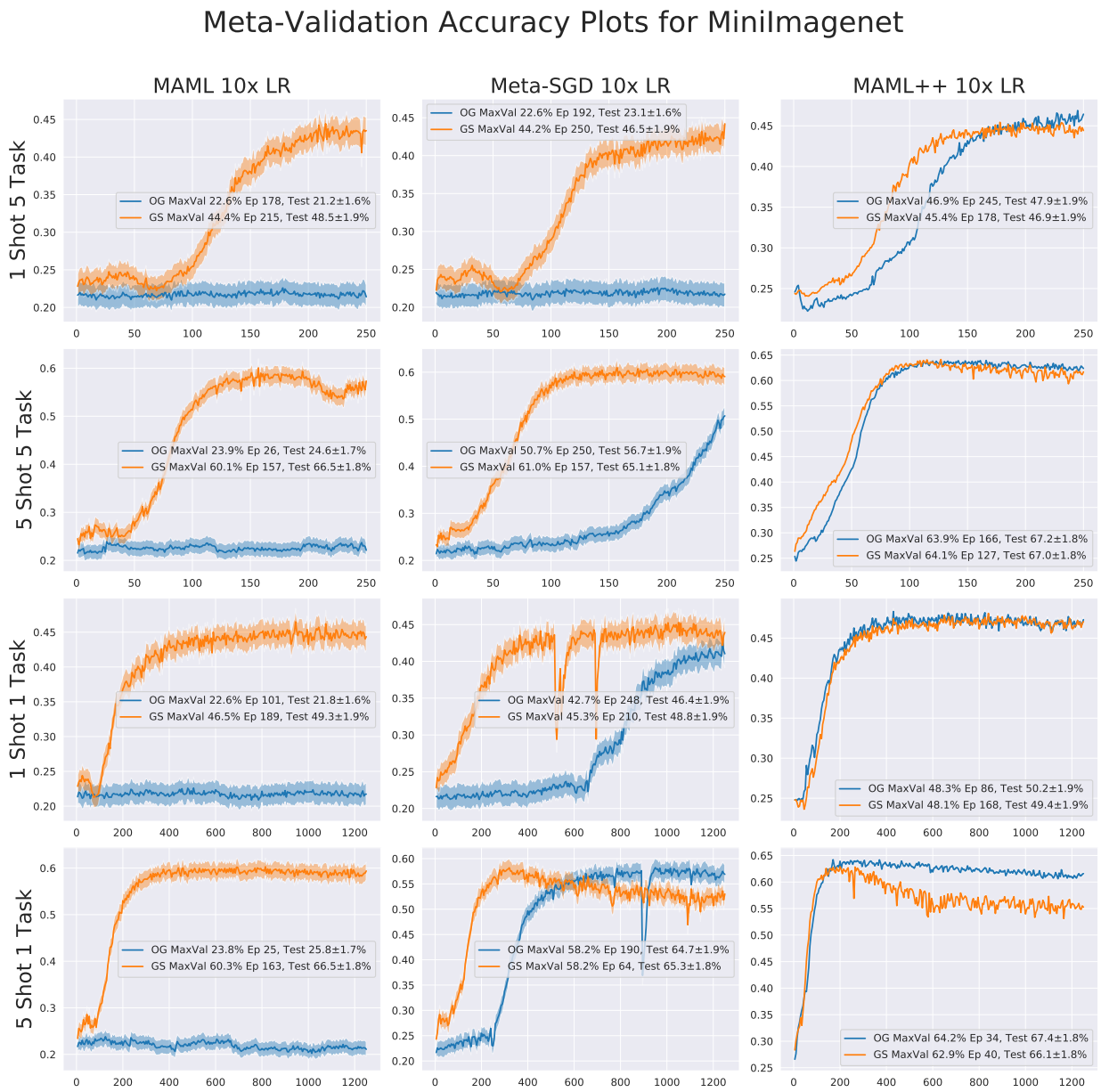

Figure B.7: Meta-Validation Accuracy Plots for the MiniImagenet dataset with 10x the Inner Loop Learning Rate. The x axes denote the number of meta-training epochs, the y axes denote the accuracy on the meta-validation set, and the shaded areas denote the 95% standard error confidence interval. In the legend, OG denotes the original baseline meta-learning method, and GS denotes the version with Gradient Sharing. MaxVal [A] Ep [B] denotes that the maximum meta-validation accuracy of [A] was achieved at epoch [B]. Test [C]±[D] denotes that the meta-test accuracy of [C] was achieved within a 95% confidence interval of [D]. The column headers denote the meta-learning method, while the row headers denote the number of shots and number of tasks in the task batch. All experiments are done in the 5-way few-shot classification setting, with the meta-test accuracy reported using an ensemble composed of the top 5 meta-validation accuracy models.

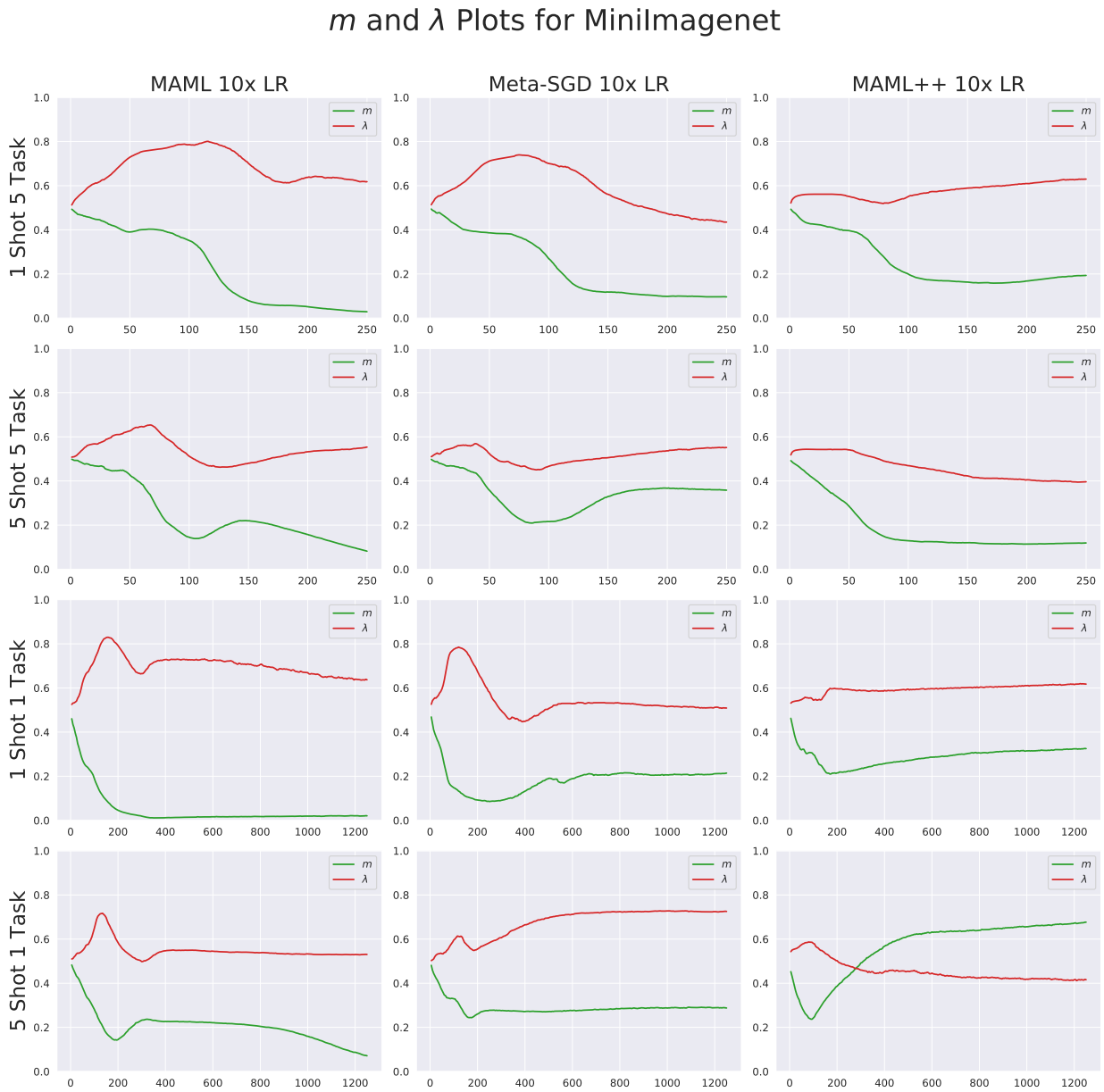

Figure B.8: Evolution of Gradient Sharing Parameters throughout Meta-Training for the MiniImagenet dataset with 10x the Inner Loop Learning Rate. The x axes denote the number of meta-training epochs, while the y axes denote the mean sigmoided value of the gradient sharing parameter. Specifically, m denotes the average value of $\sigma(m_k)$ and λ denotes the average value of $\sigma(\lambda_k)$ across $k \in [1, K]$. $K = 5$ was set for all our experiments.

Appendix C: Supplementary Information for Chapter 7

C.1 Solution to the Raven's Matrix puzzle

Figure C.1: The three basic glyphs are formed from half a circle, a triangle, and a rectangle respectively.

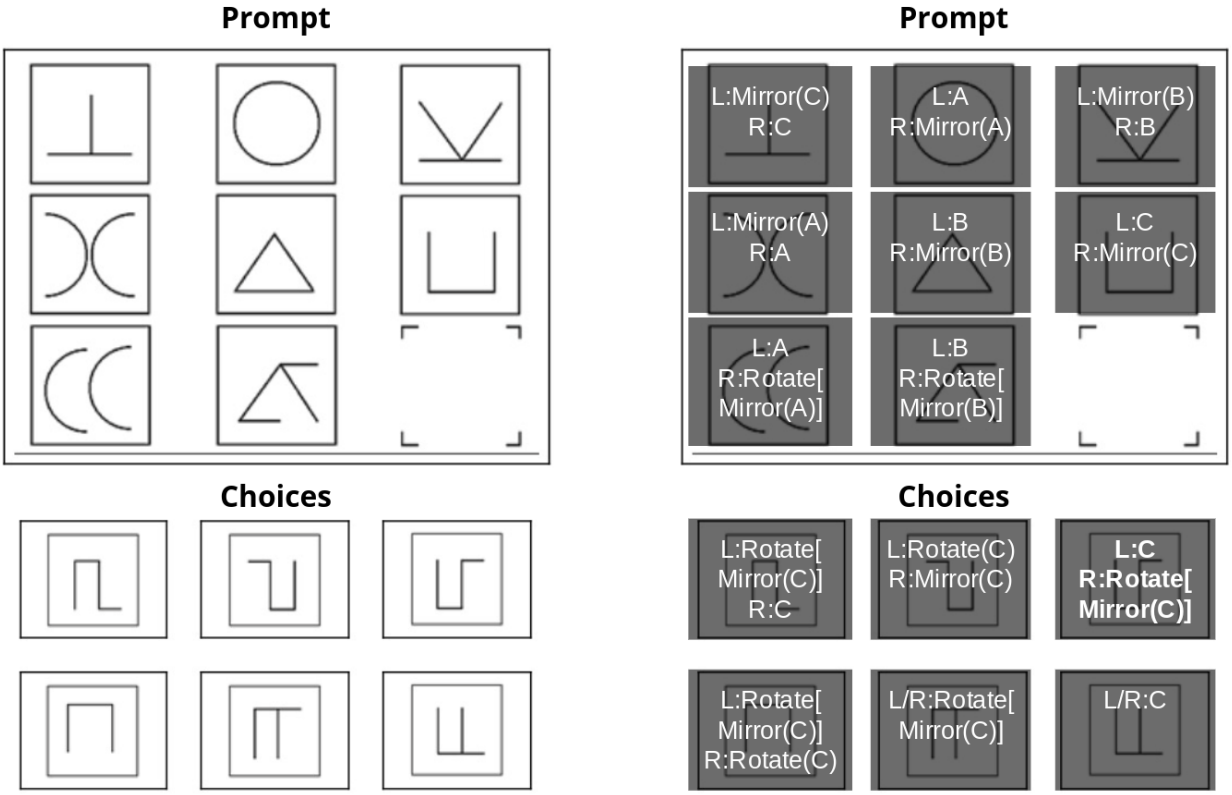

Figure C.2: The solution to the Raven's Matrix puzzle is the choice on the top right.

The source of this puzzle and its solution is [235] on Puzzling Stack Exchange.

Each panel is composed of a glyph on the left hand side (L) and a glyph on the right hand side (R). There are three basic glyphs (see Figure C.1): a crescent (A), a half triangle (B), and a half rectangle (C). Each glyph can also be mirrored (Mirror), i.e. flipped horizontally, or rotated by 180 degrees (Rotate). In Figure C.2, we annotate every panel in both the prompt and the choices with the symbols that represent it. It is clear that the blank in the prompt should be filled by a left glyph C and a right glyph Rotate[Mirror(C)], which is the choice on the top right.

C.2 Related Work on Non-Visual Sudoku

On a dataset with 216,000 puzzles split in a 10:1:1 train-val-test ratio, a deep (recurrent relational) network that has access to positional information for each cell scores 100% test accuracy on puzzles with 33 pre-filled cells and 96.6% on puzzles with 17 pre-filled cells [221]. [220] use a differentiable quadratic programming layer called OptNet, which like SATNet has no a priori knowledge of the rules, in a neural network to solve for Sudoku. OptNet does not scale well computationally and can only solve 4-by-4 Sudokus.

C.3 Experimental Settings

In the Supplementary materials, we provide source code and the shell commands to replicate all the experimental results in the paper.

C.3.1 SATNet Fails at Symbol Grounding

The experimental settings for SATNet in Section 7.3 are identical to the original paper and based on the authors' open-sourced implementation available at `https://github.com/locuslab/SATNet`. Specifically, the CNN used is the sequence of layers: *Conv1-ReLU-MaxPool-Conv2-ReLU-MaxPool-FC1-ReLU-FC2-Softmax*, where *Conv1* has a 5x5 kernel (stride 1) and 20 output channels, *Conv2* has a 5x5 kernel (stride 1) and 50 output channels, *FC1* has size 800x500, *FC2* has size 500x10, and the *MaxPool* layers have a 2x2 kernel (stride 2). This is roughly the LeNet5

architecture, but with one less fully connected layer at the end and around 10x the number of parameters. The SATNet layer contains 300 auxiliary variables, with $n = 729$ and $m = 600$. The full model is trained using Adam for 100 epochs using batch size 40, with a learning rate of $2\text{x}10^{-3}$ for the SATNet layer and $1\text{x}10^{-5}$ for the CNN.

C.3.2 MNIST Mapping Problem

We use batch size 64 for training throughout all the experiments. We use the Sudoku CNN described above in Appendix Section C.3.1 as the backbone layer for all the experiments, except the one in Finding 4 where we vary the architecture. We use $m = 200, aux = 100$ for the SATNet layer for all the experiments, except the one in Finding 1 where we vary m and aux.

Non-SATNet baseline: The whole network was trained with Adam using a $2\text{x}10^{-3}$ learning rate.

Finding 1: The SATNet layer was trained with a $2\text{x}10^{-3}$ learning rate, and the backbone layer was trained with a $1\text{x}10^{-5}$ learning rate, both using Adam as was done above in Appendix Section C.3.1.

Finding 2: Both the SATNet layer and the backbone layer were trained with Adam.

Findings 3 and 4: The SATNet layer was trained with a $1\text{x}10^{-3}$ learning rate using Adam, and the backbone layer was trained with a $1\text{x}10^{-1}$ learning rate with SGD.

C.4 More Experimental Results for the MNIST Mapping Problem

C.4.1 Non-SATNet Baseline

The training accuracy for the non-SATNet baseline is 72.4±13.4% (3).

C.4.2 Experiment 1

Table C.1: Effects of *m* and *aux* on Training and Test Accuracy

m	*aux*	Training Accuracy	Test Accuracy
20	50	86.7±8.4% (1)	86.8±8.4% (1)
40	50	95.6±0.3% (0)	95.5±0.3% (0)
60	50	95.7±0.3% (0)	95.6±0.4% (0)
80	50	96.2±0.2% (0)	96.0±0.3% (0)
20	100	82.2±8.4% (1)	82.4±8.4% (1)
40	100	85.9±8.3% (1)	85.9±8.3% (1)
60	100	95.3±0.5% (0)	95.3±0.5% (0)
80	100	95.1±0.2% (0)	94.9±0.2% (0)
20	200	43.9±13.5% (6)	44.0±13.4% (6)
40	200	59.6±13.3% (4)	59.7±13.3% (4)
60	200	60.0±13.4% (4)	60.2±13.3% (4)
80	200	94.7±0.3% (0)	94.6±0.3% (0)
100	200	86.3±8.4% (1)	86.2±8.4% (1)
100	400	44.8±12.5% (4)	45.0±12.6% (4)
100	600	25.6±7.7% (7)	26.2±7.9% (7)
100	800	35.1±10.3% (6)	35.8±10.4% (6)
200	200	96.2±0.1% (0)	95.8±0.2% (0)
200	400	45.6±12.9% (4)	45.3±12.9% (4)
200	600	62.4±11.5% (2)	62.4±11.7% (2)
200	800	32.7±10.4% (5)	33.2±10.5% (5)
400	200	96.4±0.2% (0)	96.0±0.2% (0)
400	400	92.1±4.2% (0)	91.8±4.0% (0)
400	600	62.8±13.5% (3)	62.7±13.4% (3)
400	800	69.3±12.8% (3)	69.4±12.7% (3)

C.4.3 Experiment 2

Table C.2: Effects of Different Learning Rates on the SATNet and Backbone Layer on Training Accuracy

SATNet Layer	Backbone Layer Learning Rate		
Learning Rate	$1x10^{-3}$	$1x10^{-4}$	$1x10^{-5}$
$1x10^{-3}$	19.6±8.5% (9)	90.4±8.8% (1)	96.7±0.2% (0)
$1x10^{-4}$	17.0±4.1% (8)	74.9±8.8% (0)	96.5±0.2% (0)
$1x10^{-5}$	14.4±3.4% (9)	31.8±7.1% (5)	71.9±5.4% (0)

C.4.4 Experiment 3

The training accuracy rose from 96.7±0.2% (0) to 99.1±0.1% (0).

C.4.5 Experiment 4

Table C.3: Effects of Different Neural Architectures on Training Accuracy

		Backbone Output Layer	
Architectures	Parameters	Softmax	Sigmoid
LeNet [232]	68,626	63.2±14.2% (4)	99.1±0.0% (0)
Sudoku CNN	860,780	99.1±0.1% (0)	**99.5±0.0% (0)**
ResNet18 [160]	11,723,722	67.6±6.2% (0)	97.4±0.4% (0)