The Dissertation Committee for Elliot Keeler Meyerson
certifies that this is the approved version of the following dissertation:

Discovering Multi-Purpose Modules through Deep Multitask Learning

Committee:

Risto Miikkulainen, Supervisor

Kristen Grauman

Greg Durrett

Geoff Nitschke

Discovering Multi-Purpose Modules through Deep Multitask Learning

by

Elliot Keeler Meyerson

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2018

To Charles.

Acknowledgments

This work would not be possible without Risto's vast and unfaltering support. He has enabled me to keep an open mind and attack diverse problems, while showing me how to tell stories, so that more than three people get the point. I'd also like to thank Alex, Mark, and Joel, and also the entire team at Sentient, for their diverse perspectives, resources, and expertise that enabled this work to spin forward.

Off the playing field, I'd like thank to Kallan for her constant patience and understanding, and for being a sounding board through both the groovy and fractured times; Lena for teaching me the zen of not being in a hurry; my parents for showing me how to love systems and complexity; and Kay and George and my Austin family for making UT a homecoming.

Discovering Multi-Purpose Modules through Deep Multitask Learning

Publication No. ____________________

Elliot Keeler Meyerson, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Risto Miikkulainen

Machine learning scientists aim to discover techniques that can be applied across diverse sets of problems. Such techniques need to exploit regularities that are shared across tasks. This begs the question: What shared regularity is not yet being exploited? Complex tasks may share structure that is difficult for humans to discover. The goal of deep multitask learning is to discover and exploit this structure automatically by training a joint model across tasks. To this end, this dissertation introduces a deep multitask learning framework for collecting generic functional modules that are used in different ways to solve different problems. Within this framework, a progression of systems is developed based on assembling shared modules into task models and leveraging the complementary advantages of gradient descent and evolutionary optimization. In experiments, these systems confirm that modular sharing improves performance across a range of application areas, including general video game playing, computer vision, natural language processing, and genomics; yielding state-of-the-art results in several cases. The conclusion is that multi-purpose modules discovered by deep multitask learning can exceed those developed by humans in performance and generality.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

To efficiently construct complex solutions to difficult problems, real-world general problem-solvers rely on *functional modules.* Each such module is a tool specialized to solve a particular class of problems or subproblems. For example, individual humans specialize into distinct roles in social organizations, and, at one level lower, each human refines their own set of complementary skills. In both cases, an effective set of modules is discovered through the experience of solving many diverse problems. That is, these sets are refined and reorganized based on performance, in order to maximize system generality while maintaining efficiency. Importantly, this implies that, through such refinement, improved performance on one task can lead to improved performance on another, even if the tasks are seemingly unrelated.

Consider predicting the onset of a rare disease. There is a shortage of data for this particular disease, but a specialist doctor has honed a set of diagnosis techniques on related diseases, and can adapt these for a best guess. Her more general medical abilities, developed in school and rotations, should also lead to insights. Further, beyond medicine, any challenging task from her life could help, e.g., competitive Judo, since through it she sharpens her general problem-solving skills. A machine with such functional modularity could search for insight on a much broader scale. Modules trained to accurately diagnose related diseases should provide a good starting point, and solutions to any medical task could lead to insight, through their assessment of the human condition. Further, solving a seemingly unrelated task, like predicting the sentiment of Tweets, could lead to more accurate diagnoses, i.e., if modules discovered for sentiment prediction generalize well to medical applications.

Though it may not be obvious how diseases and Tweets are related, such generalization yields practical benefits today. For example, advances in neural networks for sequence processing, originally developed for speech and natural language, are successfully repurposed for genomics and heart-monitoring tasks. The generic formulation of such machine learning techniques enables rapid experimentation and modular recombination for new problems. However, this methodological modularity cannot integrate knowledge across tasks as deeply as biological or social problem-solvers. The standard toolbox of machine learning techniques is refined by humans, but complex solutions to multiple problems may contain powerful generic modules that are difficult for humans to discover. This dissertation develops an approach to automatically discover such modules.

1.1 Motivation

The field of machine learning is naturally well-suited to take advantage of the modular approach. The success of machine learning is grounded in a relatively small set of standard functions, which can be applied in a modular fashion to a broad array of problem areas. Such functional modularity appears to be a requirement for general problem-solving systems in the real world. It supports flexibility, adaptivity, and efficiency over what would be possible with a monolithic system. For example, doctors would be mentally slower and less energy-efficient if they had to activate their entire brain uniformly to solve every new problem that came their way. At a higher level, large consulting firms would be slower and less efficient if they could not quickly assemble project pipelines out of specialized human teams.

The general effectiveness of machine learning and these other problem-solving systems can be attributed to structural regularities shared across diverse problems. These shared regularities can be exploited by standardized modules, be they machine learning techniques, brain regions, or complete humans. The generality of a system is then determined by the breadth of application of its modules. In machine learning, multi-purpose modules are discovered by humans through experimentation, intuition, and analysis of semantic structural regularities across problems.

Multitask learning is a machine learning approach that improves generalization of models by automatically exploiting regularities that exist across a set of problems (Caruana, 1998). Regularities are exploited by sharing learned functionality across the models trained for each problem. In recent years, multitask learning has been extended to the realm of deep learning (Lecun et al., 2015), where the multitask approach has been used to complement the success of deep models across core areas of artificial intelligence, including computer vision (Zhang et al., 2014; Bilen & Vedaldi, 2016; Misra et al., 2016; Rudd et al., 2016; Lu et al., 2017; Rebuffi et al., 2017; Yang & Hospedales, 2017), natural language processing (Collobert & Weston, 2008; Dong et al., 2015; Liu et al., 2015a; Luong et al., 2016; Hashimoto et al., 2017), speech processing (Huang et al., 2013; Seltzer & Droppo, 2013; Huang et al., 2015; Wu et al., 2015), and reinforcement learning (Devin et al., 2016; Fernando et al., 2017; Jaderberg et al., 2017b; Teh et al., 2017). The regularities discovered by deep multitask learning are encoded in high-dimensional tensors of learned parameters that are shared across models. These regularities exist at a subsymbolic level, and thus could not be discovered by humans through methodological development.

However, like a human-developed toolbox, trained deep models are inherently modular. Decomposing the computational graph of a deep model into subgraphs yields a module corresponding to each of these subgraphs. Thanks to the flexible compositional structure of neural networks, these subgraphs can in principle be used for multiple purposes. Thanks to its generic training scheme, deep multitask learning can be used to learn these modules and evaluate them for each purpose. Thus, deep multitask learning is a well-motivated approach for discovering multi-purpose modules.

Existing deep multitask learning approaches focus on discovering multi-purpose, monolithic feature extractors. Improving feature extraction is a core goal of deep learning (Lecun et al., 2015), but restricting multitask learning to sharing of this kind significantly constrains the kinds of functional regularities that can be discovered. In contrast, a more directly modular approach to deep multitask learning could discover generic functionality that monolithic systems and humans cannot. This modularity would lead to more flexible, higher-performing solutions that could be applied across the many deep learning application areas, and would align more closely with how functionality is organized in the real world.

1.2 Challenges

The pursuit of multi-purpose modules through deep multitask learning raises three key challenges that any practical method will have to address if it is to achieve the flexibility, adaptability, and efficiency that the modular approach promises. These challenges arise from the questions of *module form*, *module assembly*, and *module generality*.

First, the form of constituent modules will be integral to the design of the system. The natural definition of a deep learning module as a computational subgraph is so broad that it includes modules defined by individual learned parameters all the way up to modules that encompass the entire model for a task. By specifying the set of subgraphs that constitute modules, a system implies what scale of modularity it is looking for, and what kinds of modules it can discover. For example, in the deep learning setting, it is natural to define a module by a network *layer*; indeed, this is one of the approaches taken in this dissertation. As two more examples, existing deep multitask learning approaches define modules at the level of feature extractors, while some modular neuroevolution approaches, such as SANE (Moriarty & Miikkulainen, 1996) and

ESP (Gomez & Miikkulainen, 1997), define modules at the level of individual neurons. Finding a practical balance in scale is a key challenge: if modules are too simple, they may not be expressive enough to capture interesting regularities; if they are too complex, they approach the monolithic case, where it may be difficult for them to adapt to diverse purposes.

Second, the system will require a method that determines how modules are assembled into complete models for each task. From the multitask learning perspective, this is the question of *how to share* learned structure across tasks. How to assemble modules is related to the problem of designing deep learning architectures. Designing deep models for a single task is already a challenging problem that is being approached with automated techniques, since the complexity of many modern architectures is beyond what humans can design manually. Designing architectures that support multiple tasks adds another level of complexity to the problem, and determining which modules to use at which location in such an architecture complexifies things further. A key challenge of any system is to pair a space of possible constructions with a practical method for discovering effective constructions within this space. For example, in a very restricted assembly space, finding optimal constructions in this space may be easy, at the cost of diminishing the upper bound of system performance.

Third, and most importantly, a successful system must force resulting modules to be generic. In the trivial case, each module is used for only a single purpose and the system collapses to a standard deep learning model.

This collapse can be avoided by ensuring that modules are trained for multiple purposes, i.e., to solve sets of distinct pseudo-tasks (Chapter 3). Again, the potential of the system is determined by the scale of generality that can emerge. For example, a set of modules in which each solves only a small set of similar pseudo-tasks will be inherently less general than a set of modules in which each solves a large diverse set. To that end, one of the grand challenges of multitask learning is to successfully exploit regularities across seemingly disparate problems. In such a setting, there may be no intuitive way to construct fixed multitask architectures, so a more general and automated approach is required. In the case of diverse tasks, and even when tasks are apparently similar, care must always be taken to avoid *negative transfer*, i.e., when well-intentioned sharing of structure actually ends up reducing performance. Such degradation can occur when a module is trained to support more functionality than it is capable of expressing. The module may indeed be generic, in that it provides value for a diverse set of applications, but its value for each of those applications may be suboptimal. Enabling discovery of highly generic modules while avoiding negative transfer is thus a key challenge.

1.3 Approach

To solve these challenges of module form, assembly, and generality, the approach taken in this dissertation is built out from the deep multitask learning framework. First, this framework is generalized to provide a more unified perspective of how neural network modules can be trained for multiple purposes. These multiple purposes are formalized as *pseudo-tasks*. Within this framework, a progression of six learning systems is presented. Through this progression, the generality of modules is expanded, leading to practical solutions to module form and assembly at each step along the way. The progression proceeds by (1) testing the inherent generality of single modules; (2) improving the generality of single modules; (3) increasing the number of modules and breadth of generality; (4+5) scaling this generality to complex automatically-designed architectures; and (6) scaling this generality across diverse architectures and problem areas. Together, these systems constitute a comprehensive approach of how deep multitask learning can be used to discover multi-purpose modules.

The first system, General Reuse of Static Modules (GRUSM), is designed to answer the question: How general are trained modules inherently, i.e., when trained for a single purpose? In this system, a module trained initially for one purpose in a single task is reused as is, i.e., without modifying its parameters, for a different purpose at a different depth in another task. A concrete implementation is developed that uses a coevolutionary flavor of neuroevolution, ESP (Gomez & Miikkulainen, 2003), to train modules and incorporate them into new locations. The implementation, GRUSM-ESP, is evaluated in general video game playing, where each task corresponds to a game. In the experiments, reused modules take the form of a single layer of weights (of adaptable dimension). The results show that sometimes modules generalize well, sometimes they do not, and that this transferability can be predicted based on task characteristics. In particular, the modules trained in more complex tasks tend to generalize better. This makes sense, since modules that support more complex functionality will naturally contain more information that can be exploited. However, predicting module generalization is not as strong a tool as training more general modules in the first place.

With this knowledge that neural network modules have the potential to generalize across diverse purposes, the question is: How can they be forced to generalize better? The second system, Pseudo-task Augmentation (PTA), approaches this question from the foundational case where there is a single module that is simultaneously trained for many purposes. This module is a generic encoder with an arbitrary fixed topology, which is shared across all tasks and trained by gradient descent, as in classical deep multitask learning. To make this encoder more general, it is forced to solve each task in multiple distinct ways, by training it with multiple decoders for each task; each decoder defines a distinct pseudo-task. Training with additional pseudo-tasks is theoretically shown to expand the training dynamics of gradient descent. Methods are then introduced that interleave gradient descent with a coevolutionary process that controls pseudo-tasks to improve generalization. By increasing the number of ways a single core module is used, PTA is shown to improve performance across an array of deep models, including achieving state-of-the-art results on the CelebA multitask facial attribute recognition dataset.

With this knowledge that training a module in more ways can improve its generality, the question is: How far can this idea be taken? The third system, *soft ordering*, takes this idea to the extreme: Each layer in a deep architecture constitutes a module, and all modules are trained simultaneously across all possible locations in all tasks in a fixed architecture. Thus, each module must support functionality everywhere to some extent. At each location, the system learns a mixture of modules used at that location, while simultaneously learning the parameters of the modules themselves, and the complete optimization is performed end-to-end using gradient descent. The method is evaluated in vision and non-spatial domains, using convolutional and fully-connected layers, and demonstrates improvements over single-task and shared feature extractor approaches, including outperforming state-of-the-art deep multitask learning approaches on Omniglot multitask character recognition. Visualizations indicate that modules are indeed learning functional primitives, whose behavior is tuned to match the needs of particular contexts. These results suggest that simultaneously training modules for many kinds of purposes across multiple tasks is a promising approach to discovering a compact set with generic functionality.

However, the soft ordering method does not scale well, because all modules are executed at each location in the joint model during each forward and backward pass. Scaling this approach requires a way to automatically select which module to use at each location during training. In the remaining three systems, such selection methods are used to scale soft ordering in two orthogonal directions: (1) to more complex multitask architectures discovered by neural architecture search; and (2) to sharing across diverse classes of architectures and task modalities. The fourth and fifth systems follow the first of these directions, and the sixth follows the second.

The fourth system, Coevolution of Task Routing (CTR), uses evolution to discover where each module should be applied, and uses gradient descent to train their parameters. That is, evolution and gradient descent are interleaved in a manner similar to PTA. In CTR, starting from a minimal architecture for each task, evolution expands its use of modules incrementally so that the correct amount of complexity can be achieved. Modules in CTR can also take on more generic functionality than in soft ordering, since evolution discovers different kinds of architectures for different tasks, so modules are trained for more diverse pseudo-tasks. Two new key mechanisms make the system practical: (1) All candidate models for all tasks are trained jointly, so module semantics are preserved across generations; and (2) task architectures are coevolved, allowing more efficient optimization of the multitask architecture. In experiments, this system demonstrates marked improvement over soft ordering.

The fifth system, Coevolution of Modules and Task Routing (CMTR), is a direct generalization of CTR, in which modules are no longer restricted to being single neural network layers. In this system, along with the optimization of how modules are assembled for each task, the topologies of the modules themselves are optimized. Module topologies are optimized in an outer loop around CTR, using a variant of CoDeepNEAT (Miikkulainen et al., 2017), a popular evolutionary architecture search algorithm, which in turn is incremental, coevolutionary, and explicitly modular. By making module topologies more flexible, CMTR yields significant improvements over CTR.

Though they improve performance in many settings, PTA, soft ordering, CTR, and CMTR cannot be used to share modules across *modalities*, i.e., when the data for different tasks have a fundamentally different structure, for example, vision vs. language. This restriction arises because, in these approaches, modules are defined as layers or graphs of layers, so they can only be applied where their input-output specification is satisfied, both technically and semantically. For example, the spatial semantics of a 2D convolutional layer are lost when applying this layer to non-spatial input.

The sixth system, Modular Universal Reparameterization (MUiR), overcomes this restriction. It supports sharing of modules across arbitrary deep architectures and task modalities, allowing regularities to be exploited across diverse problem areas. This system decomposes the parameters of a given architectures for a set of tasks into a set of equally-sized linear maps. The parameters of each map are then generated by a *hypermodule*, which reduces the parameters to a small number of degrees of freedom. Hypermodules generalize the modules of the other systems by allowing each module to be marginally tuned for different purposes. This flexibility can be especially valuable when applications are highly diverse. The mapping of modules to locations is optimized in a manner similar to CTR, incrementally increasing sharing by interleaving gradient descent and evolution. However, for this system, coevolution of assembly occurs not across tasks, but across all module locations, i.e., across all pseudo-tasks. Coevolving at this level yields theoretically-grounded speedups that are especially helpful when the number of pseudo-tasks is large. Coevolution is implemented with a surrogate fitness function that uses the mixture-learning mechanism of soft ordering. Experiments demonstrate intriguing dynamics of MUiR, including positive sharing across tasks with fundamentally different modalities, and the emergence of surprising sharing behaviors. Importantly, by supporting sharing of modules across modalities, MUiR is especially valuable when a task with a new modality arises with only a small amount of data, for example, with temporal data collected from a new kind of geosensor, or a rare disease detectable by data from a new kind of medical device. In such a case, MUiR can boost models for the new modality by harnessing generic functionality discovered from vast datasets and problem repositories for more common modalities.

Since modules can now be shared across diverse architectures and modalities, and improved by optimizing their topologies and the topologies in which they are applied, a natural extension would be to combine these features in an approach that optimizes cross-modal architectures to make modularity even more effective. Such an approach would be a straightforward unification of CMTR and MUiR, and is left for future work.

Overall, by progressively increasing module generality, and developing practical methods for assembling modules of various forms along the way, these six systems verify the value of the deep multitask learning approach to discovering multi-purpose modules at a level that humans cannot. These developments provide a foundation for developing future systems that combine their advantages towards a fully-general and robust module discovery algorithm that continuously refines itself to efficiently construct high-performing solutions to a broad range of critical real-world applications.

1.4 Guide to the Reader

The remainder of this dissertation is organized as follows:

Chapter 2 covers the foundations for the methods developed in this dissertation, including neuroevolution, deep learning, multitask learning, other forms of parameter sharing.

Chapter 3 introduces the framework that formalizes modules serving multiple purposes as functions solving pseudo-tasks, and discusses design choices that must be considered when developing a system within the framework, along with methods for evaluating such systems.

Chapters 4-8 describe the six systems, including their development, implementation, and evaluation: Chapter 4 covers General Reuse of Static Modules; Chapter 5 covers Pseudo-task Augmentation; Chapter 6 covers soft ordering of shared layers; Chapter 7 covers Coevolution of Task Routing, along with Covevolution of Modules and Task Routing; and Chapter 8 covers Modular Universal Reparameterization.

Chapter 9 reviews the systems and their applications, discusses tradeoffs that emerged, and promising avenues of future work, including taking an explicitly ecological perspective, searching for modules offline, and extending the systems to the setting of lifelong learning.

Finally, Chapter 10 reviews the contributions of this dissertation and concludes that a modular approach to deep multitask learning is a practical paradigm for discovering highly generic functionality that goes beyond what is possible by humans.

Chapter 2

Background

This chapter reviews the foundations and related work for the framework and systems developed in the subsequent six chapters. First, two neural networks methodologies are reviewed: neuroevolution and deep learning. These methodologies provide powerful and well-established toolsets for developing systems that uncover functional modules. Then, multitask learning is reviewed, which provides the main problem definition considered in this dissertation. It is discussed alongside other methods that improve generalization by training multiple deep models. These methods set the background for the framework developed in Chapter 3, which enables discovery of multi-purpose functional modules by training and applying modules across different contexts.

2.1 Neuroevolution

Neuroevolution (Fogel et al., 1990; Yao, 1999; Floreano et al., 2008; Miikkulainen, 2016) is one approach to designing and training neural networks. It is grounded in methods of evolutionary computation (Bäck et al., 1997; Eiben & Smith, 2003; De Jong, 2006), and was traditionally used to evolve the weights of fixed structure neural networks directly (Yao, 1999). The general framework

Algorithm 2.1 Generic Neuroevolution Framework

```
Initialize population of network representations from random distribution
Forever:
    Assemble representations into functional networks
    Evaluate all networks in task environment
    Refine population by removing low performers
    Generate new representations from top performers
```

for the classical neuroevolution algorithm is shown in Algorithm 2.1. The basic idea is to iteratively generate networks that have a high chance of success, by recombining the best networks found so far and discovering improvements in their local neighborhoods. The *initialization* step starts with a population that already covers some interesting and complementary areas of the search space. The *assembly* step captures the fact that the substrate that is actually evolved may not be a functional network, but rather the data that describes it, e.g., the vector of its weights, or some more structured representation like that one seen in Figure 2.1. The *evaluate* step is where the models are actually run on the task of interest, and their fitnesses assigned; this step can often be trivially parallelized for practical efficiency. The *refine* step focuses search on the most promising areas of the search space, and the *generate* step creates new individuals based on recombination and variation of current high performers. Neural networks are a good substrate for evolution, because they implement smooth functions and tend to encode redundant distributed information, so small changes to their weights tend not to break their functionality. This robustness stands in contrast to more brittle substrates, such as evolving rigid physical structures.

Because it uses a population of neural networks is optimized simultaneously, neuroevolution performs global search in the space of models. This allows it to avoid the pitfalls of local optima, which is an advantage over gradient descent. This feature is especially useful in the case of relatively small models, where such optima can occur frequently. Avoiding local optima is often also a big advantage in sequential decision-making problems, where it can be difficult for an agent to explore its way out of deceptive sinks in a complex environment (Lehman & Stanley, 2011a). Even in its most simple forms, neuroevolution is competitive with state-of-the-art deep reinforcement algorithms (Salimans et al., 2017; Such et al., 2017; Gaier et al., 2018), which is promising since it has been relatively underexplored.

Also in contrast to deep learning, training parameters through evolution does not require a differentiable loss function for a task. Therefore, it can be used when no such natural loss function is available, as is the case in sequential decision-making problems. This general applicability makes neuroevolution especially adept at evolving the *structure*, i.e., topology, of networks (Angeline et al., 1994; Vittorio, 1994; Stanley & Miikkulainen, 2002). Evolving on the single-neuron scale, starting with a minimal topology and incrementally complexifying over a run of evolution was shown to ease learning, especially in more complex domains (Gomez & Miikkulainen, 1997; Hausknecht et al., 2013; Schrum & Miikkulainen, 2016). Coevolution of neurons as the basic building block of neural networks also showed remarkable success, with canonical examples being the ESP (Gomez & Miikkulainen, 1997, 1999, 2003) and SANE (Moriarty & Miikkulainen, 1996, 1997; Richards et al., 1998) algorithms.

SANE evolves a population of neurons, along with a population of *blueprints*, each of which specifies which neuron to use at which location. Say the population of neurons f_i is given by $\{f_1, \dots, f_N\}$, and the architecture has a single hidden layer of size H. Then, the complete model $\mathcal{M}_b$ for a given blueprint $b : \{1, \dots, H\} \rightarrow \{1, \dots, N\}$ is given by:

$$\mathcal{M}_b(\boldsymbol{x}) = \phi(\sum_{i=1}^{H} f_{b(i)}) \tag{2.1}$$

where ϕ is an optional nonlinearity, e.g., sigmoid, applied before the output. The performance of this model gives the blueprint its fitness, which is propagated down to neurons based on the blueprints in which they were included. A functional depiction of the SANE approach is given in Figure 2.1. By optimizing this set of useful modules which can be used by any blueprint in the population, SANE increases search efficiency, since this modular information is transmitted more quickly through the search process. In this approach, the size of the module population is larger than the number of modules used in each network, and each module is not expected to be used more than once in the final best model. The main purpose of the modularization is to accelerate optimization of a single task by sharing parameters throughout the population.

Evolving at a slightly higher level, coevolution of hierarchical modules discovers compact sets of modules with complementary functionality (Moriarty & Miikkulainen, 1998; Reisinger et al., 2004; Li & Miikkulainen, 2014). Related

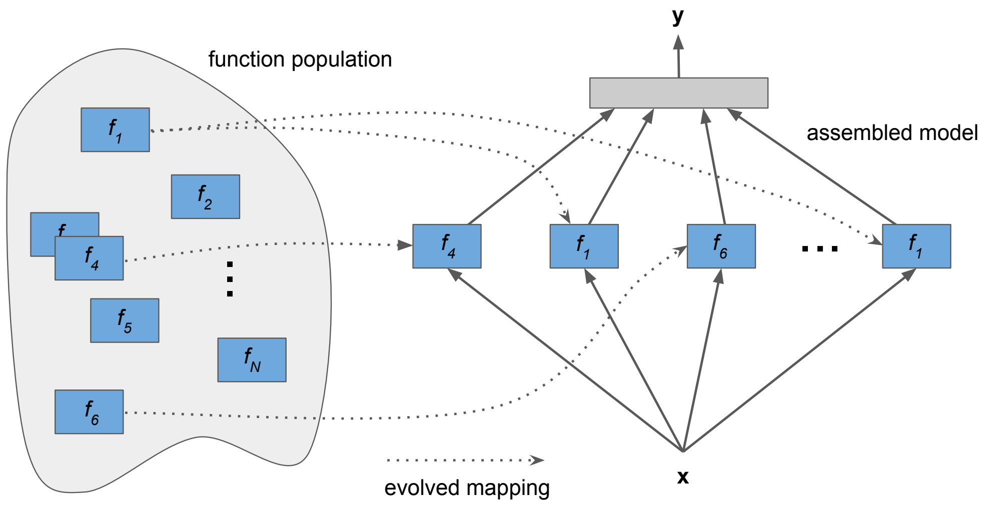


Figure 2.1: **Functional View of SANE.** This figure gives an overview of the SANE (Symbiotic Adaptive Neuroevolution) architecture (Moriarty & Miikkulainen, 1996). A set of N functions is evolved (in practice, individual neurons), along with a set of blueprints, each of which specifies which function to use in each function location in a fixed architecture neural network. Blueprints receive fitness based on the performance of the assembled model, and fitness is percolated down to individual functions based on the performance of blueprints in which they are present. This architecture has been repeatedly shown to outperform evolution of a monolithic neural network representation, as well as many popular reinforcement learning methods.

methods have also been used in cases where the modularity and reuse is more explicit at a higher symbolic level: to transfer learned functionality to more complex tasks (Whiteson et al., 2005; Taylor et al., 2007; Verbancsics & Stanley, 2010; Braylan & Miikkulainen, 2016), to reuse functionality across various modes of operation (Pugh & Stanley, 2013; Schrum & Miikkulainen, 2016; Huizinga & Clune, 2018), and to evolve controllers for modular teams of agents (D'Ambrosio et al., 2010; Nitschke et al., 2012; Bryant & Miikkulainen, 2018).

Recently, a similar class of modular methods has been adopted to evolve the architectures of deep models (Miikkulainen et al., 2017; Liu et al., 2018; Real et al., 2018). By taking advantage of the complementary advantages of deep learning and neuroevolution, these methods are able to scale automated design of neural networks to a broad range of real world problems, even breaking the benchmarks set by human design on the most battered largescale problem: ImageNet (Real et al., 2017, 2018). These methods will be reviewed further in Section 2.3.2. The evolutionary approaches developed in this dissertation operate at a similar scale.

Some of these architecture search methods, as well as other neuroevolution methods, rely on *interleaving* evolution and training through gradient descent (Whiteson & Stone, 2006; Parker & Bryant, 2009; Real et al., 2017), combining the ability of evolution to explore a broad space of solutions and that of gradient descent to efficiently optimize around a current solution. From the analogy to natural evolution, this interleaving is an instance of Lamarckian evolution, in which individuals genetically transfer what they have learned to their offspring. Whatever the true magnitude of Lamarckian evolution in nature, in neuroevolution interleaving can provide substantial benefits to efficiency, and as such it is used in several of the systems developed in this dissertation.

Neuroevolution is one among many classes of evolutionary processes that solve complex problems by discovering functionality incrementally (Bonner, 1988; Iqbal et al., 2014). This feature makes evolution an appropriate tool for systems that iteratively improve how they use functional modules to solve multiple problems, while refining the modules themselves. Evolutionary approaches also benefit from modern techniques from the evolutionary computation field that can be applied to any evolutionary system. One particularly useful class of techniques is behavioral methods, which optimize a population with respect to a higher-dimensional space of behaviors based on what a model actually *does* in its task environment, as opposed to looking at the loss or fitness alone (Lehman & Stanley, 2011a; Mouret & Doncieux, 2012; Pugh et al., 2016; Meyerson & Miikkulainen, 2017). For example, *novelty search* (Lehman & Stanley, 2011a) evaluates individual networks based on how different their behavior is from that of other networks evaluated so far. More precisely, the fitness score of an individual is replaced by a novelty score, given by a density estimate in the behavior space:

$$\text{novelty}(x) = \frac{1}{k} \sum_{i=1}^{k} d(b(x), b(y_i)) \tag{2.2}$$

where b is a function that maps each individual to its behavior vector, d is some distance metric between behaviors (e.g., L2 distance), and y_i is the ith nearest behavioral neighbor to x among the current population and an archive of previously evaluated individuals. Thus, by optimizing with respect to the novelty score, evolution will be driven to explore areas of the search space with behavior qualitatively different from anything seen before. Similar to discovering multi-purpose modules by solving diverse tasks, novelty search aims to discover powerful functionality by collecting diverse experiences. Related to the multitask learning approach taken in this dissertation, there are existing approaches that takes advantage of data from multiple tasks to learn a better behavior function for neuroevolution (Meyerson et al., 2016). However, the shared knowledge discovered by these algorithms are stored in the evolutionary operators themselves, not neural network modules. Finally, apart from their benefit when included in optimization schemes, behavioral evolutionary methods have exhibited notable results in machine creativity (Secretan et al., 2008; Lehman & Stanley, 2011b; Nguyen et al., 2015; Lehman et al., 2018). Existing generic modules have been discovered through human creativity, so artificial creativity should be useful for machines to have the same success.

2.2 Deep Learning

For investigating the extent to which functionality can be generalized across diverse contexts, deep learning is a good option. Deep learning has emerged in recent years as the dominant approach for solving machine learning problems across a range of important subfields in artificial intelligence, including computer vision (Krizhevsky et al., 2012; Jia et al., 2014; Szegedy et al., 2016), natural language processing (Collobert & Weston, 2008; Suutskever et al., 2014; Zhang et al., 2015), speech recognition (Hinton et al., 2012; Graves et al., 2013; Chan et al., 2016), reinforcement learning (Mnih et al., 2015a,b; Lillicrap et al., 2015), and even more classical unstructured machine learning problems (Klambauer et al., 2017).

Even before deep learning, neural networks were a workhorse of machine learning due to their general applicability. The ability to approximate any function (Hornik et al., 1989), along with an efficient training algorithm (Rumelhart et al., 1986), and a plethora of successful applications (Fukuda & Shibata, 1992; Hussain, 1999; Christodoulou & Georgiopoulos, 2000), made neural networks a good option for many problems. That said, for many years, it was difficult to up neural networks to large, complex problems. Along with the dramatic increase in computational resources over the past few decades, several important ideas were recently introduced that made such scale possible. They include more principled parameter initialization (Glorot & Bengio, 2010; He et al., 2016) activation functions (Glorot et al., 2011; Klambauer et al., 2017), and optimization schemes (Tieleman & Hinton, 2012; Kingma & Ba, 2014). These optimization schemes are based on stochastic gradient descent (SGD), which moves the weights of any differentiable function in the direction of steepest descent with respect to its current empirical loss. For a model $\mathcal{M}$ with parameters $\theta_{\mathcal{M}}$ solving a task with N samples $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$, this weight update step is given by

$$\theta_{\mathcal{M}} := \theta_{\mathcal{M}} - \frac{\alpha}{N} \sum_{i=1}^{N} \nabla_{\theta_{\mathcal{M}}} \mathcal{L}(\mathcal{M}(\boldsymbol{x}_i; \theta_{\mathcal{M}}), \boldsymbol{y}_i), \tag{2.3}$$

where $\mathcal{L}$ is a differentiable *loss function* indicating the distance of the prediction $\hat{\boldsymbol{y}}_i = \mathcal{M}(\boldsymbol{x}_i; \theta_{\mathcal{M}})$ from the target $\boldsymbol{y}_i$, and α is the *learning rate* which controls how far the optimizer steps in the direction of the gradient. The algorithm for computing these gradients for arbitrary neural networks using the chain rule is known as *backpropagation* (Rumelhart et al., 1986). Modern optimizers are all variants of this basic iterative algorithm, but include some additional complexity, for example, in the form of momentum, learning rate decay, or adaptive per-parameter learning rates. Overall, the goal of optimization is to find $\theta^\star_{\mathcal{M}}$ that minimizes the loss over the true distribution, or analogously, maximizes the fitness of the model in its environment.

Now that reliable methods have been established, and powerful open-source tools have emerged for constructing and training deep networks, including Tensorflow (Abadi et al., 2015), PyTorch (Paske et al., 2017), Keras (Chollet et al., 2015), and CuDNN (Chetlur et al., 2014), a new standard level of abstraction has appeared at the level of layers and tensors. Instead of looking at neural networks at the neuron level, these tools view them more compactly as directed computational graphs in which each node is a potentially nonlinear tensor transformation. Each such tensor may contain thousands or millions of values. At this higher level there is a standard collection of building blocks that are chained together to form most deep models. Standard modules include the classic fully-connected layer, convolutional layers (Lecun & Bengio, 1995), recurrent layers (in particular, LSTM (Hochreiter & Schmidhuber, 1997)), attention blocks (Vaswani et al., 2017), residual blocks (He et al., 2016), and regularization functions such as dropout (Srivastava et al., 2014) and batch normalization (Ioffe & Szegedy, 2015). For example, the layer of neurons from a classical neural network has been abstracted into a *fully-connected layer* or *dense layer* function, given by the affine transformation

$$f(\boldsymbol{x}) = \phi(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}), \tag{2.4}$$

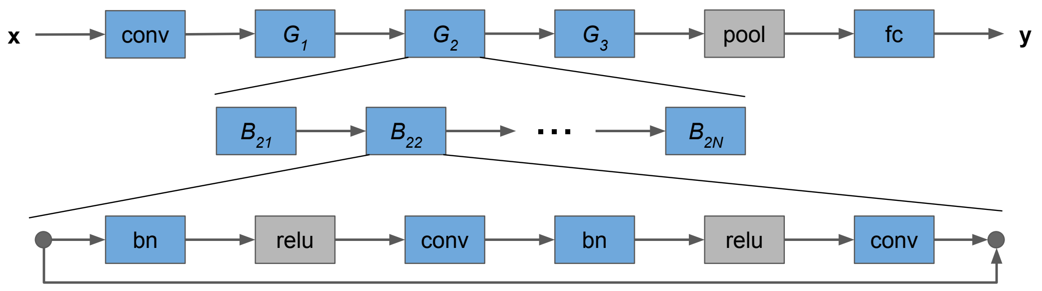


Figure 2.2: **Wide ResNet.** As a relatively simple example of a state-of-the-art deep architecture, this figure gives a graphical depiction of the Wide Residual Network architecture family (Zagoruyko & Komodakis, 2016). Functions containing trainable parameters are shown in blue. The architecture contains three groups G_i of N residual blocks $B_{i1}, \ldots, B_{iN}$. In turn, each block contains a pair of batch normalization layers, ReLU activation layers, and 2D convolutional layers, as well as a short-cut connection that allows information to pass through the block untouched. This example illustrates key features of modern neural network architectures: depth, modular design, shortcut connections, and various standard layer types.

where $\boldsymbol{W}$ is a weight matrix, and $\boldsymbol{b}$ is a vector of biases, both trained through backpropagation, and ϕ is an optional elementwise nonlinearity.

One illustrative example of a complete model constructed from such components is shown in Figure 2.2. This figure gives a graphical depiction of the Wide Residual Network architecture (Zagoruyko & Komodakis, 2016), which is a relatively simple, and yet very high-performing and versatile, family of convolutional neural networks, originally developed for image recognition, and then reused for other computer vision problems. This architecture captures many of the themes of modern deep architectures, including depth, modularity, short-cut connections, and various standard components, i.e., 2D convolutions, batch normalization, global average pooling, and a fully-connected final classification layer. In the architecture in Figure 2.2, human designers have compressed the structural design space by working with multiple levels of modular hierarchy. This indicates that information at different depths in the architecture benefit from similar transformations. Furthermore, studies probing residual networks have discovered that sets of their convolutional layers often have a high amount of mutual information (Greff et al., 2016). These observations suggest that convolutional networks can benefit not just from topological regularity, but more direct functional sharing as well. Despite recently developed improvements to convolutional networks, the fundamental building block of these networks remains the *convolutional layer*. Convolutional layers were originally developed for processing images, but they can operate over spatial input of any dimension (Lecun & Bengio, 1995). In its standard form, a 2D convolutional layer f performs an operation on an input $\boldsymbol{X} \in \mathbb{R}^{\text{width}\times\text{height}\times m}$, i.e., with m *input channels*, which produces $f(\boldsymbol{X}) \in \mathbb{R}^{\text{width}\times\text{height}\times n}$, where

$$f(\boldsymbol{X})_{ij} = \sum_{a=0}^{A-1}\sum_{b=0}^{B-1} \boldsymbol{W}_{ab}^{\top}\boldsymbol{X}_{(i+a)(j+b)} + \boldsymbol{b} \tag{2.5}$$

for all $(i, j) \in \{1, \dots, \text{width}\} \times \{1, \dots, \text{height}\}$, where $\boldsymbol{W} \in \mathbb{R}^{A\times B\times m\times n}$ is a 4-dimensional tensor of learned weights with spatial dimensions $A \times B$ (with boundary-handling omitted for clarity), and $\boldsymbol{b} \in \mathbb{R}^n$ is an optional learned bias vector. Notice that, like the classic fully-connected layer, the convolutional layer is built from matrix products, but here they are combined in a more involved way: each output value is computed from an $A \times B$ receptive field of the input. In other words, convolutional layers are designed to *share parameters* *across space*. For example, this layer uses the same parameters to compute the output of the bottom left of an image (or intermediate 2D representation) as it does the upper right. This translational invariance is one reason convolutional networks are viewed as *feature extractors*: they search for the existence and location of a set of features across the input space. Humans have designed this parameter-sharing mechanism based on evidence that the same sort of sharing happens in the human visual cortex (Lee et al., 2008). A goal of this dissertation is to discover other valuable ways to share parameters that humans could not discover through such analysis.

Another canonical and complementary example of a modern neural network architecture is the LSTM (Hochreiter & Schmidhuber, 1997), which has led to performance breakthroughs in several areas of artificial intelligence, including natural language and speech. In contrast to the convolutional layer, which operates simultaneously over space, the LSTM is a recurrent architecture, which operates iteratively over *time*. A functional diagram of the standard LSTM is given in Figure 2.3. At the tth timestep, the LSTM's output is a function of the current input $\boldsymbol{x}_t$, its previous hidden output $\boldsymbol{h}_{t-1}$, and the previous state of its memory cell $\boldsymbol{c}_{t-1}$. To enable memory to be cleanly stored, updated, and recalled over extended periods of time, the LSTM includes multiplicative gating mechanisms: an input gate, which determines how much of the current input should be written to memory; a forget gate, which determines how much of the previous memory state should be discarded; and an output gate, which determines how much of the current memory should be read as

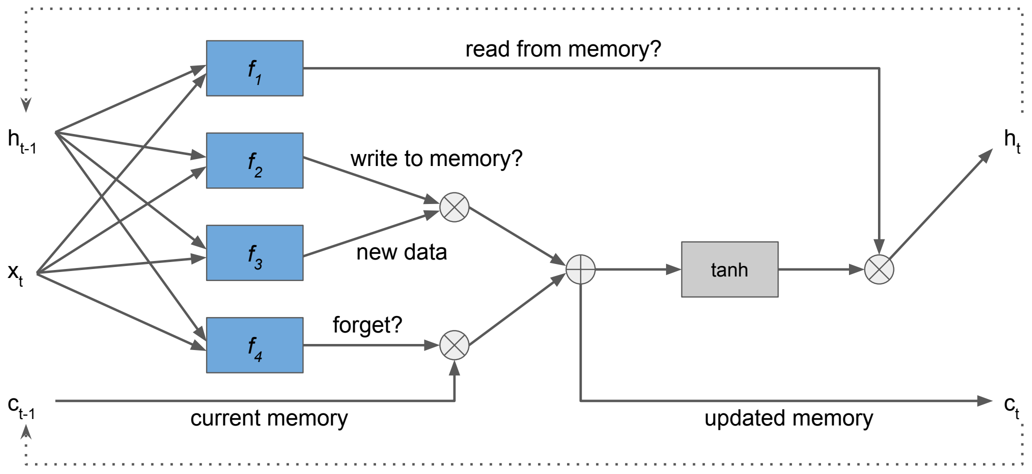


Figure 2.3: **LSTM Architecture.** This figure gives a functional depiction of the standard LSTM architecture. In this figure, $\boldsymbol{x}_t$ is the input, $\boldsymbol{h}_t$ is the output, and $\boldsymbol{c}_t$ is the cell state, or *memory*, at the tth timestep. Functions in blue contain trainable parameters. $\oplus$ denotes elementwise addition, and $\otimes$ denotes elementwise multiplication, which enables the implementation of *gates*, for deciding how much to read from, update, and forget memory state. The construction of these gates rely on learning four functions f_1, f_2, f_3, f_4, which are implemented as fully-connected layers of equal shape. Dotted arrows indicate the recurrence induced by applying this function iteratively over many time steps. The LSTM architecture serves to illustrate the kinds of modular designs arising from humans manually encoding knowledge about the world.

output. The complete equations describing the operation at a single timestep are as follows:

$$\boldsymbol{f}_t = f_1([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]) \qquad \text{(forget gate)}$$

$$\boldsymbol{i}_t = f_2([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]) \qquad \text{(input gate)}$$

$$\boldsymbol{o}_t = f_4([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]) \qquad \text{(output gate)}$$

$$\boldsymbol{c}_t = (\boldsymbol{f}_t \otimes \boldsymbol{c}_{t-1}) \oplus (\boldsymbol{i}_t \otimes f_3([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}])) \qquad \text{(cell update)}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \oplus \tanh(\boldsymbol{c}_t) \qquad \text{(hidden output)}$$

where $\boldsymbol{x}_t$, $\boldsymbol{h}_t$, and $\boldsymbol{c}_t$ are the input, hidden output, and updated memory cell state of the LSTM at the tth timestep, $[\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]$ denotes the concatenation of $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$, and f_1, f_2, f_3, f_4 are implemented with fully-connected layers. The nonlinearity used by f_3 is usually the hyperbolic tangent function (*tanh*), or simply the identity function, while f_1, f_2, f_4 use a sigmoid nonlinearity so that they implement multiplicative gating.

Notice again that the operation that uses learned parameters in these equations the same affine transformation as that of the fully-connected layer. The LSTM is a way of linking these operations together to implement a memory architecture inspired by how memory is controlled in computer hardware, and, theoretically, in the human brain. Since the same learned parameters are applied at each timestep, we can say that LSTMs *share parameters over time.* That is, in principle, there could be separate parameter matrices applied at each timestep, but humans have discovered that temporal sharing is indeed a powerful regularizer. Again, we expect that machines will be able to discover powerful parameter linkages that humans cannot. However, as of yet, despite recent results that automatically discover new architectures, e.g., in the case of LSTMs (Zoph & Le, 2017; Rawal & Miikkulainen, 2018) and convolutional blocks (Real et al., 2018), the components in the standard deep learning toolbox have all been discovered manually by humans.

That said, improvements, variations, and advancements upon this standard toolbox are being made continuously, taking advantage of the modular subcomponents used to construct these tools, namely standard primitive differentiable tensor operations. By using these primitive operations to build and improve broadly-applicable tools, deep learning has demonstrated the power of functional modularity. This power makes it a promising substrate for discovering further generic components automatically. The modules that learn and store the majority of the information in a deep model contain learnable parameter tensors whose scale determine the expressivity of the model. These parameters are trained from scratch so they can take on any possible values when a new model is trained for a particular application. So, the current mainstream deep learning mindset is that a particular composition and learned parameterization of a compact set of high-level high-dimensional components can effectively solve most machine learning problems.

However, although the set of parameterizable components may be small, they have a high number of parameters, and therefore the set of functions they can represent is still highly unconstrained. The goal of the methods developed in this dissertation is to find sets of components whose parameters are not completely unconstrained, but forced into a subspace where they must support more generic functionality. We have already seen how convolutional and recurrent layers used a hand-designed subspace to force parameters to be used in multiple scenarios, i.e., to be used identically over space and time, respectively. Complementary to these geometric parameter-sharing methods, several approaches have been investigated for compressing the parameters of deep models, motivated by evidence that learned parameters are highly correlated. One early approach showed that the majority of parameters in deep networks could be generated from predictions based on a relatively small set of parameters, without hurting performance (Denil et al., 2013). This observation, along with theoretical compression results, led to the development of methods that parameterize weight tensors with lower-dimensional parameterizations (Chen et al., 2015; Cheng et al., 2015; Yang et al., 2015; Li et al., 2018). In one extreme version of such compression, the parameters of an entire network are viewed as a single long vector $\theta \in \mathbb{R}^M$, and are generated by a fixed random projection $\boldsymbol{P} \in \mathbb{R}^{M \times D}$ and fixed random offset $\boldsymbol{v} \in \mathbb{R}^M$ applied to a relatively low-dimensional vector $\boldsymbol{z} \in \mathbb{R}^D$, i.e., $D << M$, and

$$\theta = \boldsymbol{P}^\top \boldsymbol{z} + \boldsymbol{v} \tag{2.6}$$

Since the only trainable parameters are in $\boldsymbol{z}$, the *intrinsic dimension* of the entire model is D, even though the model can implement the complete function of a deep neural network (Li et al., 2018). Experiments with this intrinsic dimension approach showed that many common task-architecture pairs could be solved by setting D much smaller than M; sometimes thousands of times smaller. The success of such compression approaches is encouraging evidence that there is a high amount of regularity in learned parameters of deep models that is ripe for exploitation. The result of such approaches is more parsimonious model representations, in which sets of parameters can be seen as having more general applicability, since they are used to parameterize multiple locations within a single model.

Towards an even greater level of generality, another approach, and the one pursued in this dissertation, is *multitask learning*, in which parameters are shared across multiple tasks, so that their functionality becomes more general than that required for solving only one task. Multitask learning is reviewed in the next section, in the context of methods that train multiple deep models.

2.3 Training Multiple Deep Models

The methods developed in this dissertation all train multiple deep models in order to discover generic functional modules. When searching for improved generalization, the modules that find success across many of these models emerge as successful generic building blocks. There is a broad range of methods that exploit synergies across multiple deep models. This section reviews these methods by classifying them into three types: Class 1: methods that jointly train a model for multiple tasks; Class 2: methods that train multiple models separately for a single task; and Class 3: methods that jointly train multiple models for a single task. The high level ideas of these three methods are depicted in Figure 2.4.

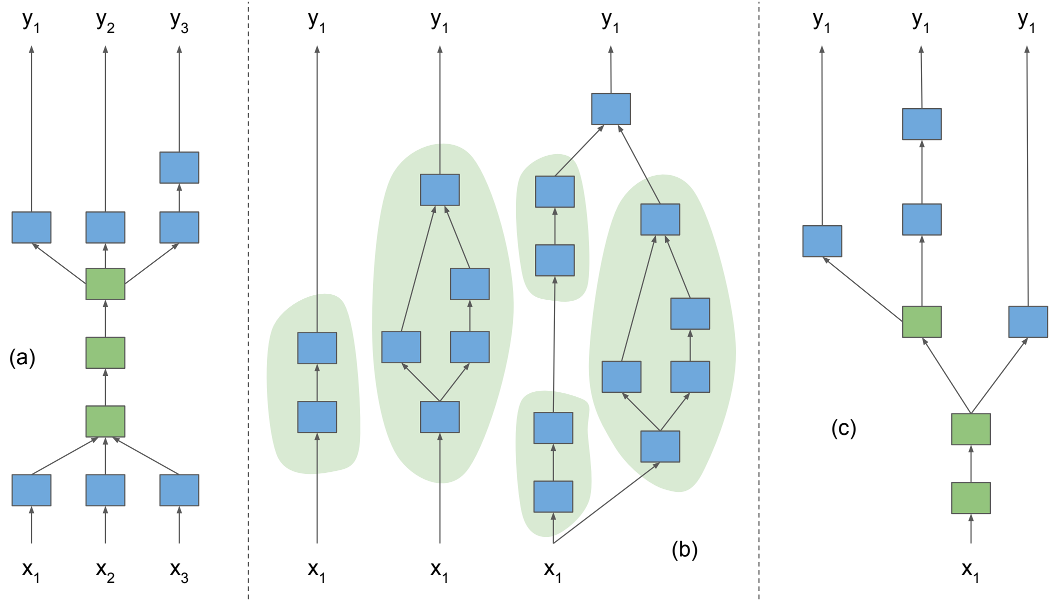


Figure 2.4: **Three Paradigms that Train Multiple Deep Models.** This figure gives a high-level view of three classes of methods that involve training multiple deep models. (a) *Joint training of models for multiple tasks* trains a distinct model for each task, and these models can share some of there trained parameters, e.g., through shared layers (shown in green). This is the foundational setup considered in this dissertation; (b) *Separate training of multiple models for a single task* trains several disjoint models with the goal of finding a single best model for one task; reused topological discoveries are circled in green. Such model search techniques are used in this dissertation for learning how to better share learned structure; (c) *Joint training of multiple models for a single task* trains several models for a single task that share parameters (again shown in green), on the way to finding a single best model for that task. This third class of methods provides a bridge between a and b, motivating the development of techniques that combine their advantages to discover more general neural network modules.

Class 1 gives the background and setup for multitask learning. This is the key problem formulation that is used throughout the systems and experiments in the subsequent chapters. Class 2 covers non-multitask model search methods. Although not focused on multitask learning, these methods provide solutions for *how* to automatically construct high-performing models out of sets of existing building blocks, and thus gives a foundation for the assembly methods investigated in subsequent chapters. Finally, Class 3 is given as a connector between Classes 1 and 2. Overall, the review below motivates the development of methods in Class 3 that unify the advantages of Classes 1 and 2. By showing how multiple models can be jointly trained for a single task, Class 3 motivates the development of the framework in Chapter 3, which gives a broad perspective of how functional modules can be used to serve different purposes.

2.3.1 Joint training of models for multiple tasks

Joint training of models for multiple tasks, i.e., *multitask learning* (MTL), was proposed decades ago (Caruana, 1998), to improve overall performance by exploiting regularities present across tasks. There are many real-world scenarios where harnessing data from multiple related tasks can improve overall performance. In general, in multitask learning, there are T tasks $\{\{\boldsymbol{x}_{ti}, \boldsymbol{y}_{ti}\}_{i=1}^{N_t}\}_{t=1}^{T}$, where N_t is the number of samples for the tth task. Note that it is possible that for $t_1 \neq t_2$, $N_{t_1} \neq N_{t_2}$, $\dim(\boldsymbol{x}_{t_1}) \neq \dim(\boldsymbol{x}_{t_2})$, and/or $\dim(\boldsymbol{y}_{t_1}) \neq \dim(\boldsymbol{y}_{t_2})$. The only requirement for multitask learning to be useful is that there is some amount of information shared across tasks, and, in theory, this is always the case (Mahmud & Ray, 2008; Mahmud, 2009).

The original substrate for multitask learning was the traditional neural network setting, i.e., with a single hidden layer (Caruana, 1998). In this model, related tasks shared knowledge by sharing their input-to-hidden layer, and having separate hidden-to-output layers. In this way, the shared first layer of the joint model would be forced to learn more generic representations of the world that were practical for all tasks. Since then, multitask learning has found success across other machine learning substrates, including trees Jaśkowski et al. (2008); Mahmud & Ray (2008), probabilistic models (Bonilla et al., 2008; Durrett & Klein, 2014), and neuroevolution (Schrum & Miikkulainen, 2012).

Although originally introduced in a neural network setting, substantial progress was made from more classical perspectives in the linear setting (Evgeniou & Pontil, 2004; Argyriou et al., 2008; Kang et al., 2011; Kumar & Daumé, 2012). For linear models, different multitask learning variants are implemented as different forms of explicit regularization terms applied across task models. The choice of regularization method implies what kinds of sharing are possible. For example, in the simplest case, tasks were assumed to be related and their weight vectors encouraged to be similar with respect to L2 distance (Evgeniou & Pontil, 2004). Many subsequent approaches took a perspective more similar to the neural network approach, in which the model for each task is decomposed into a factor that is shared across multiple tasks, and a factor that is task-specific (Argyriou et al., 2008). In this framework, it is possible to implement regularization that models desirable multitask learning features, such as automatically learning groupings of related tasks (Kang et al., 2011), and modeling structural overlap across groups (Kumar & Daumé, 2012), which avoid pitfalls that may otherwise lead to negative transfer, i.e., when sharing hurts more than it helps. As neural networks have regained popularity, many of these linear approaches have been reunified from a neural networks perspective (Yang & Hospedales, 2015). Overall, the factorization approach to multitask learning leads naturally to deep learning, since deep models inherently factorize models over their constituent layers.

Indeed, in recent years, MTL has been extended to deep learning, in which it has improved performance in applications such as vision (Zhang et al., 2014; Bilen & Vedaldi, 2016; Misra et al., 2016; Rudd et al., 2016; Lu et al., 2017; Rebuffi et al., 2017; Yang & Hospedales, 2017), natural language (Collobert & Weston, 2008; Dong et al., 2015; Liu et al., 2015a; Luong et al., 2016; Hashimoto et al., 2017), speech (Huang et al., 2013; Seltzer & Droppo, 2013; Huang et al., 2015; Wu et al., 2015), reinforcement learning (Devin et al., 2016; Fernando et al., 2017; Jaderberg et al., 2017b; Teh et al., 2017), and even seemingly unrelated tasks from disparate domains (Kaiser et al., 2017). Deep MTL relies on training signals from multiple datasets to train deep structure that is shared across tasks. Since the shared structure must support solving multiple problems, it is inherently more general, which leads to better generalization to holdout data.

Though more sophisticated methods now exist, the most common approach to deep multitask learning is still based on the original work using neural networks, in which a joint model is decomposed into an underlying model $\mathcal{F}$ (parameterized by $\theta_{\mathcal{F}}$) that is shared across all tasks, and task-specific decoders $\mathcal{D}_t$ (parameterized by $\theta_{\mathcal{D}_t}$) for each task. The model for the tth task is then defined as

$$\hat{\boldsymbol{y}}_{ti} = \mathcal{D}_t(\mathcal{F}(\boldsymbol{x}_{ti}; \theta_{\mathcal{F}}); \theta_{\mathcal{D}_t}) \,. \tag{2.7}$$

Given a fixed model architecture for all $\mathcal{D}_t$ and $\mathcal{F}$, the joint model is completely defined by the parameters $\theta = (\{\theta_{\mathcal{D}_t}\}_{t=1}^{T}, \theta_{\mathcal{F}})$. To maximize overall performance, the goal is to find optimal parameters θ^* such that

$$\theta^* = \operatorname*{argmin}_{\theta} \frac{1}{T} \sum_{t=1}^{T} \frac{1}{N_t} \sum_{i=1}^{N_t} \mathcal{L}(\boldsymbol{y}_{ti}, \hat{\boldsymbol{y}}_{ti}) \tag{2.8}$$

for a suitable sample-wise loss function $\mathcal{L}$, e.g., mean squared error or cross-entropy loss. More sophisticated deep MTL approaches can be characterized by how thy answer the design question: *How should learned parameters be shared across tasks?* The landscape of existing deep MTL approaches can be organized based on how they answer this question at the joint network architecture level (Figure 2.5):

Classical approaches. Many deep learning extensions remain close in nature to this approach, learning a shared representation at a high-level layer, followed by task-specific (i.e., unshared) decoders that extract labels for each task (Devin et al., 2016; Dong et al., 2015; Huang et al., 2013, 2015; Jaderberg et al., 2017b; Liu et al., 2015a; Ranjan et al., 2016; Wu et al., 2015; Zhang et al., 2014) (Figure 2.5a). This approach can be extended to task-specific input encoders (Devin et al., 2016; Luong et al., 2016), and the underlying single-task model may be adapted to ease task integration (Ranjan et al., 2016; Wu et al., 2015), but the core network is still shared in its entirety.

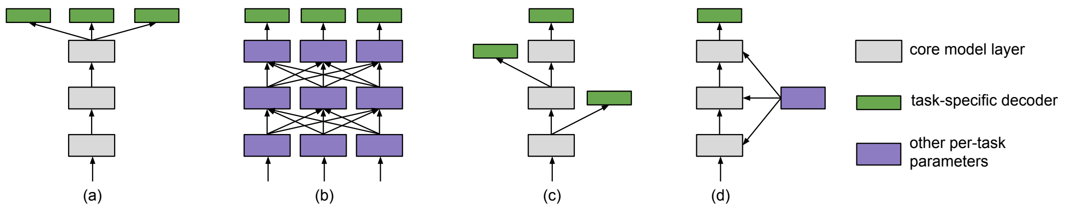


Figure 2.5: **Classes of existing deep multitask learning architectures.** (a) *Classical approaches* add a task-specific decoder to the output of the core single-task model for each task; (b) *Column-based approaches* include a network column for each task, and define a mechanism for sharing between columns; (c) *Supervision at custom depths* adds output decoders at depths based on a task hierarchy; (d) *Universal representations* adapts each layer with a small number of task-specific scaling parameters. Underlying each of these approaches is the assumption of parallel ordering of shared layers (Section 6.2): each one requires aligned sequences of feature extractors across tasks.

Column-based approaches. Column-based approaches (Jou & Chang, 2016; Misra et al., 2016; Rusu et al., 2016; Yang & Hospedales, 2017) assign each task its own layer of task-specific parameters at each shared depth (Figure 2.5b). They then define a mechanism for sharing parameters between tasks at each shared depth, e.g., by having a shared tensor factor across tasks (Yang & Hospedales, 2017), or allowing some form of communication between columns (Jou & Chang, 2016; Misra et al., 2016; Rusu et al., 2016). Observations of negative effects of sharing in column-based methods (Rusu et al., 2016) can be attributed to mismatches between the features required at the same depth between tasks that are too dissimilar.

Supervision at custom depths. There may be an intuitive hierarchy describing how a set of tasks are related. Several approaches integrate supervised feedback from each task at levels consistent with such a hierarchy (Hashimoto et al., 2017; Toshniwal et al., 2017; Zhang & Weiss, 2016) (Figure 2.5c). This method can be sensitive to the design of the hierarchy (Toshniwal et al., 2017), and to which tasks are included therein (Hashimoto et al., 2017). One approach learns a task-relationship hierarchy during training (Lu et al., 2017), for the problem of recognizing facial features from images, though learned parameters are still only shared across matching depths. Supervision at custom depths has also been extended to include explicit recurrence that reintegrates information from earlier predictions (Bilen & Vedaldi, 2016; Zamir et al., 2016). Although these recurrent methods still rely on pre-defined hierarchical relationships between tasks, they provide evidence of the potential of learning transformations that have a different function for different tasks at different depths, i.e., in this case, at different depths unrolled in time.

Universal representations. One approach shares all core model parameters except batch normalization scaling factors (Bilen & Vedaldi, 2017) (Figure 2.5d). When the number of classes is equal across tasks, even the output layers can be shared, and the small number of task-specific parameters enables strong performance to be maintained. This method was applied to a diverse array of vision tasks, demonstrating the power of a small number of scaling parameters in adapting layer functionality for different tasks. This observation helps to motivate the soft-merge method used in Chapters 6, 7, and 8.

Overall, independently of how structure is shared, i.e., independently of how each $\hat{\boldsymbol{y}}_{ti}$ is defined, in general the goal of training is to minimize the loss aggregated over all tasks, as in Equation 2.8. Thus, by requiring models to fit multiple real world datasets simultaneously, MTL is a promising approach to learning more realistic, and thus more generalizable, models.

2.3.2 Separate training of multiple models for STL

How to construct and train a deep neural network effectively is an open-ended design problem even in the case of a single task. A range of methods have been developed that aim at overcoming this problem by training multiple models separately for a single task. One class of methods searches for optimal fixed designs, e.g., by automatically optimizing learning hyperparameters (Bergstra et al., 2011; Snoek et al., 2012) or more open-ended network topologies (Miikkulainen et al., 2017; Real et al., 2017; Zoph & Le, 2017), or simply by manual trial-and-error.

Among these methods, evolutionary approaches are some of the most promising. For example, Covariance Matrix Adaptation Evolutionary Strategies (CMA-ES) has emerged as one of the most robust hyperparameter search methods (Loshchilov & Hutter, 2016). Evolutionary methods have also seen success in architecture search (Miikkulainen et al., 2017; Suganuma et al., 2017; Real et al., 2017, 2018), for which the ability of evolution to handle complex non-differentiable structures makes it particularly appropriate. For example, one of these approaches, CoDeepNEAT (Miikkulainen et al., 2017), extends the topology search methodology of neuron-level evolution, and is applied in

Chapter 7 to the even higher-level problem of evolution of deep multitask topologies. One advantage of this approach is that it is capable of discovering modular, repetitive structures reflecting those seen in successful and heavily hand-designed architectures Szegedy et al. (2015, 2016); He et al. (2016). An example of the kind of sets of modular topologies that such a method can discover is shown in Figure 2.4b.

When searching across multiple models for one that performs best on a single task, the multiple models synergize by providing complementary information about different areas of the search space, and, over time, the results of past models can be used to generate better models. Population-based training takes this one step further, by copying the weights of successful models to new models (Jaderberg et al., 2017a). This weight copying is similar to methods that transfer learned behavior across a sequence of pre-defined architectures (Hinton et al., 2015; Chen et al., 2016; Wei et al., 2016). The synergy of multiple models can also be exploited via ensembling, assuming the models are sufficiently powerful and diverse (Dietterich, 2000). Overall, the widespread success of the above methods have shown the value of training multiple models separately, both sequentially and in parallel.

2.3.3 Joint training of multiple models for STL

Some existing methods can be viewed as jointly training multiple models for a single task. For instance, to improve training of deep models, deep supervision includes loss layers at multiple depths (Lee et al., 2015). An example of deep supervision is shown in Figure 2.4c. As a by-product, this approach yields a distinct model for the task at each such depth, though only the deepest model is ever evaluated. That is, after training, the shallower models are discarded. The goal of this approach is not to discover multiple high-performing models for the task, but to use shallower models to improve the learning of the single deepest model.

As another example, dropout (Srivastava et al., 2014), and pseudo-ensembles more generally (Bachman et al., 2014), can be seen as implicitly training many relatively weak models that are combined during evaluation. Also, PathNet (Fernando et al., 2017) jointly trains multiple networks induced by various paths through a set of shared modules. However, the goal is not to improve single task performance, but discover structure that can be effectively reused by future tasks. Although the above methods jointly train multiple models for a single task, they do not perform joint training in the MTL sense.

Self-supervised training is an approach that is more aligned with the MTL setting in that there are multiple complementary losses whose gradient is combined when training (Doersch & Zisserman, 2017; Li et al., 2017; Jayaraman et al., 2018). In self-supervised learning, auxiliary targets are created directly from the input samples, without any additional hand-labeling, but taking advantage of existing external knowledge about the structure of the universe. Training towards these auxiliary targets jointly with the real target task can improve the generalization of the model by forcing it to capture more general structure in this universe. Since there is still only one underlying task of interest, self-supervised learning can still be categorized as training multiple models for a single task. Advances in self-supervised learning are orthogonal to the work in this dissertation: They focus on how to effectively construct the auxiliary tasks from external knowledge, whereas the work here focuses on how to discover generic components given a fixed set of tasks.

Ideally, the benefits of the methods in Sections 2.3.1 and 2.3.2 could be combined, yielding methods that train multiple models that share underlying parameters *and* sample complementary high-performing areas of the model space. The overarching angle taken in this dissertation is to approach this goal by collecting high-performing complementary modules that can be effectively applied in different ways. The framework in which these methods take form is introduced in the next chapter.

2.4 Conclusion

The methodologies and applications of neuroevolution and deep learning demonstrate that neural networks are a natural substrate for discovering useful modules. Multitask learning has been used to improve the generality of deep models, and structure search methods, including neuroevolution, have been used to optimize how they are designed. The functional perspectives from these three areas motivate the pseudo-task framework developed in the next chapter. The practical tools from these areas can then be used to develop systems within this framework.

Chapter 3

Framework: Functional Modules Solve Pseudo-tasks

This chapter describes a general framework for systems that use deep multitask learning to discover and apply collections of reusable functional modules. First, the notion of *pseudo-task* is defined. It extends the idea of decomposition into sub-problems from multitask learning to decomposition within the model for each particular task. The definition of pseudo-task is used to formalize what we mean by a *functional module*, i.e., a parameterizable function that can be applied to solve multiple pseudo-tasks. The value, behavior, and generality of a module can then be deduced based on the set of pseudo-tasks it is able to effectively solve. Given these definitions, there are a number of design choices that should be taken into account when developing a system that collects and applies functional modules. Finally, evaluation criteria are described for analyzing the behavior of such a system. This framework is used to characterize and analyze the concrete systems that have been developed and applied in the subsequent five chapters.

3.1 Pseudo-tasks

As in the standard multitask learning setup described in Chapter 2, consider a setup with T tasks $\{\{\boldsymbol{x}_{ti}, \boldsymbol{y}_{ti}\}_{i=1}^{N_t}\}_{t=1}^{T}$, with each task associated with a predictive model $\mathcal{M}_t$. These tasks will be referred to as the *true* or *underlying* tasks, with $\{\mathcal{M}_t\}_{t=1}^{T}$ referred to as the underlying models for these tasks. A pseudo-task for the tth task is defined by the task data for this task, along with a set of input nodes and a set of output nodes (all of which are constituent nodes in $\mathcal{M}_t$), and values for all parameters of $\mathcal{M}_t$ that are not contained in the subgraph whose boundaries are these inputs and outputs. The subgraph defined by the input and output nodes must be closed in the sense that every path from the interior of the subgraph to a node outside of the subgraph must pass through an output node before visiting any external node. This condition implies that the subgraph can be cleanly cut out of the model, i.e., the only new dangling nodes are the given inputs and outputs. Formally, suppose that at any time, the joint model contains L pseudo-tasks, and the ℓth pseudo-task has input and output nodes in $\mathcal{M}_t$. Then, the ℓth pseudo-task is denoted by a 5-tuple

$$(\mathcal{E}_\ell, \theta_{\mathcal{E}_\ell}, \mathcal{D}_\ell, \theta_{\mathcal{D}_\ell}, \{\boldsymbol{x}_{ti}, \boldsymbol{y}_{ti}\}_{i=1}^{N_t}), \tag{3.1}$$

where $\mathcal{E}_\ell$ is the encoder mapping each $\boldsymbol{x}_{ti}$ to the input of a function solving the pseudo-task, and $\mathcal{D}_\ell$ takes the output of that function (along with possibly $\boldsymbol{x}_{ti}$ again) to the predicted output $\hat{\boldsymbol{y}}_{ti}$. The parameters $\theta_{\mathcal{E}_\ell}$ and $\theta_{\mathcal{D}_\ell}$ characterize $\mathcal{E}_\ell$ and $\mathcal{D}_\ell$, respectively.

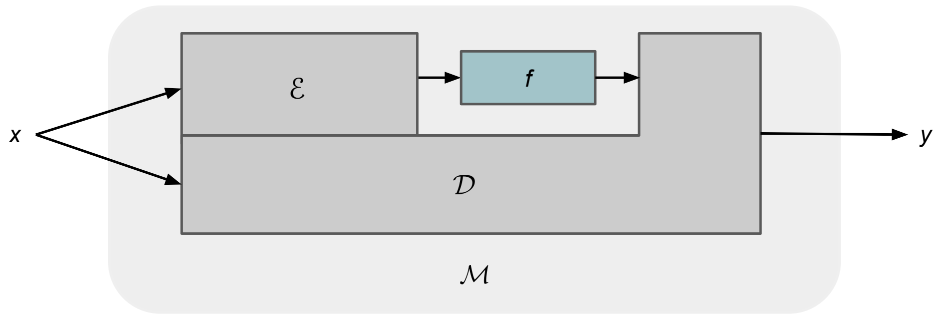


Figure 3.1: **Pseudo-task Decomposition**. Model $\mathcal{M}$, for the underlying task $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$, induces a pseudo-task solved by a function f. The block $\mathcal{E}$ is an encoder that provides input to f, and $\mathcal{D}$ is a decoder which uses the output of f to produce the final prediction. If f is effective for many [task, encoder, decoder] combinations, then it demonstrates generic functionality.

Given a pseudo-task, the model for the tth task is completed by specifying a fully-parameterized function f, with parameters θ_f, that connects the pseudo-task's inputs to its outputs. The goal for solving this pseudo-task is to find a function that minimizes the loss of the underlying task, while keeping the remaining parameters of the model fixed. This perspective enables us to decompose complex models into smaller problems, each of which can be considered separately. In particular, the completed model is given by

$$\hat{\boldsymbol{y}}_t = \mathcal{D}_\ell(f(\mathcal{E}_\ell(\boldsymbol{x}_t; \theta_{\mathcal{E}_\ell}); \theta_f), \boldsymbol{x}_t; \theta_{\mathcal{D}_\ell}). \tag{3.2}$$

This formulation is depicted in Figure 3.1. Note that $\mathcal{D}_\ell$ and $\mathcal{E}_\ell$ can share arbitrary subsets of their structure and parameters; the decomposition into $\mathcal{D}_\ell$, $\mathcal{E}_\ell$, and f is designed make clear where the input to f comes from, and where the output of f goes, i.e., how it is used to generate predictions.

The subproblems resulting from this decomposition are termed *pseudo-*

tasks because the goal of each is to minimize the loss of a true underlying task, but, since there is arbitrary nonlinear computation performed between the outputs of the pseudo-task and the true task loss, for a given set of inputs the pseudo-task may have many possible optimal outputs. In other words, it is called a *pseudo*-task because, although the loss is well defined, the targets are not unique. The prefix *pseudo* also conveys the transience of the pseudo-task; since the remaining parameters in $\theta_{\mathcal{M}}$ are usually trained simultaneously with the function solving the pseudo-task, each particular instance of a pseudo-task may only exist for a single learning iteration. Note that the definition of a pseudo-task given above generalizes the definition presented by Meyerson & Miikkulainen (2018b). Their definition took on a specific form coinciding with classical multitask learning. Aside from the simplicity of only splitting models into encoders followed by decoders, there is no inherent reason pseudo-tasks should be restricted to cases of model decomposition of that form.

3.2 Functional Modules

Given a pseudo-task, the model is completed by a fully-parameterized function with input-output description matching that of the pseudo-task. Say this function f is parameterized by θ_f, then f can be applied anywhere (i.e., to any pseudo-task) that has the same input-output dimensionality specification. The goal is to collect a set of modules $U = \{f_i\}_{i=1}^{|U|}$, such that any pseudo-task in a universe of domains and models can be solved satisfactorily by application of modules in U. Importantly, for f to be considered a functional *module* it must have the capacity to be applied in more than one location in the pseudo-task decomposition of a set of models.

The generality and generalizability of a module can then be characterized by the set of pseudo-tasks it effectively solves.

In the applications of this framework presented in this dissertation, the input and output sets for each pseudo-task each reside in a single neural network layer. This design allows us to neatly denote the input and output specification of a pseudo-task by a single tuple of dimensions, and only consider functional modules that map a single tensor of the input dimension to a single tensor of the output dimension.

Furthermore, if f and $\mathcal{M}$ are differentiable, then the completion of $\mathcal{M}$ is also differentiable, since differentiability is conserved under composition. This fact makes the functional module approach particularly well-suited to the machine learning substrate of neural networks, since, with the application of modules, $\mathcal{M}$ can still be trained end-to-end as if it were constructed in the usual non-modular way.

As an example, notice that if a complete network $\mathcal{M}$ is used to solve two distinct true task that have the same input and output shape, say CIFAR-10 and SVHN (Bilen & Vedaldi, 2017), then $\mathcal{M}$ itself is considered a module. Notice also that their is a base case, where a single free parameter is a module that is *universal*, in the sense that any model can be decomposed into pseudo-tasks, each defined by withholding a single parameter from the model.

Though seemingly trivial, even reparameterization at this atomic scale can yield interesting results, and structural insights, as demonstrated by Hashing Networks (Chen et al., 2015), where parameters in dense layers are randomly grouped into bins in which their value is shared. However, generally we are interested in finding useful sets of modules that are much more structured. For example, a functional module could be a neural network layer of a standard type that can be applied in multiple locations in a network, and shares parameters across all of its usages. In fact, this is the starting point for the approach taken in Chapter 6. This definition also captures other methods of parameter tying, such as those described in Section 2.2. For example, the kernel of a convolutional layer simultaneously solves many spatially related pseudo-tasks, each defined by a particular pixel location. Similarly, the kernels of a recurrent layer simultaneously solve many temporally related pseudo-tasks, each defined by a timestep. Finding a highly structured set of effective modules sheds light on how knowledge can be accumulated, shared and reapplied in deep models.

3.3 System Design Considerations

When implementing a system for discovering functional modules with application to a particular universe of tasks, there are some key design choices that need to be made. These choices can affect the applicability of the system, the scalability, the kinds of analysis that can be performed on the results, and the kinds of experimental effects that can be observed.

Learning. How are module weights learned? At the highest level, this question amounts to how learnable parameters are updated via the underlying machine learning method, e.g., via neuroevolution (Chapter 4) or gradient descent (Chapters 6, 7, 5 and 8). More specific to the issue of module reuse, there is the question of *when* module parameters are updated, i.e., they could be frozen at some point to ensure their encapsulated knowledge is never lost.

Applicability. When and where can modules be applied? It may be that modules are learned in one subset of pseudo-task locations and then are applied afterwards to another, but not visa versa. There is also a choice in whether different subsets in the set of collected modules have different input-output specifications. If there are such disjoint subsets, this would imply that certain subsets of pseudo-task locations would only accept the corresponding subsets of modules.

Discovery. How are new modules created? It can be helpful to ground modules by associating them initially with existing structure in a model. Alternatively, it is possible for modules to appear out of the blue, or augment the topology of an existing model. Similarly, once a module is initialized, it is possible for it to change its structure over the lifetime of the system, i.e., grow to better fit its particular niche. For modules that end up not being useful, it can be practical to include a method of module deletion, to remove modules that would otherwise simply be a performance drag.

Integration. How are modules integrated into a model when they are applied in new locations? This question is closely related to the learning question, but with a particular focus on how to ensure that modules that are in fact appropriate for a particular location are actually accepted and effectively utilized by the system. For example, if a new module is added at a location, how can we ensure that the model does not go into shock, e.g., by rejecting the module or having its performance tank?

Form. What is the functional form of the modules? This is one of the most important questions, since it has clear consequences for the type of structure that can be captured by the set of modules. The form of the modules can also be induced by how the underlying models are decomposed into pseudo-tasks. For example, the granularity of this decomposition can inform the kinds of algorithms that are used to optimize how modules are repurposed. In Chapters 4 and 6, modules take the form of neural network layers; in Chapters 7 and 5 they take the form of arbitrary subnetworks; and in Chapter 8 they take the form of blocks of parameters that perform linear transformations within layers.

Mapping. How are applicable modules mapped to locations where they will succeed? It is possible to have all modules in consideration available at all times at all locations (Chapters 6 and 5), to have only a subset of modules be available for each training run (Chapter 4), or to have modules be dynamically mapped to locations by a search procedure auxiliary to the parameter-update procedure (Chapters 7 and 8).

Of course, the above design considerations are closely intertwined, and will inform each other based on the goals of the system.

3.4 Evaluation Criteria

The design considerations given in the previous section can give way to a broad array of systems for discovering functional modules through deep multitask learning. When evaluating the performance of such a system, there may be some system-specific metrics that arise from particular choices. That said, the following evaluation criteria are often relevant to understanding the behavior of the system, and could be considered across all such systems.

Multitask Performance This is the most natural and universal performance metric for evaluating multitask learning systems. As described in Chapter 2, the multitask performance is defined as an aggregation (usually mean) of performance metrics across tasks.

Single-task Performance Sometimes the aggregation of multitask performance can be difficult to apply clearly, for example, in the case where the performance metrics for different tasks are on different and difficult to compare scales. In this case, the performance on each single task can be considered. If one multitask method performs better on some tasks than others, looking at the single-task performance can shed light on when the method works well, and suggest ways to improve it to work on tasks where it does not work as well.

Complexity of Solution Since the goal is to discover knowledge that can be reused for different purposes, a parsimonious solution is preferred. The simplest measure of complexity is the total number of parameters in the model. Another measure is the number of modules that are collected. A third is what fraction of the underlying models are able to be replaced by reusable modules. In general, to make experiments comparable to baseline experiments with underlying models, factors affecting complexity should be kept constant across the different setups.

Structural Insights Structural insights is probably the most important criterion for the value of this kind of system. If the behavior of the system suggests structural insights about how knowledge and information can be shared across different problems, these insights can help identify which directions and aspects of the framework are most interesting and promising to focus on. Incidentally, the types of structural insights that can be observed depend heavily on the design of the system.

3.5 Overview of Implemented Systems

The next five chapters present particular implementations of systems that discover functional modules. Each system takes a distinct approach to multitask modularity, and progressively increase the generality of learned modules. Together they constitute a comprehensive exploration of the space of such systems.

Chapter 4 investigates the inherent generality of neural network modules. To this end, it takes an optimistic approach, in which a layer of a network learned for one task is frozen and can be used as a subcomponent in a network for any other task. The new task is often able to exploit knowledge from this frozen layer, however, the generality of the modules are limited by the fact that the internals of reusable components are only learned in a single setting.

Chapter 5 focuses on increasing the generality of a single module by increasing the number of pseudo-tasks. This effect is investigated in the classical case where all models share a single module and differ only in their output decoders. The results show that additional pseudo-tasks can improve generalization even in this foundational case.

Chapter 6 takes this idea of training modules for more pseudo-tasks to the extreme. In this system, each layer is trained simultaneously across different depths in different tasks, thus forcing their functionality to be general. Each task model is constructed by applying a distinct mixture of these general layers at each depth. Although general modules are learned and assembled in different ways for different tasks, the scalability of the system is limited by the computational cost of applying each module at each location, and the generality of the modules is limited by their single-layer form.

Chapter 7 addresses these limitations by using a coevolutionary approach to incrementally complexify how modules are assembled in topologically distinct ways for each task. This approach is also extended to the case where each module may consist not only of a single layer but of an arbitrary topology, also coevolved. These approaches rely on a novel coevolutionary mechanism that jointly trains multiple models for each task.

Chapter 8 introduces a final system that enables sharing of modules across qualitatively diverse cross-modal architectures and tasks. The key idea is to decompose a set of models into a set of equal-sized linear subproblems, each of which is solvable by modules of equivalent form. The practical implementation of the approach draws on the ideas and mechanisms developed in the four preceding chapters. In particular, the system can have orders of magnitude more modules than the previous systems, so to address the scalability issues that arose in Chapter 6, this system uses a coevolutionary mechanism like that in 7, while using the mixture learning of 6 as a surrogate fitness function. As a capstone system, the approach constitutes a significant step towards the grand promise of multitask learning: All real world problems can be solved by applying a compact set of universal modules.

Chapter 4

General Reuse of Static Modules

This chapter[1] investigates how general trained modules are inherently, i.e., when trained to solve a single pseudo-task. To that end, a general approach to knowledge transfer in which an agent controlled by a neural network adapts how it reuses existing networks as it learns in a new domain. Networks trained in a new domain can improve their performance by routing activation selectively through previously learned neural structure, regardless of how or for what it was learned. Thus, modules learned for one pseudo-task are repurposed to solve pseudo-tasks in other domains. A neuroevolution implementation of this approach is presented with application to high-dimensional sequential decision-making domains. This approach is more general than previous approaches to neural transfer for reinforcement learning. It is domain-agnostic and requires no prior assumptions about the nature of task relatedness or mappings. The method is analyzed in a stochastic version of the Arcade Learning Environment, demonstrating that it improves performance in some of the more complex Atari 2600 games, and that the success of transfer can be predicted based on a

[1]The content of this chapter was previously presented at AAAI (Braylan, Hollenbeck, Meyerson, and Miikkulainen, 2016). Alex Braylan and Mark Hollenbeck worked on experimental design, implementation and analysis; while Risto Miikkulainen provided guidance and feedback through discussions.

high-level characterization of game dynamics. The conclusion is that neural network modules already have potential for generality, which sets the stage for the methods in subsequent chapters that seek to expand this generality.

4.1 Introduction

The ability to apply available previously learned knowledge to new tasks is a hallmark of general intelligence. *Transfer learning* is the process of reusing knowledge from previously learned *source* tasks to bootstrap learning of *target* tasks. Transfer learning is thus a special case of multitask learning, in which there is a fixed temporal order in which tasks must be learned. In long-range sequential control domains, such as robotics and video game-playing, transfer is particularly important, because previous experience can help agents explore new environments efficiently (Taylor & Stone, 2009; Konidaris et al., 2012). Knowledge acquired during previous tasks also contains information about an agent's domain-independent decision making and learning dynamics, and thus can be useful even if the domains seem unrelated.

Existing approaches to transfer learning in such domains have demonstrated successful transfer of varying kinds of knowledge, but they make two fundamental assumptions that restrict how generally applicable they are: (1) some sort of a priori human-defined understanding of how tasks are related, and (2) separability of knowledge extraction and target learning. The first assumption limits how well the approach can be applied by restricting its use only to cases where the agent has been provided with this additional relational knowledge, or, if it can be learned (Talvitie & Singh, 2007; Taylor et al., 2008; Ammar et al., 2015b), cases where task mappings are useful. The second assumption implies that it is known what knowledge will be useful and how it should be incorporated *before* learning on the target task begins, preventing the agent from adapting the way it uses source knowledge as it gains information about the target domain.

General ReUse of Static Modules (GRUSM) is proposed in this chapter as a general neural network approach to transfer learning that avoids both of these assumptions. GRUSM augments the learning process to allow learning networks to route through existing neural modules (source networks) selectively as they simultaneously develop new structure for the target task. Unlike previous work, which has dealt with mapping task variables between source and target, GRUSM is domain-independent, in that no knowledge about the structure of the source domain or even knowledge about where the network came from is required for it to be reused. Instead of using mappings between task-spaces to facilitate transfer, it searches directly for mappings in the solution space, that is, connections between existing source networks and the target network. This approach is motivated by studies that have shown in both naturally occurring complex networks (Milo et al., 2002) and in artificial neural networks (Swarup & Ray, 2006) that certain network structures repeat and can be useful across domains, without any context for how exactly this structure should be used. This work is further motivated by the idea that neural resources in the human brain are reused for countless purposes in varying complex ways

(Anderson, 2010).

In this chapter, an implementation of GRUSM based on the Enforced Subpopulations (ESP) neuroevolution framework (Gomez & Miikkulainen, 1997, 1999) is presented. The approach is validated on the stochastic Atari 2600 general game playing platform, finding that GRUSM-ESP improves learning for more complex target games, and that these improvements may be predicted based on domain complexity features. This result demonstrates that even without traditional transfer learning assumptions, successful knowledge transfer via general reuse of existing neural modules is possible and useful for long-range sequential control tasks. In principle, this approach scales naturally to transfer from an arbitrary number of source tasks, which suggests that in the future it may be possible to build GRUSM agents that accumulate and reuse knowledge throughout their lifetimes across a variety of diverse domains.

The remainder of this chapter is organized as follows: The next section provides background on transfer learning and related work. The subsequent section describes the GRUSM approach in detail. Then, results from experiments are analyzed, and the implications of these results and motivations for future work are discussed.

4.2 Related Work in Transfer Learning

Transfer learning encompasses machine learning techniques that involve reusing existing *source* knowledge in a different *target* task or domain. A *domain* is an environment in which learning takes place, characterized by the input and output space; a *task* is a particular function from input to output to be learned (Pan & Yang, 2010). In sequential-decision domains, a task is characterized by the values of sensory-action sequences corresponding to the pursuit of a given goal. A taxonomy of types of knowledge that may be transferred was also enumerated by Pan and Yang. Because the GRUSM approach reuses the structure of existing neural networks, it falls under *feature representation transfer*.

4.2.1 Transfer Learning for Reinforcement Learning

Transfer learning for sequential decision-making domains has been studied extensively within the *reinforcement learning* (RL) paradigm (Taylor & Stone, 2009). Reinforcement learning domains are often formulated as Markov decision processes (MDPs) in which the state space comprises all possible observations, and the probability of an observation depends only on the previous observation and action taken by a learning agent. However, many real world RL domains are non-Markovian, including many Atari 2600 games, for example, the velocity of a moving object cannot be determined by looking at a single frame.

The Atari 2600 platform also supports a wide variety of games. Existing RL approaches to transfer differ on the types of differences allowed between source and target task. Some approaches that are general with respect to the kind of knowledge that can be transferred are restricted in that they require a consistent *agent-space* (Konidaris et al., 2012), or an a priori specification of inter-task *mappings* defining relationships between source and target state and action variables (Brys et al., 2015). Existing approaches to transfer learning that encode policies as neural networks require such a specification (Taylor et al., 2007; Verbancsics & Stanley, 2010). On the other hand, existing modular neuroevolution approaches that are more general with respect to connectivity (Reisinger et al., 2004; Khare et al., 2005) have not been applied to cross-domain transfer.

Some of the most general existing approaches to transfer for RL automatically learn task mappings, so they need not be provided beforehand. These approaches are general enough to apply to any reinforcement learning domains, but initial approaches (Taylor et al., 2008; Talvitie & Singh, 2007) were intractable for high dimensional state and action spaces due to combinatorial blowup in the number of possible mappings. However, recent approaches in policy gradient RL (Ammar et al., 2015a,b) can both tractably learn mappings and be applied across diverse domains. These approaches have been successful in continuous control domains, but it is unclear how they would scale to domains with many discretely-valued features such as Atari. Also, the above approaches assume MDP environments, whereas GRUSM can use recurrent neural networks to extend to POMDPs.

4.2.2 General Neural Structure Transfer

There are existing algorithms similar to GRUSM in that they make it possible to reuse existing neural structure. They can apply to a wide range of domains and tasks in that they automatically select source knowledge and avoid inter-task mappings. For example, Shultz & Rivest (2001) developed a technique to build increasingly complex networks by inserting source networks chosen by how much they reduce error. This technique is only applicable to supervised learning, because the source selection depends heavily on an immediate error calculation. Also, connectivity between source and target networks is limited to the input and output layer of the source. As another example, Swarup & Ray (2006) introduced an approach that creates sparse networks out of primitives, or commonly used sub-networks, mined from a library of source networks. This subgraph mining approach depends on a computationally expensive graph mining algorithm, and tends to favor exploitation over innovation and small primitives rather than larger networks as sources.

The GRUSM approach is more general in that it can be applied to unsupervised and reinforcement learning tasks, makes few a priori assumptions about what kind of sources and mappings should work best, and is able to develop memory via recurrent connections. Although an evolutionary approach is developed in this chapter, GRUSM should be extensible to any neural network-based learning algorithm.

4.3 Approach

This section introduces the general idea behind GRUSM, provides an overview of the ESP neuroevolution framework, and describe the particular implementation: GRUSM-ESP.

4.3.1 General ReUse of Static Modules (GRUSM)

The underlying idea is that an agent learning a neural network for a target task can reuse knowledge selectively from existing neural modules (source networks) while simultaneously developing new structure unique to a target task. This approach attempts to balance reuse and innovation in an integrated architecture. Both source networks and new hidden nodes are termed *recruits*. Recruits are added to the target network during the learning process. Recruits are incorporated adaptively into the target network as it learns connection parameters from the target to the recruit and from the recruit to the target. All internal structure of source networks is *frozen* to allow learning of connection parameters to remain consistent across recruits. This mechanism forces the target network to transfer learned knowledge, rather than simply overwrite it. Connections to and from source networks can, in the most general case, connect to any nodes in the source and target, minimizing assumptions about what knowledge will be useful.

A GRUSM network is a 3-tuple $\mathcal{G} = (\mathcal{M}, S, T)$ where $\mathcal{M}$ is a traditional neural network (feedforward or recurrent) containing the new nodes and connections unique to the target task, with input and output nodes corresponding to inputs and outputs defined by the target domain; S is a (possibly empty) set of pointers to recruited source networks $\mathcal{S}_1, ..., \mathcal{S}_k$; and T is a set of weighted *transfer connections* between nodes in $\mathcal{M}$ and nodes in source networks, that is, for any connection $((u, v), w) \in T$, $(u \in \mathcal{M} \wedge v \in \mathcal{S}_i) \vee (u \in \mathcal{S}_i \wedge v \in \mathcal{M})$ for some $0 \leq i \leq k$. This construction strictly extends traditional neural networks so that each $\mathcal{S}_i$ can be a traditional neural network or a GRUSM network of its own. When $\mathcal{G}$ is evaluated, only the network induced by directed paths from inputs of $\mathcal{M}$ to outputs of $\mathcal{M}$, including those which pass through some $\mathcal{S}_i$ via connections in T is evaluated, i.e., each $\mathcal{S}_i$ solves a pseudo-task defined by these paths. During each evaluation of $\mathcal{G}$, all recruited source network inputs are fixed at 0, since the agent is concerned only with performing the current target task. The parameters to be learned are the usual parameters of $\mathcal{M}$, along with the contents of S and T. The internal parameters of each $\mathcal{S}_i$ are *frozen* in that they cannot be rewritten through $\mathcal{G}$.

The motivation for this architecture is that if the solution to a source task contains *any* information relevant to solving a target task, then the neural network constructed for the source task will contain *some* structure (subnetwork or module) that will be useful for a target network. This has been previously observed in naturally occurring complex networks (Milo et al., 2002), as well as cross-domain artificial neural networks (Swarup & Ray, 2006). Unlike the subgraph-mining approach to neural structure transfer (Swarup & Ray, 2006), this general formalism makes no assumptions as to what subnetworks actually will be useful. One interpretation is that a lifelong learning agent maintains a system of interconnected neural modules that it can potentially reuse at any time for a new task. Even if existing modules are unlabeled, they may still be useful, due to the fact that they contain knowledge of how the agent can successfully learn. Furthermore, advances in reservoir computing (Lukoševičius & Jaeger, 2009) have demonstrated the power of using large amounts of frozen neural structure to facilitate learning of complex and chaotic tasks.

The above formalism is general enough to allow for an arbitrary number of source networks and arbitrary connectivity between source and target. In this chapter, to validate the approach and simplify analysis, at most one source network is used at a time and only connections from target input to source hidden layer and source output layer to target output are permitted. By not allowing target input to connect to source input, this restriction avoids high-dimensional transformations between domain-specific sensor substrates, and more intuitively captures the domain-agnostic goals of the approach, differentiating the approach from previous methods that have used direct mappings between sensor spaces. This restriction is sufficient to show that the implementation can reuse hidden source features successfully, and it is possible to analyze the cases in which transfer is most useful. Future refinements are discussed in the Discussion and Future Work section. The current implementation, described below, is a neuroevolution approach based on ESP.

4.3.2 Enforced Subpopulations (ESP)

Enforced Sub-Populations (ESP; Gomez & Miikkulainen 1997; 1999) is a neuroevolution technique in which different components of a neural network are evolved in separate *subpopulations* rather than evolving the whole network in a single population, or evolving a single population of neurons as in SANE (see Section 2.1) (Moriarty & Miikkulainen, 1996). ESP has been shown to perform well in a variety of reinforcement learning domains, and has shown promise in extending to POMDP environments, in which use of recurrent connections for memory is critical (Gomez & Miikkulainen, 1999; Gomez & Schmidhuber, 2005; Schmidhuber et al., 2007). In traditional ESP, there is a single hidden layer, each neuron of which is evolved in its own subpopulation. Recombination occurs only between members of the same subpopulation, and mutants in a subpopulation derive only from members of that subpopulation. The genome of each individual in a subpopulation is a vector of weights corresponding to the weights of connections to and from that neuron, including node bias. In each generation, networks to be evaluated are randomly constructed by inserting one neuron from each subpopulation. Each individual that participated in the network receives the fitness achieved by that network.

When fitness converges, i.e., does not improve over several consecutive generations, ESP makes use of δ-coding, also known as *burst phases*. In initial burst phases each subpopulation is repopulated by mutations of the single best neuron ever occurring in that subpopulation, so that it reverts to searching a δ-neighborhood around the best solution found so far. If a second consecutive burst phase is reached, i.e., no improvements were made since the previous burst phase, a new neuron with a new subpopulation may be added (Gomez, 2003).

4.3.3 GRUSM-ESP

The idea of enforced sub-populations is extended to transfer learning via GRUSM networks. For each reused source network $\mathcal{S}_i$, the transfer connections in T between $\mathcal{S}_i$ and $\mathcal{M}$ evolve in a distinct subpopulation. At the same time new hidden nodes can be added to $\mathcal{M}$; they evolve within their own subpopulations in the manner of standard ESP. In this way, the integrated evolutionary process simultaneously searches the space for how to reuse each potential source network and how to innovate with each new node. The GRUSM-ESP architecture (Figure 4.1) is composed of the following elements: (1) A pool of potential source networks. In the experiments in this chapter, each target network reuses at most one source at a time; (2) *Transfer genomes* encoding the weights of cross-network connections between source and target. Each potential source network in the pool has its own subpopulation for evolving transfer genomes between it and the target network. Each connection in T is contained in some transfer genome. In our experiments, the transfer connections included are those such that the target's inputs are fully connected to the source's hidden layer, and the source's outputs are fully connected into the target's outputs; (3) A burst mechanism that determines when innovation is necessary based on a recent history of performance improvement. New hidden recruits (source networks when available, and single nodes otherwise) added during the burst phase evolve within their own subpopulations as in standard ESP.

All hidden and output neurons use a hyperbolic tangent activation function. Base networks include a single hidden layer, and include recurrent self loops on hidden nodes; they are otherwise feedforward. When a network contains a layer from a previous task as a submodule, it has more than one

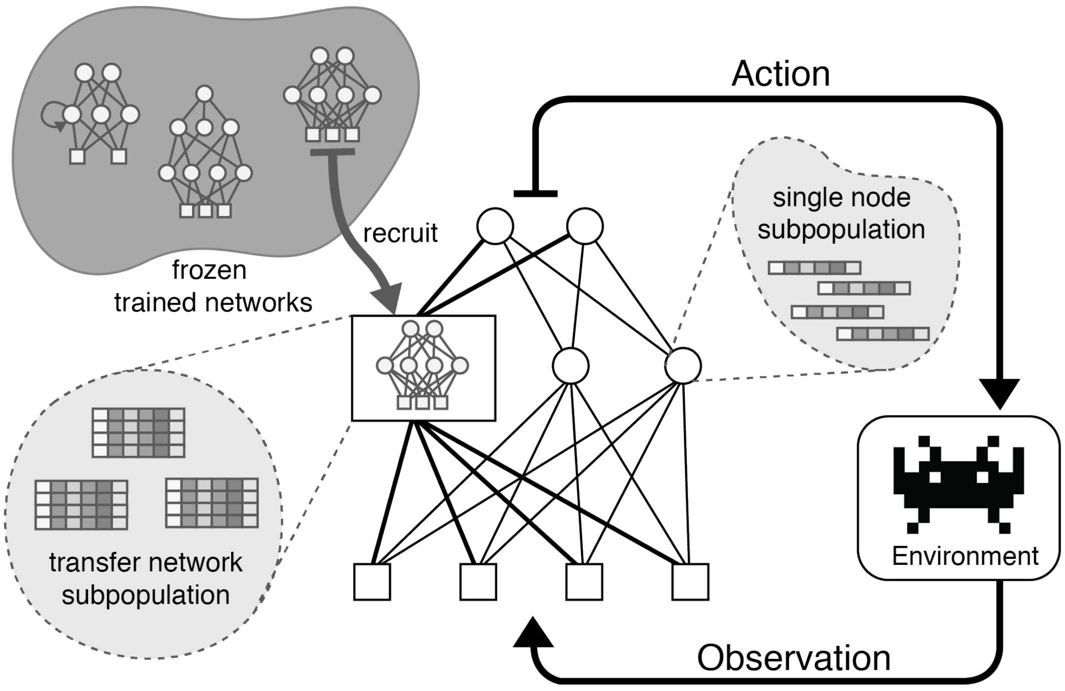


Figure 4.1: The GRUSM-ESP architecture, showing the balance between reused and new structure. In this example, the target network has three recruits: one source network, and two single nodes. Each bold edge between target network nodes and source network recruit indicate connections to multiple source nodes. The genome in each subpopulation encodes weight information for the connections from and to the corresponding recruit.

hidden depth, and can be considered a deep model. The details of the genetic algorithm in our implementation used to evolve each subpopulation mirror those described by Gomez (2003). This algorithm has been shown to work well within the ESP framework, though any suitable evolutionary algorithm could potentially be substituted in its place.

4.4 Experiments

GRUSM-ESP was evaluated in a stochastic version of the Atari 2600 general video game-playing platform using the Arcade Learning Environment simulator (ALE; Bellemare et al. 2013). Atari 2600 is currently a very popular platform, because it challenges modern approaches, contains non-markovian games, and entertained a generation of human video game players, who would regularly reuse knowledge gained from previous games when playing new games. To make the simulator more closely resemble the human game-playing experience, the ϵ-repeat action approach as suggested by Hausknecht & Stone (2015) is used in this chapter to make the environment stochastic; in this manner, like human players, the algorithm cannot as easily find loopholes in the deterministic nature of the simulator. The recommended $\epsilon = 0.25$[2] is used. Note that the vast majority of previously published Atari 2600 results are in the deterministic setting; we are unaware of any existing scores that have been published in the ϵ-repeat setting.

Agents were trained to play eight games: Pong, Breakout, Asterix, Bowling, Freeway, Boxing, Space Invaders, and Seaquest. Neuroevolution techniques are competitive in the Atari 2600 platform (Hausknecht et al., 2013), and ESP in particular has yielded state-of-the-art performance for several games (Braylan et al., 2015). Three GRUSM-ESP conditions are evaluated: `scratch`, `transfer`, and `random`. In the `scratch` condition, networks are trained from

[2]https://github.com/mgbellemare/Arcade-Learning-Environment/tree/dev

scratch on a game using standard ESP (GRUSM-ESP with $S = \emptyset$). In the `transfer` condition, each scratch network is reused as a source network in training new GRUSM networks for different target games. In the `random` control condition, random networks are initialized and reused as source networks. Such networks contain on average the same number of parameters as fully-trained scratch networks.

Each run lasted 200 generations with 100 evaluations per generation. Since the environment is stochastic, each evaluation consists of five independent trials of individual i playing game g, and the resulting score $s(i, g)$ is the average of the scores across these trials. The score of an evolutionary run at a given generation is the highest $s(i, g)$ achieved by an individual by that generation. A total of 333 runs were run split across all possible setups. Evolutionary parameters were selected based on their success with standard ESP.

To interface with ALE, the output layer of each network consists of a 3x3 substrate representing the nine directional movements of the Atari joystick in addition to a single node representing the Fire button. The input layer consisted of a series of object representations manually generated as previously described by Hausknecht et al. (2013). The location of each object on the screen was represented in an 8×10 input substrate corresponding to the object's class. The numbers of object classes varied between one and four. Although object representations were used in these experiments, pixel-level vision could also be learned from scratch below the neuroevolution process, e.g., via convolutional networks as was done by Koutník et al. (2014).

Domain Characterization Understanding *when* transfer will be useful is important for any transfer learning approach. In many cases, attempting transfer can impede learning, leading to *negative transfer*, when an approach is not able to successfully adapt knowledge from the source to the target domain. Negative transfer is a serious concern for many practitioners (Taylor & Stone, 2009; Pan & Yang, 2010). To help understand when GRUSM-ESP should be applied, it is useful to consider the diverse array of games within a unified descriptive framework. Biological neural reuse is generally thought to be most useful in transferring knowledge from simple behaviors to more complex, and the vast majority of previous computational approaches do exactly that. Thus, the characterization of games in this chapter is grounded by a sense of relative complexity.

Each game can be characterized by generic binary features that determine what successful game play requires: (1) horizontal movement (joystick left/right), (2) vertical movement (joystick up/down), (3) shooting (fire button); (4) delayed rewards; and (5) long-term planning. Intuitively, more complex games will include more of these features. A partial ordering of games by complexity defined by these features is shown in Figure 4.2. The assignment of features (1), (2) and (3) is completely defined based on game interface (Bellemare et al., 2013). Freeway and Seaquest are said to have *delayed rewards* because a high score can only be achieved by long sequences of rewardless behavior. Only Space Invaders and Seaquest were deemed to require long-term

	v	h	s	d	p
pong	■				
breakout		■			
asterix	■	■			
bowling	■		■		
freeway	■			■	
boxing	■	■	■		
s. invaders		■	■		■
seaquest	■	■	■	■	■

v = vertical movement
h = horizontal movement
s = shooting
d = delayed rewards
p = long-term planning

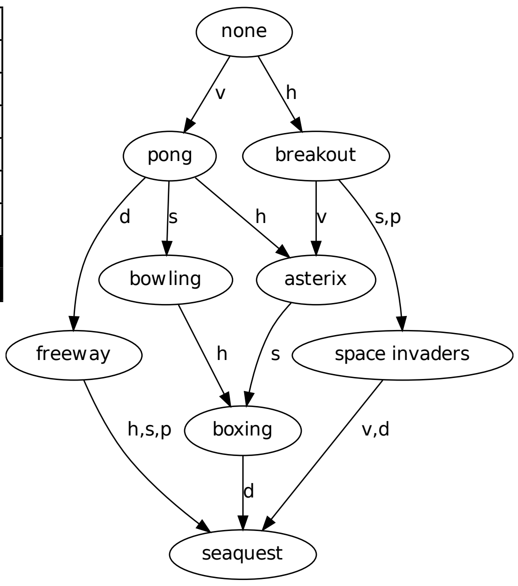


Figure 4.2: (left) Feature representation for each game, and (right) games partially-ordered by feature inclusion: every path from *none* to g contains along its edges each complexity feature of g exactly once, showing how games are related across the feature space. The existence of such a hierarchy motivates the use of atari for transfer.

planning (Mnih et al., 2015a), since the long-range dynamics of these games penalize reflexive strategies, and as such, agents in these games can perform well with a low frequency decision-making (Braylan et al., 2015). In addition to being intuitive, these features are validated below based on how well they characterize games by complexity and how well they predict successful transfer.

Analysis Methods There are many possible metrics for evaluating success of transfer, depending on what kind of transfer is desired or expected. Learning curves are irregular across different games, as illustrated in Figure 4.3, which makes it difficult to choose a single metric that makes sense across all source-target pairs. Thus, the analysis is focused on a broad notion of *transfer effectiveness* (TE), which aggregates metrics such as jumpstart and max overall score, with a weighted approximation of area under the curve (Taylor & Stone, 2009). *Success* of a setup is defined as the sum of the average score of that setup at a series of non-uniformly-spaced generations: $[1, 10, 50, 100, 200]$. This series favors early performance over later performance, as in general, in the long run, training from transfer and scratch should converge, as scratch eventually relearns everything that was effectively transferred. Then, the TE of a setup is its success minus the success of the control on the target game, the difference normalized by the size of the range of max scores achieved across all runs for that game, in order to draw comparison across games.

The first hypothesis is that `transfer` would outperform `scratch` in some setups, and that those setups could be predicted (i.e., they are not coincidental). However, any outperformance of `transfer` over `scratch` could be due to a larger number of network parameters. Therefore, as a second hypothesis, `random` setups were used as a control for the number of parameters, to test how `transfer` could predictably outperform `random`. We postulated and tested several useful indicators for predicting the outperformance of transfer, i.e., TE: (1) *feature similarity*: count of features that are 1 for both source and target); (2) *source feature complexity*: feature count of source game; (3) *target feature complexity*: feature count of target game; (4) *source training complexity*: source game average time to threshold; (5) *target training complexity*: target

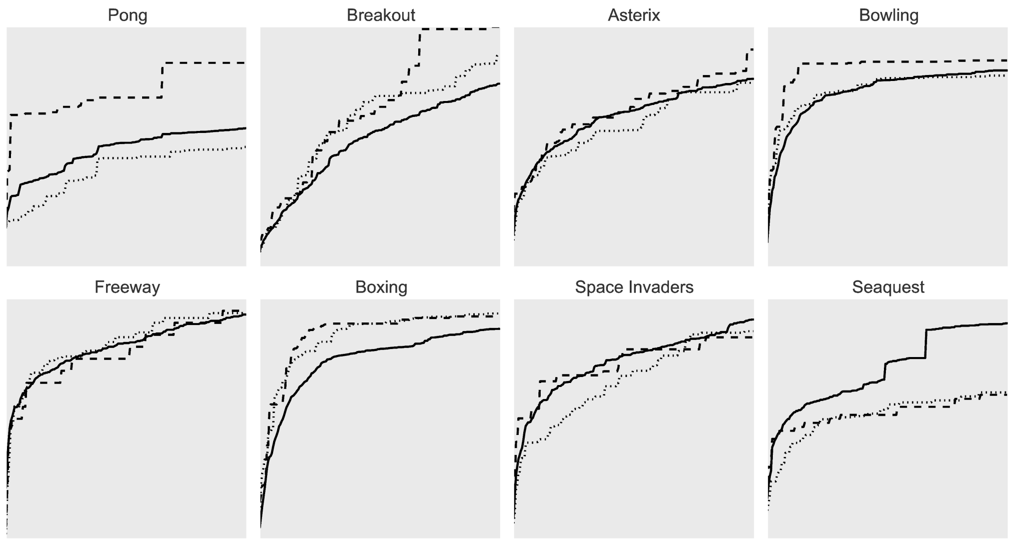


Figure 4.3: Raw mean score learning curves by generation for each target game aggregated over `transfer` (solid), `random` (dashed), and `scratch` (dotted) setups. The diversity of these learning curves shows the difficulty in comparing performance across games.

game average time to threshold, where the threshold for each game is the minimum max score achieved across all `scratch` runs for that game, and time to threshold is the average number of generations to reach this threshold.

To predict TE, a linear regression model was trained in a leave-one-out cross-validation analysis. For each possible source-target pair (s, t), the model was trained on all pairs $(s', t' \neq t)$ with TE as the dependent variable and the five indicators as the independent variables. Subsequently, the trained model was used to predict the TE of (s, t). Correlation between the actual and predicted TE across all test pairs was used to gauge the predictability of TE. This experiment was conducted identically for both `transfer` versus `scratch` and `transfer` versus `random` conditions.

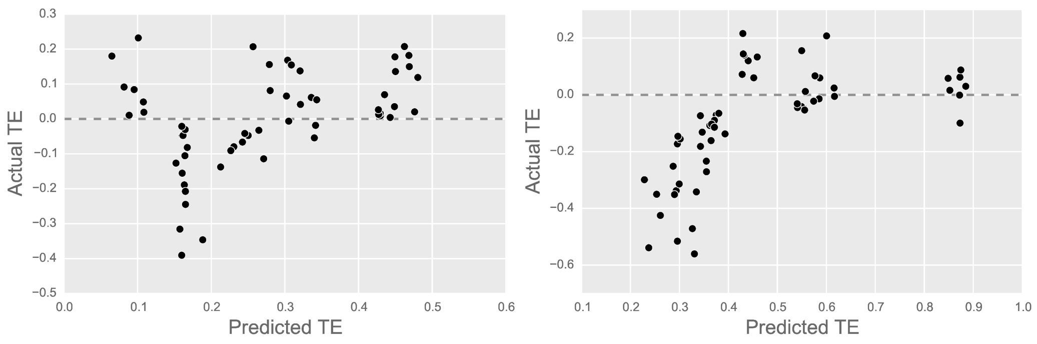
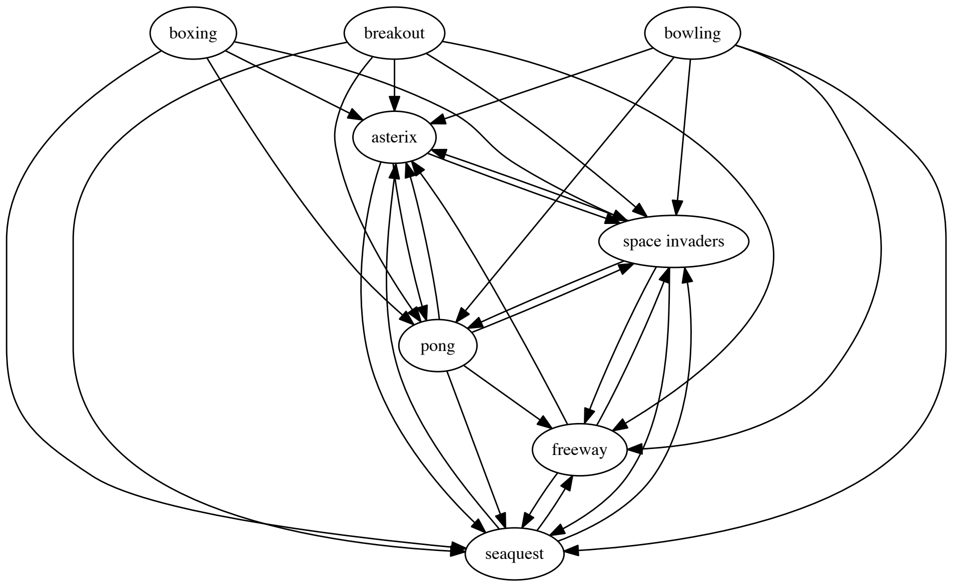


Figure 4.4: Predicted vs. actual transfer effectiveness with respect to `scratch` (left) and `random` (right). Both predictors have a significant correlation between predicated and actual transfer effectiveness.

Results For both hypotheses, the indicator-based model proved to be a statistically significant predictor of transfer effectiveness in the test data: correlation $R = 0.40$ and p-value < 0.0025 for `transfer` versus `scratch`; correlation $R = 0.53$ and p-value $< 10^{-7}$ for `transfer` versus `random` (Figure 4.4). The strongest indicators for `transfer` versus `scratch` were target feature complexity and target training complexity, and for `transfer` versus `random` the strongest indicator was target feature complexity.

The fact that more complex games are more successful targets should not be surprising. As noted before, in most transfer learning scenarios, only simple-to-complex transfer is considered. The ability to predict when GRUSM-ESP will work is an important tool when applying this method to larger problems, and it is encouraging that the predictive indicator coincides with the 'common sense' expectations of transfer effectiveness in the current experiments. TE for all source-target pairs is visualized in Figure 4.5. Also, although it is difficult to compare to the deterministic Atari 2600 domain, Table 4.1 provides



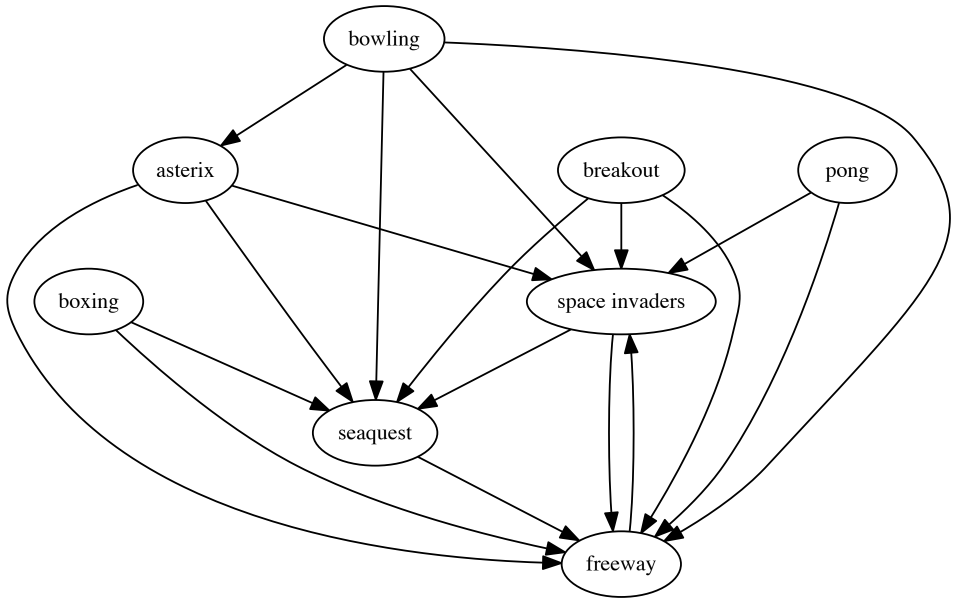


Figure 4.5: Transferability graphs over all pairs of tasks with respect to `scratch` (top) and `random` (bottom) illustrating the target-centric clustering of successful source-target pairs. Each graph includes a directed edge from g_1 to $g_2 \iff$ the TE (see Analysis) for g_2 reusing g_1 is positive.

Game	`scratch`	`random`	`transfer`	human	DQN
Pong	0.0	21.0	10.0_{ast}	9.3	18.9
Breakout	31.0	35.0	30.3_{box}	31.8	401.2
Asterix	2800	3216.7	3355_{bow}	8503	6012
Bowling	249.3	265.0	265.0_{fr}	154.8	42.4
Freeway	31.4	31.5	32.2_{brk}	29.6	30.3
Boxing	93.9	92.7	95.0_{sea}	4.3	71.8
Space Invaders	1438.0	1407.5	1655.0_{po}	1652	1976
Seaquest	466.0	460.0	975.0_{sp}	20182	5286

Table 4.1: For each game, average scores for `scratch`, `random`, and `transfer` from best source (subscripted). Interestingly, the best source for each target is unique. We also show human and DQN scores (Mnih et al., 2015a). Note: DQN uses deterministic ALE, so the most apt external comparison here may be to humans, who cannot deterministically optimize trajectories at the frame level.

a comparison of GRUSM-ESP to recent results in that domain for context (Mnih et al., 2015a).

4.5 Discussion and Future Work

The results show that GRUSM-ESP (an evolutionary algorithm for general transfer of neural network structure) can improve learning in Atari game playing by reusing previously developed knowledge. They also make it possible to characterize the conditions under which transfer may be useful. More specifically, the improvement in learning performance in the target domain depends heavily on the complexity of the target domain. The effectiveness of transfer in complex games aligns with the common-sense notion of hierarchical knowledge representation, as argued previously in transfer learning (Konidaris et al., 2012) as well as in biology (Anderson, 2010; Milo et al., 2002). It will be interesting to investigate whether the same principles extend to other general video game playing platforms, such as VGDL (Perez et al., 2015; Schaul, 2013). Such work should help understand how subsymbolic knowledge can be recycled indefinitely across diverse domains.

Transfer is likely inefficient in simpler games due to the effort involved in finding the necessary connections for reusing knowledge from a given source network effectively, in which case it is more efficient to relearn from scratch. For particular low-complexity games, it can also be seen that `random` consistently outperforms both `scratch` and `transfer` (e.g., pong). The initial flexibility of untrained parameters in the `random` condition may explain this result. Unfreezing reused networks, and allowing them to change with a low learning rate may help close this gap. In the methods presented in the following four chapters, no freezing is performed, and the benefits from learning across multiple tasks are more consistent.

Some `transfer` pairs do not consistently outperform training from `scratch` or `random`, indicating negative transfer. This highlights the importance of source and target selection in transfer learning. These results have taken a step towards answering the target-selection problem: What kinds of games make good targets for transfer? More data across many more games is required to answer the source-selection problem: For a given game, what sources should be used? A next step will involve pooling multiple candidate sources and testing GRUSM-ESP's ability to exploit the most useful structure available.

Ultimately, methods must be developed that are robust enough to handle cases in which negative transfer is possible. For example, the method introduced in Chapter 8 addresses the case when tasks are drawn from domains with fundamentally different modalities.

Despite negative transfer in some of the setups, the technique of training a classifier to predict transfer success is shown to be a useful approach for helping decide when to transfer: given some space of complex disparate domains, try transfer with a subset of source-target pairs, and use the results to build a classifier to inform when to attempt transfer in the future. In this chapter, domain-characterization features were provided, but domain-agnostic features could be learned from analysis of the networks and/or learning process; this is an interesting avenue for future work.

Another area of future work involves increasing the flexibility in the combined architecture by (1) relaxing the requirement for all transfer connections to be input-to-hidden and output-to-output, (2) allowing deeper architectures for the source and target networks, and (3) including multiple source networks with adaptive connectivity to each. These extensions will promote reuse of subnetworks of varying depth, along with flexible positioning and combination of modules. The following four chapters include these generalizations. Note that, for GRUSM-ESP, as networks become large and plentiful, maintaining full connectivity between layers will become intractable, and it will be necessary to enforcing sparsity. Chapters 7 and 8 present approaches that enforce sparsity, and enable such scaling.

4.6 Conclusion

To investigate the inherent generality of neural network modules, this chapter introduced an approach for general transfer learning in neural networks. The approach minimizes a priori assumptions of task relatedness and makes it possible to learn adaptively in many domains. In a stochastic version of the Atari 2600 general video game-playing platform, a specific implementation developed in this chapter as GRUSM-ESP can boost learning by reusing neural structure across disparate domains. The success of transfer is shown to correlate with intuitive notions of domain complexity. These results indicate the potential for general neural reuse to predictably assist agents in increasingly complex environments. In other words, trained neural networks modules can be inherently general, and this propensity increases as problems become more complex. These observations motivate the system developed in the next chapter, which aims to improve the generality of larger deep models in high-dimensional domains.

Chapter 5

Pseudo-task Augmentation

This chapter[1] investigates the effects of forcing a single deep module to be more general, by training it with additional pseudo-tasks. As described in Section 2.3.1, deep multitask learning boosts performance by sharing learned structure across related tasks. This chapter adapts ideas from deep multitask learning to the setting where only a single task is available. The method is formalized as *pseudo-task augmentation*, in which models are trained with multiple decoders for each task. The additional pseudo-tasks simulate the effect of training towards closely-related tasks drawn from the same universe. This formalization is a special case of the framework introduced in Chapter 3, in which a single encoder module simultaneously solves all pseudo-tasks. In a suite of experiments, pseudo-task augmentation improves performance on single-task learning problems. When combined with multitask learning, further improvements are achieved, including state-of-the-art performance on the CelebA dataset, showing that pseudo-task augmentation and multitask learning have complementary value. All in all, pseudo-task augmentation is a broadly applicable and efficient way to boost performance in deep learning systems.

[1]The content of this chapter was previously presented at ICML (Meyerson & Miikkulainen, 2018b). Risto Miikkulainen provided guidance and feedback through discussions.

Furthermore, the success of PTA in making a single module more general motivates the systems in subsequent sections, which train multiple modules across more diverse pseudo-task locations.

5.1 Introduction

This chapter adapts ideas from deep MTL to the single-task learning (STL) case, i.e., when only a single task is available for training. The method is formalized as *pseudo-task augmentation* (PTA), in which a single task has multiple distinct decoders projecting the output of the shared structure to task predictions. By training the shared structure to *solve the same problem in multiple ways*, PTA simulates the effect of training towards distinct but closely-related tasks drawn from the same universe. Theoretical justification shows how training dynamics with multiple pseudo-tasks strictly subsumes training with just one, and a class of algorithms is introduced for controlling pseudo-tasks in practice.

In an suite of experiments, PTA is shown to significantly improve performance in single-task settings. Although different variants of PTA traverse the space of pseudo-tasks in qualitatively different ways, they all demonstrate substantial gains. Experiments also show that when PTA is combined with MTL, further improvements are achieved, including state-of-the-art performance on the CelebA dataset. In other words, although PTA can be seen as a base case of MTL, PTA and MTL have complementary value in learning more generalizable models. The conclusion is that pseudo-task augmentation is an efficient, reliable, and broadly applicable method for boosting performance in deep learning systems.

The remainder of this chapter is organized as follows: Section 2.3 covers background on deep learning methods that train multiple models; Section 5.2 introduces the pseudo-task augmentation framework and practical implementations; Section 5.3 describes experimental setups and results; Sections 5.4 and 5.5 discuss future work and overall implications.

5.2 Pseudo-task Augmentation (PTA)

This section introduces the PTA method. First, the classical deep MTL approach is extended to the case of multiple decoders per task. Then, the concept of a pseudo-task is introduced, and increased training dynamics under multiple pseudo-tasks is demonstrated. Finally, practical methods for controlling pseudo-tasks during training are described, which will be compared empirically in Section 5.3.

5.2.1 A Classical Approach

The most common approach to deep MTL is still the "classical" approach (Eq. 2.7), in which all layers are shared across all tasks up to a high level, after which each task learns a distinct decoder that maps high-level points to its task-specific output space (Caruana, 1998; Ranjan et al., 2016; Lu et al., 2017). Even when more sophisticated methods are developed, the classical approach is often used as a baseline for comparison. The classical approach is also computationally efficient, in that the only additional parameters beyond a single task model are in the additional decoders. Thus, when applying ideas from deep MTL to single-task multi-model learning, the classical approach is a natural starting point.

Consider again the case where there are T distinct *true* tasks, but now let there be D decoders for each task. Then, the model for the dth decoder of the tth task is given by

$$\hat{\boldsymbol{y}}_{tdi} = \mathcal{D}_{td}(\mathcal{F}(\boldsymbol{x}_{ti}; \theta_{\mathcal{F}}); \theta_{\mathcal{D}_{td}})\,, \tag{5.1}$$

and the overall loss for the joint model from Eq. 2.8 becomes

$$\theta^* = \operatorname*{argmin}_{\theta} \frac{1}{TD} \sum_{t=1}^{T} \frac{1}{N_t} \sum_{i=1}^{N_t} \sum_{d=1}^{D} \mathcal{L}(\boldsymbol{y}_{ti}, \hat{\boldsymbol{y}}_{tdi})\,, \tag{5.2}$$

where $\theta = (\{\{\theta_{\mathcal{D}_{td}}\}_{d=1}^{D}\}_{t=1}^{T}, \theta_{\mathcal{F}})$. In the same way as the classical approach to MTL encourages $\mathcal{F}$ to be more general and robust by requiring it to support multiple tasks, here $\mathcal{F}$ is required to support *solving the same task in multiple ways.* A visualization of a resulting joint model is shown in Figure 5.1. A theme in MTL is that models for related tasks will have similar decoders, as implemented by explicit regularization (Evgeniou & Pontil, 2004; Kumar & Daumé, 2012; Long et al., 2017; Yang & Hospedales, 2017). Similarly, in Eq. 5.2, through training, two decoders for the same task will instantiate similar models, and, as long as they do not converge completely to equality, they will simulate the effect of training with multiple closely-related tasks.

Notice that the innermost summation in Eq. 5.2 is over decoders. This calculation is computationally efficient: because each decoder for a given task

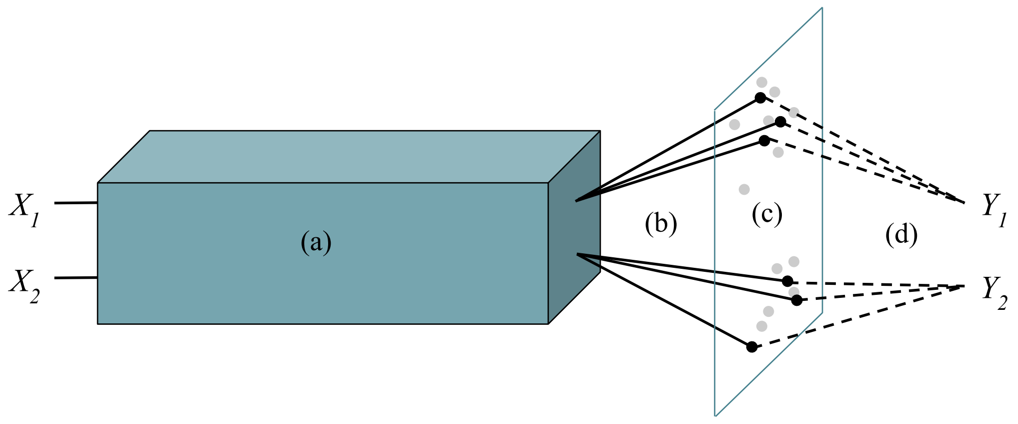


Figure 5.1: General setup for pseudo-task augmentation with two tasks. (a) *Underlying model.* All task inputs are embedded through an underlying model that is completely shared; (b) *Multiple decoders.* Each task has multiple decoders (solid black lines) each projecting the embedding to a distinct classification layer; (c) *Parallel traversal of model space.* The underlying model coupled with a decoder defines a task model. Task models populate a model space, with current models shown as black dots and previous models shown as gray dots; (d) *Multiple loss signals.* Each current task model receives a distinct loss to compute its distinct gradient. A task coupled with a decoder and its parameters defines a pseudo-task for the underlying model.

takes the same input, $\mathcal{F}(\boldsymbol{x}_{ti})$ (usually the most expensive part of the model) need only be computed once per sample (and only once over all tasks if all tasks share $\boldsymbol{x}_{ti}$). However, when evaluating the performance of a model, since each decoder induces a distinct model for a task, what matters is not the average over decoders, but the best performing decoder for each task, i.e.,

$$\theta^*_{\text{eval}} = \operatorname*{argmin}_{\theta} \frac{1}{T} \sum_{t=1}^{T} \frac{1}{N_t} \operatorname*{argmin}_{d \in 1..D} \sum_{i=1}^{N_t} \mathcal{L}(\boldsymbol{y}_{ti}, \hat{\boldsymbol{y}}_{tdi}). \tag{5.3}$$

Eq. 5.2 is used in training because it is smoother; Equation 5.3 is used for model validation, and to select the best performing decoder for each task from the final joint model. This decoder is then applied to future data, e.g., a holdout set. Once the models are trained, in principle they form a set of distinct and equally powerful models for each task. It may therefore be tempting to ensemble them for evaluation, i.e.,

$$\theta^*_{\text{ens}} = \underset{\theta}{\operatorname{argmin}} \frac{1}{T} \sum_{t=1}^{T} \frac{1}{N_t} \sum_{i=1}^{N_t} \mathcal{L}\left(\boldsymbol{y}_{ti}, \frac{1}{D} \sum_{d=1}^{D} \hat{\boldsymbol{y}}_{tdi}\right). \tag{5.4}$$

However, with linear decoders, *training* with Eq. 5.4 is equivalent to training with a single decoder for each task, while training with Eq. 5.2 with multiple decoders yields more expressive training dynamics. These ideas are developed more fully in the next section.

5.2.2 Pseudo-tasks Defined by Decoders

Following the intuition that training $\mathcal{F}$ with multiple decoders amounts to solving the task in multiple ways, each "way" is defined by a pseudo-task

$$(\mathcal{E}_{td} = I, \theta_{\mathcal{E}_{td}} = \emptyset, \mathcal{D}_{td}, \theta_{\mathcal{D}_{td}} = \theta_{td}, \{\boldsymbol{x}_{ti}, \boldsymbol{y}_{ti}\}_{i=1}^{N_t}) \tag{5.5}$$

of the underlying task $\{\boldsymbol{x}_{ti}, \boldsymbol{y}_{ti}\}_{i=1}^{N_t}$, where I is the identity function. This is a special case of the definition of psuedo-task from Chapter 3, where all encoders $\mathcal{E}_{td}$ are the identity function. Since the encoders are trivial, we can write $\theta_{\mathcal{D}_{td}} = \theta_{td}$ for clarity. Now, when $D > 1$, training $\mathcal{F}$ amounts to training each task with multiple pseudo-tasks for each task at each gradient update step. Thus, $\mathcal{F}$ becomes a functional module that solves all such pseudo-tasks simultaneously. This process is the essence of PTA.

As a first step, this chapter considers linear decoders, i.e. each $\mathcal{D}_{td}$ consists of a single dense layer of weights (any following nonlinearity can be considered part of the loss function). Prior work has assumed that models for closely-related tasks differ only by a linear transformation (Evgeniou & Pontil, 2004; Kang et al., 2011; Argyriou et al., 2008). Similarly, with linear decoders, distinct pseudo-tasks for the same task simulate multiple closely-related tasks. When θ_{td} are considered fixed, the learning problem (Eq. 5.2) reduces to

$$\theta^*_{\mathcal{F}} = \operatorname*{argmin}_{\theta_{\mathcal{F}}} \frac{1}{TD} \sum_{t=1}^{T} \frac{1}{N_t} \sum_{i=1}^{N_t} \sum_{d=1}^{D} \mathcal{L}(\boldsymbol{y}_{ti}, \hat{\boldsymbol{y}}_{tdi})\,. \tag{5.6}$$

In other words, although the overall goal is to learn models for T tasks, $\mathcal{F}$ is at each step optimized towards DT pseudo-tasks. Thus, training with multiple decoders may yield positive effects similar to training with multiple true tasks.

After training, the best model for a given task is selected from the final joint model, and used as the final model for that task (Eq. 5.3). Of course, using multiple decoders with identical architectures for a single task does not make the final learned predictive models more expressive. It is therefore natural to ask whether including additional decoders has any fundamental effect on learning dynamics. It turns out that even in the case of linear decoders, the training dynamics of using multiple pseudo-tasks strictly subsumes using just one.

Definition 5.2.1 (Pseudo-task Simulation)**.** *A set of pseudo-tasks S_1* simulates *another S_2 on $\mathcal{F}$ if for all $\theta_{\mathcal{F}}$ the gradient update to $\theta_{\mathcal{F}}$ when trained with S_1 is equal to that with S_2.*

Theorem 5.2.1 (Augmented Training Dynamics). *There exist differentiable functions $\mathcal{F}$ and sets of pseudo-tasks of a single task that cannot be simulated by a single pseudo-task of that task, even when all decoders are linear.*

Proof. Consider a task with a single sample $(\boldsymbol{x}, y)$, where y is a scalar. Suppose $\mathcal{L}$ (from Eq. 5.6) computes mean squared error, $\mathcal{F}$ has output dimension M, and all decoders are linear, with bias terms omitted for clarity. $\mathcal{D}_d$ is then completely specified by the vector $\boldsymbol{w}_d = \langle w_d^1, w_d^2, ..., w_d^M \rangle^\top$. Suppose parameter updates are performed by gradient descent. The update rule for $\theta_\mathcal{F}$ with fixed decoders $\{\mathcal{D}_d\}_{d=1}^D$ and learning rate α is then given by

$$\theta_\mathcal{F} := \theta_\mathcal{F} - \alpha \sum_{d=1}^{D} \nabla_\mathcal{F} \big(y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x}; \theta_\mathcal{F})\big)^2 . \tag{5.7}$$

For a single fixed decoder to yield equivalent behavior, it must have equivalent update steps. The goal then is to choose $(\boldsymbol{x}, y)$, $\mathcal{F}$, $\{\theta_k\}_{k=1}^K$, $\{\boldsymbol{w}_d\}_{d=1}^D$, and $\alpha > 0$, such that there are no $\boldsymbol{w}_o$, $\gamma > 0$, for which $\forall\, k$

$$\begin{aligned} \alpha \sum_{d=1}^{D} \nabla_\mathcal{F} (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x}; \theta_k))^2 &= \gamma \nabla_\mathcal{F} (y - \boldsymbol{w}_o^\top \mathcal{F}(\boldsymbol{x}; \theta_k))^2 \\ \implies \alpha \sum_{d=1}^{D} 2(y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x}; \theta_k)) \boldsymbol{w}_d^\top J_\mathcal{F}(\boldsymbol{x}; \theta_k) &= \\ 2\gamma(y - \boldsymbol{w}_o^\top \mathcal{F}(\boldsymbol{x}; \theta_k)) \boldsymbol{w}_o^\top J_\mathcal{F}(\boldsymbol{x}; \theta_k) , \end{aligned} \tag{5.8}$$

where $J_\mathcal{F}$ is the Jacobian of $\mathcal{F}$. By choosing $\mathcal{F}$ and $\{\theta_k\}_{k=1}^K$ so that all $J_\mathcal{F}(\boldsymbol{x}; \theta_k)$ have full row rank, Eq. 5.8 reduces to

$$\alpha \sum_{d=1}^{D} (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x}; \theta_k)) w_d^i = \gamma (y - \boldsymbol{w}_o^\top \mathcal{F}(\boldsymbol{x}; \theta_k)) w_o^i \quad \forall\, i \in 1..M. \tag{5.9}$$

Choosing $\mathcal{F}$, $\{\theta_k\}_{k=1}^K$, $\{\boldsymbol{w}_d\}_{d=1}^D$, and $\alpha > 0$ such that the left hand side of Eq. 5.9 is never zero, we can safely write

$$\frac{\sum_{d=1}^D (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x};\theta_k))w_d^i}{(y - \boldsymbol{w}_o^\top \mathcal{F}(\boldsymbol{x};\theta_k))w_o^i} = \frac{\sum_{d=1}^D (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x};\theta_k))w_d^j}{(y - \boldsymbol{w}_o^\top \mathcal{F}(\boldsymbol{x};\theta_k))w_o^j} \quad \forall\ (i,j)$$
$$\implies \frac{\sum_{d=1}^D (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x};\theta_k))w_d^i}{\sum_{d=1}^D (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x};\theta_k))w_d^j} = \frac{w_o^i}{w_o^j}\,. \qquad (5.10)$$

Then, since $\boldsymbol{w}_o$ is fixed, it suffices to find $\mathcal{F}(\boldsymbol{x};\theta_1)$, $\mathcal{F}(\boldsymbol{x};\theta_2)$ such that for some (i,j)

$$\frac{\sum_{d=1}^D (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x};\theta_1))w_d^i}{\sum_{d=1}^D (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x};\theta_1))w_d^j} \neq \frac{\sum_{d=1}^D (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x};\theta_2))w_d^i}{\sum_{d=1}^D (y - \boldsymbol{w}_d^\top \mathcal{F}(\boldsymbol{x};\theta_2))w_d^j}\,. \qquad (5.11)$$

For instance, with $D = 2$, choosing $y = 1$, $w_1 = \langle 2, 3\rangle^\top$, $w_2 = \langle 4, 5\rangle^\top$, $\mathcal{F}(\boldsymbol{x};\theta_1) = \langle 6, 7\rangle^\top$, and $\mathcal{F}(\boldsymbol{x};\theta_1) = \langle 8, 9\rangle^\top$ satisfies the inequality. Note $\mathcal{F}(\boldsymbol{x};\theta_1)$ and $\mathcal{F}(\boldsymbol{x};\theta_2)$ can be chosen arbitrarily since $\mathcal{F}$ is only required to be differentiable, e.g., implemented by a neural network. $\square$

Showing that a single pseudo-task can be simulated by D pseudo-tasks for any $D > 1$ is more direct: For any $\boldsymbol{w}_o$ and γ, choose $\boldsymbol{w}_d = \boldsymbol{w}_o\ \forall\ d \in 1..D$ and $\alpha = \gamma/D$. Further extensions to tasks with more samples, higher dimensional outputs, and cross-entropy loss are straightforward. Note that this result is related to work on the dynamics of deep linear models (Saxe et al., 2014), in that adding additional linear structure complexifies training dynamics. However, training an ensemble directly, i.e., via Eq. 5.4, does *not* yield augmented training dynamics, since

$$\frac{1}{D}\sum_{d=1}^{D}\hat{\boldsymbol{y}}_{tdi} = \frac{1}{D}\sum_{d=1}^{D}\boldsymbol{W}_{td}^{\top}\mathcal{F}(\boldsymbol{x}_{ti};\theta_{\mathcal{F}})$$

$$\implies \boldsymbol{W}_{to}^{\top} = \frac{1}{D}\sum_{d=1}^{D}\boldsymbol{W}_{td}^{\top} \text{ and } \beta = \alpha\,. \quad (5.12)$$

Now that we know that training with additional pseudo-tasks yields augmented training dynamics that may be exploited, the question is how to take advantage of these dynamics in practice. The next section introduces methods to address this question.

5.2.3 Control of Multiple Pseudo-task Trajectories

Given linear decoders, the primary goal is to optimize $\mathcal{F}$; if an optimal $\mathcal{F}$ were found, optimal decoders for each task could be derived analytically. So, given multiple linear decoders for each task, how should their induced pseudo-tasks be controlled to maximize the benefit to $\mathcal{F}$? For one, their weights $\{\boldsymbol{W}_{td}\}_{d=1}^{D}$ must not all be equal, otherwise we would have $\boldsymbol{W}_{to} = \boldsymbol{W}_{t1}$ and $\gamma = D\alpha$ in the proof of Theorem 5.2.1. Following Eq. 5.2, decoders can be trained jointly with $\mathcal{F}$ via gradient-based methods, so that they learn to work well with $\mathcal{F}$. Through optimization, a trained decoder induces a trajectory of pseudo-tasks. Going beyond this *implicit* control, Algorithm 5.1 gives a high-level framework for applying *explicit* control to pseudo-task trajectories.

An instance of the algorithm is parameterized by choices for *DecInitialize*, which defines how decoders are initialized; and *DecUpdate*, which defines non-

Algorithm 5.1 PTA Training Framework

```
1: Given $T$ tasks $\{\{\boldsymbol{x}_{ti}, \boldsymbol{y}_{ti}\}_{i=1}^{N_t}\}_{t=1}^{T}$, and $D$ decoders per task
2: $\{\{\theta_{\mathcal{D}_{td}}\}_{d=1}^{D}\}_{t=1}^{T} \leftarrow \textit{DecInitialize}()$
3: Initialize $\theta_{\mathcal{F}}$
4: Initialize decoder costs $c_{td} \leftarrow \infty \ \forall \ (t, d)$
5: while not done training do
6:     for $m = 1$ to $M$ do                                  ▷ $M$ is meta-iteration length
7:         Update $\theta_{\mathcal{F}}$ and $\theta_{\mathcal{D}_{td}}$ via a joint gradient step.
8:     for $t = 1$ to $T$ do
9:         for $d = 1$ to $D$ do
10:            $c_{td} \leftarrow \text{evaluate}(\theta_{\mathcal{D}_{td}}, \theta_{\mathcal{F}}, t)$            ▷ e.g., get validation error
11:        for $d = 1$ to $D$ do
12:            $\theta_{\mathcal{D}_{td}} \leftarrow \textit{DecUpdate}\big(d, \{\theta_{\mathcal{D}_{td_o}}, c_{td_o}\}_{d_o=1}^{D}\big)$
13: return $\big(\{\{\theta_{\mathcal{D}_{td}}\}_{d=1}^{D}\}_{t=1}^{T}, \theta_{\mathcal{F}}\big)$
```

gradient-based updates to decoders every M gradient steps, i.e., every *meta-iteration*, based on the performance of each decoder (*DecUpdate* defaults to no-op). As a first step, several intuitive methods are evaluated in this chapter for instantiating Algorithm 5.1. These methods can be used together in any combination:

Independent Initialization (I) *DecInitialize* randomly initializes all $\theta_{\mathcal{D}_{td}}$ independently. This is the obvious initialization method, and is assumed in all methods below.

Freeze (F) *DecInitialize* freezes all decoder weights except $\theta_{\mathcal{D}_{t1}}$ for each task. Frozen weights do not receive gradient updates in Line 7 of Algorithm 5.1. Because they cannot adapt to $\mathcal{F}$, constant pseudo-task trajectories provide a stricter constraint on $\mathcal{F}$. One decoder is left unfrozen so that the optimal model for each task can still be learned.

Independent Dropout (D) *DecInitialize* sets up the dropout layers preceding linear decoder layers to drop out values independently for each decoder. Thus, even when the weights of two decoders for a task are equal, their resulting gradient updates to $\mathcal{F}$ and to themselves will be different.

For the next three methods, let $c_t^{\min} = \min(c_{t1}, \ldots, c_{tD})$.

Perturb (P) *DecUpdate* adds noise $\sim \mathcal{N}(\mathbf{0}, \epsilon_p \boldsymbol{I})$ to each $\theta_{\mathcal{D}_{td}}$ for all d where $c_{td} \neq c_t^{\min}$. This method ensures that $\theta_{\mathcal{D}_{td}}$ are sufficiently distinct before each training period.

Hyperperturb (H) Like *Perturb*, except *DecUpdate* updates the hyperparameters of each decoder other than the best for each task, by adding noise $\sim \mathcal{N}(0, \epsilon_h)$. In this chapter, each decoder has only one hyperparameter: the dropout rate of any *Independent Dropout* layer, because adapting dropout rates can be beneficial (Ba & Frey, 2013; Li et al., 2016; Jaderberg et al., 2017a).

Greedy (G) For each task, let $\theta_t^{\min}$ be the weights of a decoder with cost $c_t^{\min}$. *DecUpdate* updates all $\theta_{td} \coloneqq \theta_t^{\min}$, including hyperparameters. This biases training to explore the highest-performing areas of the pseudo-task space. When combined with any of the previous three methods, decoder weights are still ensured to be distinct through training.

Combinations of these six methods induce an initial class of PTA training algorithms PTA-* for the case of linear decoders. The next section evaluates eight representative combinations of these methods, i.e., PTA-I, PTA-F, PTA-P, PTA-D, PTA-FP, PTA-GP, PTA-GD, and PTA-HGD, in various experimental settings. Note that H and G are related to methods that copy the weights of the entire network (Jaderberg et al., 2017a). Also note that, in a possible future extension to the nonlinear case, the space of possible PTA control methods becomes much more broad, as will be discussed in Section 5.4.

5.3 Experiments

In this section, PTA methods are evaluated and shown to excel in a range of settings: (1) single-task character recognition; (2) multitask character recognition; (3) single-task sentiment classification; and (4) multitask visual attribute classification. All experiments are implemented using the Keras framework (Chollet et al., 2015). For PTA-P and PTA-GP, $\epsilon_p = 0.01$; for PTA-HGD, $\epsilon_h = 0.1$ and dropout rates range from 0.2 to 0.8. A dropout layer with dropout rate initialized to 0.5 precedes each decoder.

5.3.1 Omniglot Character Recognition

This section evaluates and compares the various PTA methods on Omniglot character recognition (Lake et al., 2015). The Omniglot dataset consists of 50 alphabets of handwritten characters, each of which induces its own character recognition task. Each character instance is a 105×105 black-

and-white image, and each character has 20 instances, each drawn by a different individual. To reduce variance and improve reproducibility of experiments, a fixed random 50/20/30% train/validation/test split was used for each task. Methods are evaluated with respect to all 50 tasks as well as a subset consisting of the first 20 tasks in a fixed random ordering of alphabets used in previous work (Meyerson & Miikkulainen, 2018a). The underlying model $\mathcal{F}$ for all setups is a simple four layer convolutional network that has been shown to yield good performance on Omniglot (Meyerson & Miikkulainen, 2018a). This model has four convolutional layers each with 53 filters and 3×3 kernels, and each followed by a 2×2 max-pooling layer and dropout layer with 0.5 dropout probability. At each meta-iteration, 250 gradient updates are performed via Adam (Kingma & Ba, 2014); each setup is trained for 100 meta-iterations.

5.3.1.1 Omniglot: Single-task Learning

The single-task learning case is considered first. For each of the 20 initial Omniglot tasks, the eight PTA methods were applied to the task with 2, 3, and 4 decoders. At least three trials were run with each setup; the mean performance averaged across trials and tasks is shown in Figure 5.2. Every PTA setup outperforms the baseline, i.e., training with a single decoder. The methods that use decoder freezing, PTA-F and PTA-FP, perform best, showing how this problem can benefit from strong regularization. Notably, the mean improvement across all methods increases with D: 1.86% for $D = 2$; 2.33% for $D = 3$; and 2.70% for $D = 4$. Like MTL can benefit from adding more tasks

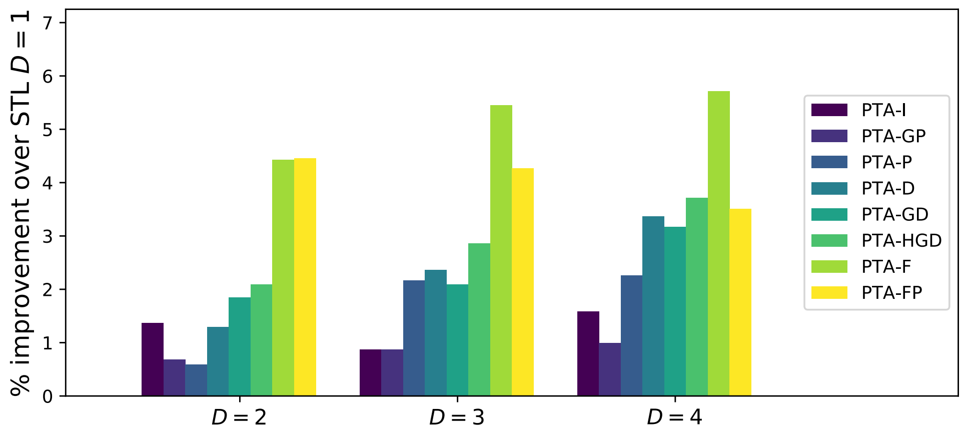


Figure 5.2: **Omniglot single-task learning results.** For each number of decoders D, mean improvement (absolute % decrease in error) over $D = 1$ is plotted for each setup, averaged across all tasks. All setups outperform the baseline. PTA-F and PTA-FP performs best, as this problem benefits from strong regularization. The mean improvement across all methods also increases with D: 1.86% for $D = 2$; 2.33% for $D = 3$; and 2.70% for $D = 4$.

(Caruana, 1998; Hashimoto et al., 2017; Jaderberg et al., 2017b), single-task learning can benefit from adding more pseudo-tasks.

5.3.1.2 Omniglot: Multitask Learning

Omniglot models have also been shown to benefit from MTL (Maclaurin et al., 2015; Rebuffi et al., 2017; Yang & Hospedales, 2017; Meyerson & Miikkulainen, 2018a). This section extends the experiments in Section 5.3.1.1 to MTL. The setup is exactly the same, except now the underlying convolutional model is fully shared across all tasks for each method. The results are shown in Figure 5.3. All setups outperform the STL baseline, and all, except for PTA-I with $D = 2$, outperform the MTL baseline. Again, PTA-F and PTA-FP

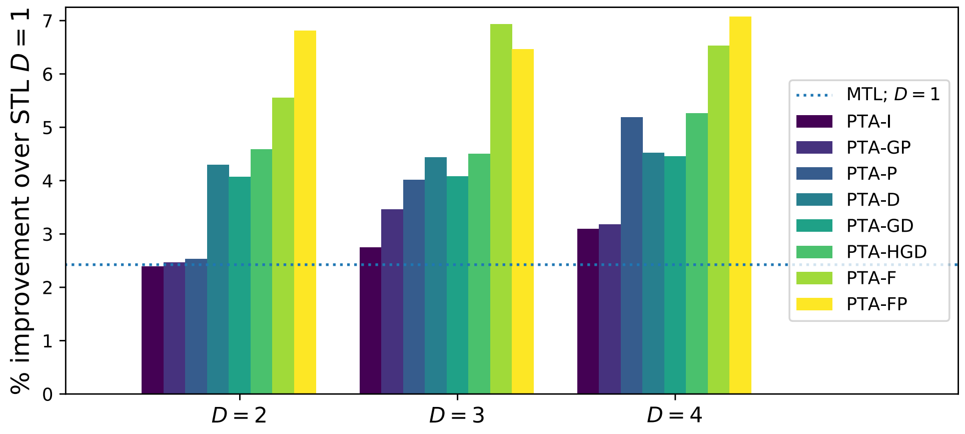


Figure 5.3: **Omniglot multitask learning results.** For each number of decoders D, mean improvement (absolute % decrease in error) across all tasks is plotted over STL with $D = 1$. All setups outperform the STL baseline, and all except PTA-I with $D = 2$ outperform the MTL baseline. Again, PTA-F and PTA-FP perform best, and the mean improvement across all methods increases with D: 3.63% for $D = 2$; 4.07% for $D = 3$; and 4.37% for $D = 4$.

perform best, and the mean improvement across all methods increases with D. The results show that although PTA implements behavior similar to MTL, when combined, their positive effects are complementary. Finally, to test the scalability of these results, three diverse PTA methods with $D = 4$ and $D = 10$ were applied to the complete 50-task dataset: PTA-I, because it is the baseline PTA method; PTA-F, because it is simple and high-performing; and PTA-HGD, because it is the most different from PTA-F, but also relatively high-performing. The results are given in Table 5.1. The results agree with the 20-task results, with all methods improving upon the baseline, and performance overall improving as D is increased.

Method	Single-task Learning		Multitask Learning	
Baseline	35.49		29.02	
	$D = 4$	$D = 10$	$D = 4$	$D = 10$
PTA-I	31.72	32.56	27.26	24.50
PTA-HGD	31.63	30.39	25.77	26.55
PTA-F	**29.37**	**28.48**	**23.45**	**23.36**
PTA-Mean	30.91	30.48	25.49	24.80

Table 5.1: **Omniglot 50-task results.** Test error averaged across all tasks for each setup is shown. Overall, the performance gains from MTL complement those from PTA, with PTA-F again the highest-performing and most robust method.

5.3.2 IMDB Sentiment Analysis

The experiments in this section apply PTA to LSTM models in the IMDB sentiment classification problem (Maas et al., 2011). The dataset consists of 50K natural-language movie reviews, 25K for training and 25K for testing. There is a single binary classification task: whether a review is positive or negative. As in previous work, 2500 of the training reviews are withheld for validation (McCann et al., 2017). The underlying model $\mathcal{F}$ is the off-the-shelf LSTM model for IMDB provided by Keras, with no parameters or preprocessing changed. In particular, the vocabulary is capped at 20K words, the LSTM layer has 128 units and dropout rate 0.2, and each meta-iteration consists of one epoch of training with Adam (Kingma & Ba, 2014). This is not a state-of-the-art model, but it is a very different architecture from that used in Omniglot, and therefore serves to demonstrate the broad applicability of PTA.

Method	Test Accuracy %	
LSTM Baseline ($D = 1$)	82.75 (±0.13)	
	$D = 4$	$D = 10$
PTA-I	83.20 (±0.07)	83.02 (±0.11)
PTA-HGD	83.22 (±0.05)	**83.51** (±0.08)
PTA-F	**83.30** (±0.12)	83.30 (±0.08)

Table 5.2: **IMDB Results.** All PTA methods outperform the LSTM baseline. The best performance is achieved by PTA-HGD with $D = 10$. This method receives a substantial boost from increasing the number of decoders from 4 to 10, as the greedy algorithm gets to perform broader search. On the other hand, PTA-I and PTA-F do not improve with the additional decoders, suggesting that, without careful control, too many decoders can overconstrain $\mathcal{F}$.

The final three PTA methods from Section 5.3.1 were evaluated with 4 and 10 decoders (Table 5.2). As in Section 5.3.1, all PTA methods outperform the baseline. In this case, however, PTA-HGD with $D = 10$ performs best. Notably, PTA-I and PTA-F do not improve from $D = 4$ to $D = 10$, suggesting that underlying models have a critical point after which, without careful control, too many decoders can be overconstraining. To contrast PTA with standard regularization, additional Baseline experiments were run with dropout rates $[0.3, 0.4, ..., 0.9]$. At 0.5 the best accuracy was achieved: 83.14 (±0.05), which is less than all PTA variants except PTA-I with $D = 10$, thus confirming that PTA adds value. To help understand what each PTA method is actually doing, snapshots of decoder parameters taken every epoch are visualized in Figure 5.4 with t-SNE (van der Maaten & Hinton, 2008) using cosine distance. The behavior matches our intuition for what should be happening in each

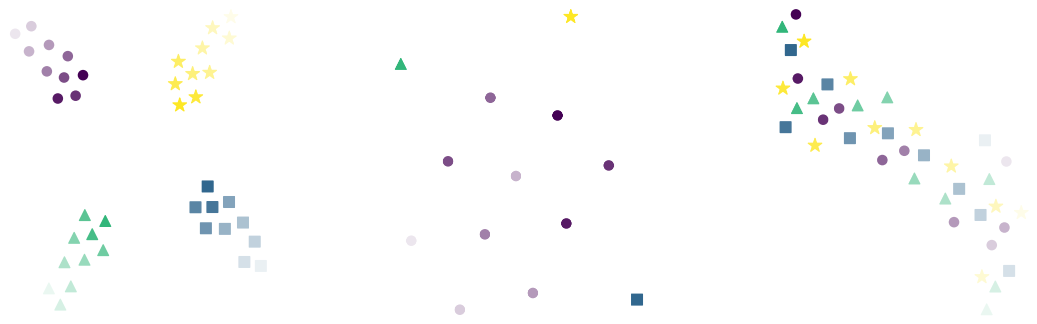


Figure 5.4: **Pseudo-task Trajectories.** t-SNE (van der Maaten & Hinton, 2008) projections of pseudo-task trajectories, for runs of PTA-I, PTA-F, and PTA-HGD on IMDB. Each shape corresponds to a particular decoder; each point is a projection of the length-129 weight vector at the end of an epoch, with opacity increasing by epoch. The behavior matches our intuition for what should be happening in each case: (a) When decoders are only initialized independently, their pseudo-tasks gradually converge; (b) when all but one decoder is frozen, the unfrozen one settles between the others; (c) when a greedy method is used, decoders perform local exploration as they traverse the pseudo-task space together.

case: When decoders are only initialized independently, their pseudo-tasks gradually converge; when all but one decoder is frozen, the unfrozen one settles between the others; and when a greedy method is used, decoders perform local exploration as they traverse the pseudo-task space together.

5.3.3 CelebA Facial Attribute Recognition

To further test applicability and scalability, PTA was evaluated on CelebA large-scale facial attribute recognition (Liu et al., 2015b). The dataset consists of $\approx$200K 178×218 color images. Each image has binary labels for 40 facial attributes; each attribute induces a binary classification task. Facial attributes are related at a high level that deep models can exploit, making CelebA a popular deep MTL benchmark. Thus, this experiment focuses on the MTL setting.

The underlying model was Inception-ResNet-v2 (Szegedy et al., 2016), with weights initialized from training on ImageNet (Russakovsky et al., 2015). Due to computational constraints, only one PTA method was evaluated: PTA-HGD with $D = 10$. PTA-HGD was chosen because of its superior performance on IMDB, and because CelebA is a large-scale problem that may require extended pseudo-task exploration; Figure 5.4 shows how PTA-HGD may support such exploration above other methods. Each meta-iteration consists of 250 gradient updates with batch size 32. The optimizer schedule is co-opted from previous work (Günther et al., 2017): RMSprop is initialized with a learning rate of 10^{-4}, which is decreased to 10^{-5} and 10^{-6} when the model converges. PTA-HGD and the MTL baseline were each trained three times. The computational overhead of PTA-HGD is marginal, since the underlying model has 54M parameters, while each decoder has only 1.5K. Table 5.3 shows the results. PTA-HGD outperforms all other methods, thus establishing a new state-of-the-art in CelebA. Figure 5.5 shows resulting dropout schedules for PTA-HGD. No one type of schedule dominates; PTA-HGD gives each task the flexibility to adapt its own schedule via the performance of its pseudo-tasks.

MTL Method	% Error
Single Task (He et al., 2017)	10.37
MOON (Rudd et al., 2016)	9.06
Adaptive Sharing (Lu et al., 2017)	8.74
MCNN-AUX (Hand & Chellappa, 2017)	8.71
Soft Order (Meyerson & Miikkulainen, 2018a)	8.64
VGG-16 MTL (Lu et al., 2017)	8.56
Adaptive Weighting (He et al., 2017)	8.20
AFFACT (Günther et al., 2017) (best of 3)	8.16
MTL Baseline (Ours; mean of 3)	8.14
PTA-HGD, $D = 10$ (mean of 3)	**8.10**
Ensemble of 3: AFFACT (Günther et al., 2017)	8.00
Ensemble of 3: PTA-HGD, $D = 10$	**7.94**

Table 5.3: **CelebA results.** Comparison of PTA against state-of-the-art methods for CelebA, with and without ensembling. Test error is averaged across all attributes. PTA-HGD outperforms all other methods, establishing a new state-of-the-art in this benchmark, and becoming the first method to crack the 8% barrier on this dataset.

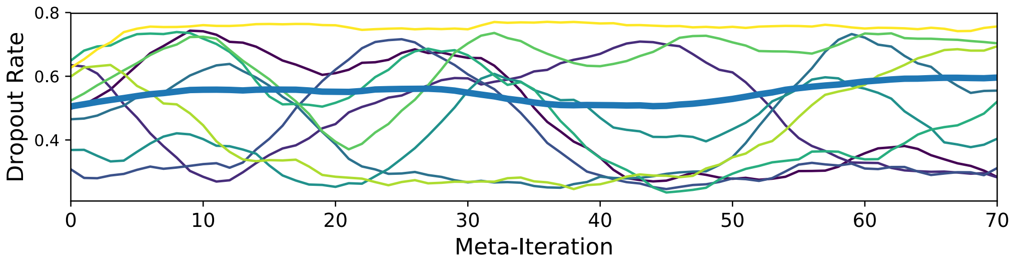


Figure 5.5: **CelebA dropout schedules.** The thick blue line shows the mean dropout schedule across all 400 pseudo-tasks in a run of PTA-HGD. Each of the remaining lines shows the schedule of a particular task, averaged across their 10 pseudo-tasks. All lines are plotted with a simple moving average of length 10. The diversity of schedules shows that the system is taking advantage of PTA-HGD's ability to adapt task-specific hyperparameter schedules.

5.4 Discussion and Future Work

The experiments in this chapter demonstrated that PTA is broadly applicable, and that it can boost performance in a variety of single-task and multitask problems. Training with multiple decoders for a single task allows a broader set of models to be visited. If these decoders are diverse and perform well, then the shared structure has learned to solve the same problem in diverse ways, which is a hallmark of robust intelligence. In the MTL setting, controlling each task's pseudo-tasks independently makes it possible to discover diverse task-specific learning dynamics (Figure 5.5). Increasing the number of decoders can also increase the chance that pairs of decoders align well across tasks.

The crux of PTA is the method for controlling pseudo-task trajectories. Experiments showed that the amount of improvement from PTA is dependent on the choice of control method. Different methods exhibit highly structured but different behavior (Figure 5.4). The success of initial methods indicates that developing more sophisticated methods is a promising avenue of future work. In particular, methods from Section 2.3.2 can be co-opted to control pseudo-task trajectories more effectively. Consider, for instance, the most involved method evaluated in this chapter: PTA-HGD. This online decoder search method could be replaced by methods that generate new models more intelligently (Bergstra et al., 2011; Snoek et al., 2012; Miikkulainen et al., 2017; Real et al., 2017; Zoph & Le, 2017). Such methods will be especially useful in extending PTA beyond the linear case considered in this chapter, to complex nonlinear decoders. For example, since a set of decoders is being trained in parallel, it could be natural to use neural architecture search methods (Miikkulainen et al., 2017; Real et al., 2017; Zoph & Le, 2017) to search for optimal decoder architectures. While ensembling separate PTA models is useful (Table 5.3), in preliminary tests naïvely ensembling decoders for evaluation (Eq. 5.4) did not yield remarkable improvements over the single best (Eq. 5.3). In a further preliminary test with IMDB, when $\mathcal{F}$ was *not* shared, PTA-I outperformed PTA-HGD and PTA-F, indicating that the latter two methods address dynamics that arise in joint training but not naïve ensemble training. Developing PTA training methods for generating a more complementary set of decoders, coupled with effective methods for ensembling this set, could push performance even further, especially when decoders are more complex. PTA also resembles generic data augmentation (Taylor & Nitschke, 2017), except applied to output of the model, instead of the input. It will be interesting to see how PTA performs when combined with modern generic data augmentation methods like cutout (Devries & Taylor, 2017).

5.5 Conclusion

This chapter introduced *pseudo-task augmentation*, a method that makes it possible to apply ideas from deep MTL to single-task learning. By training shared structure to solve the same task in multiple ways, pseudo-task augmentation simulates training with multiple closely-related tasks, yielding performance improvements similar to those in MTL. However, the methods are complementary: Combining pseudo-task augmentation with MTL results in further performance gains. Broadly applicable, pseudo-task augmentation is thus a promising method for improving deep learning performance. Overall, this chapter has taken first steps towards a future class of efficient model search algorithms that exploit intratask parameter sharing. The systems presented in subsequent chapter instantiate such algorithms, using multiple modules, and using modules multiple times within task models.

Chapter 6

Soft Layer Ordering

This chapter[1] takes PTA to an extreme, by training each module at each possible depth in each task model. This approach is presented in contrast to an underlying assumption of existing deep multitask learning (MTL) methods: They align layers shared between tasks in a *parallel ordering.* Such an organization significantly constricts the types of shared structure that can be learned. In this chapter, the necessity of parallel ordering for deep MTL is first tested by comparing it with *permuted ordering* of shared layers, in which layers must solve qualitatively distinct pseudo-tasks for different underlying tasks. The results indicate that a flexible ordering can enable more effective sharing; thus, they lead to a *soft ordering* approach, which learns *how* shared layers are applied in different ways for different tasks. The resulting models apply a distinct mixture of the available set of layers at each pseudo-task location, that is, at each depth for each task. Deep MTL with soft ordering outperforms parallel ordering methods across a series of domains. These results suggest that the power of deep MTL comes from learning highly general building blocks that can be assembled to meet the demands of each task. Chapters 7 and 8

[1]The content of this chapter was previously presented at ICLR (Meyerson & Miikkulainen, 2018a). Risto Miikkulainen provided guidance and feedback through discussions.

then aim to scale this idea towards more practical deep learning systems.

6.1 Introduction

Although existing multitask learning approaches typically improve performance over single-task learning in these settings, these approaches have generally been constrained to joint training of relatively few and/or closely-related tasks. On the other hand, from a perspective of Kolmogorov complexity, "transfer should always be useful"; any pair of distributions underlying a pair of tasks must have *something* in common (Mahmud, 2009; Mahmud & Ray, 2008). In principle, even tasks that are "superficially unrelated" such as those in vision and NLP can benefit from sharing (even without an *adaptor* task, such as image captioning). In other words, for a sufficiently expressive class of models, the inductive bias of requiring a model to fit multiple tasks simultaneously should encourage learning to converge to more *realistic* representations. The expressivity and success of deep models suggest they are ideal candidates for improvement via MTL. So, why have existing approaches to deep MTL been so restricted in scope?

MTL is based on the assumption that learned transformations can be shared across tasks. This chapter identifies an additional implicit assumption underlying existing approaches to deep MTL: this sharing takes place through *parallel ordering* of layers. That is, sharing between tasks occurs only at aligned levels (layers) in the feature hierarchy implied by the model architecture. This constraint limits the kind of sharing that can occur between tasks. It requires subsequences of task feature hierarchies to match, which may be difficult to establish as tasks become plentiful and diverse.

This chapter investigates whether parallel ordering of layers is necessary for deep MTL. As an alternative, it introduces methods that make deep MTL more flexible. First, existing approaches are reviewed in the context of their reliance on parallel ordering. Then, as a foil to parallel ordering, *permuted ordering* is introduced, in which shared layers are applied in different orders for different tasks. These permutations force each layer to have more generic functionality, since it must solve qualitatively distinct pseudo-tasks for different underlying tasks. The increased ability of permuted ordering to support integration of information across tasks is analyzed, and the results are used to develop a *soft ordering* approach to deep MTL. In this approach, a joint model learns *how* to apply shared layers in different ways at different depths for different tasks as it simultaneously learns the parameters of the layers themselves. In a suite of experiments, soft ordering is shown to improve performance over single-task learning as well as over fixed order deep MTL methods.

Importantly, soft ordering is not simply a technical improvement, but a new way of thinking about deep MTL. Learning a different soft ordering of layers for each task amounts to discovering a set of *generalizable modules that are assembled in different ways for different tasks.* This perspective points to future approaches that train a collection of layers on a set of training tasks, which can then be assembled in novel ways for future unseen tasks. Some of the most striking structural regularities observed in the natural, technological and sociological worlds are those that are repeatedly observed across settings and scales; they are ubiquitous and universal. By forcing shared transformations to occur at matching depths in hierarchical feature extraction, deep MTL falls short of capturing this sort of functional regularity. Soft ordering is thus a step towards enabling deep MTL to realize the diverse array of structural regularities found across complex tasks drawn from the real world.

6.2 Parallel Ordering of Layers in Deep MTL

Based on the high-level classification of existing deep MTL approaches presented in Section 2.3.1, this section exposes the reliance of these approaches on the *parallel ordering assumption*.

A common interpretation of deep learning is that layers extract progressively higher level features at later depths (Lecun et al., 2015). A natural assumption is then that the learned transformations that extract these features are also tied to the depth at which they are learned. The core assumption motivating MTL is that regularities across tasks will result in learned transformations that can be leveraged to improve generalization. However, the methods reviewed in Section 2.3.1 add the further assumption that *subsequences of the feature hierarchy align across tasks and sharing between tasks occurs only at aligned depths* (Figure 2.5); we call this the *parallel ordering assumption*.

Consider T tasks $t_1, \ldots, t_T$ to be learned jointly, with each t_i associated with a model $y_i = \mathcal{F}_i(x_i)$. Suppose sharing across tasks occurs at D consecutive depths. Let $\mathcal{E}_i$ ($\mathcal{D}_i$) be t_i's task-specific encoder (decoder) to (from) the core sharable portion of the network from its inputs (to its outputs). Let W_k^i be the layer of learned weights (e.g., affine or convolutional) for task i at shared depth k, with ϕ_k an optional nonlinearity. The parallel ordering assumption implies

$$y_i = (\mathcal{D}_i \circ \phi_D \circ W_D^i \circ \phi_{D-1} \circ W_{D-1}^i \circ \ldots \circ \phi_1 \circ W_1^i \circ \mathcal{E}_i)(x_i),$$
$$\text{with } W_k^i \approx W_k^j \;\forall\; (i, j, k). \quad (6.1)$$

The approximate equality "$\approx$" means that at each shared depth the applied weight tensors for each task are similar and compatible for sharing. For example, learned parameters may be shared across all W_k^i for a given k, but not between W_k^i and W_l^j for any $k \neq l$. For closely-related tasks, this assumption may be a reasonable constraint. However, as more tasks are added to a joint model, it may be more difficult for each layer to represent features of its given depth for all tasks. Furthermore, for very distant tasks, it may be unreasonable to expect that task feature hierarchies match up at all, even if the tasks are related intuitively. The conjecture explored in this chapter is that parallel ordering limits the potential of deep MTL by the strong constraint it enforces on the use of each layer.

6.3 Deep MTL with Soft Ordering of Layers

Now that parallel ordering has been identified as a constricting feature of deep MTL approaches, its necessity can be tested, and the resulting observations can be used to develop more flexible methods.

6.3.1 A Foil for the Parallel Ordering Assumption: Permuting Shared Layers

Consider the most common deep MTL setting: hard-sharing of layers, where each layer in $\{W_k\}_{k=1}^{D}$ is shared in its entirety across all tasks. The baseline deep MTL model for each task t_i is given by

$$y_i = (\mathcal{D}_i \circ \phi_D \circ W_D \circ \phi_{D-1} \circ W_{D-1} \circ \ldots \circ \phi_1 \circ W_1 \circ \mathcal{E}_i)(x_i). \tag{6.2}$$

This setup satisfies the parallel ordering assumption. Consider now an alternative scheme, equivalent to the above, except with learned layers applied in different orders for different task. That is,

$$y_i = (\mathcal{D}_i \circ \phi_D \circ W_{\rho_i(D)} \circ \phi_{D-1} \circ W_{\rho_i(D-1)} \circ \ldots \circ \phi_1 \circ W_{\rho_i(1)} \circ \mathcal{E}_i)(x_i), \tag{6.3}$$

where ρ_i is a task-specific permutation of size D, and ρ_i is fixed before training. If there are sets of tasks for which joint training of the model defined by Eq. 6.3 achieves similar or improved performance over Eq. 6.2, then parallel ordering is not a necessary requirement for deep MTL. Of course, in this formulation, it is required that the W_k can be applied in any order. See Section 6.6 for examples of possible generalizations.

Note that this multitask permuted ordering differs from an approach of training layers in multiple orders for a single task. The single-task case results in a model with increased commutativity between layers, a behavior that has also been observed in residual networks (Veit et al., 2016), whereas here the result is *a set of layers that are assembled in different ways for different tasks.*

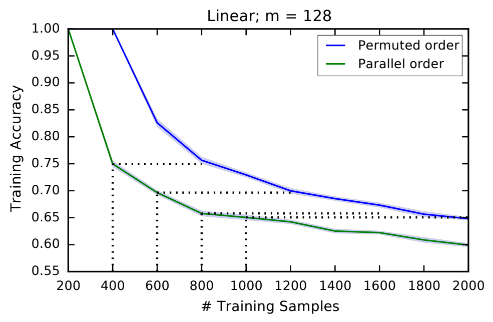
6.3.2 The increased expressivity of permuted ordering

Fitting tasks of random patterns. Permuted ordering is evaluated by comparing it to parallel ordering on a set of tasks. Randomly generated tasks (similar to (Kirkpatrick et al., 2017)) are the most disparate possible tasks, in that they share minimal information, and thus help build intuition for how permuting layers could help integrate information in broad settings. The following experiments investigate how accurately a model can jointly fit two tasks of n samples. The data set for task t_i is $\{(x_{ij}, y_{ij})\}_{j=1}^{n}$, with each x_{ij} drawn uniformly from $[0,1]^m$, and each y_{ij} drawn uniformly from $\{0,1\}$. There are two shared learned affine layers $W_k : \mathbb{R}^m \to \mathbb{R}^m$. The models with permuted ordering (Eq. 6.3) are given by

$$y_1 = (O \circ \phi \circ W_2 \circ \phi \circ W_1)(x_1) \text{ and } y_2 = (O \circ \phi \circ W_1 \circ \phi \circ W_2)(x_2), \quad (6.4)$$

where O is a final shared classification layer. The reference parallel ordering models are defined identically, but with W_k in the same order for both tasks. Note that fitting the parallel model with n samples is equivalent to a single-task model with $2n$. In the first experiment, $m = 128$ and $\phi = I$. Although adding depth does not add expressivity in the single-task linear case, it is useful for examining the effects of permuted ordering, and deep linear networks are known to share properties with nonlinear networks (Saxe et al., 2014). In the second experiment, $m = 16$ and $\phi = \text{ReLU}$.

The results are shown in Figure 6.1. Remarkably, in the linear case, permuted ordering of shared layers does not lose accuracy compared to the

(a) 

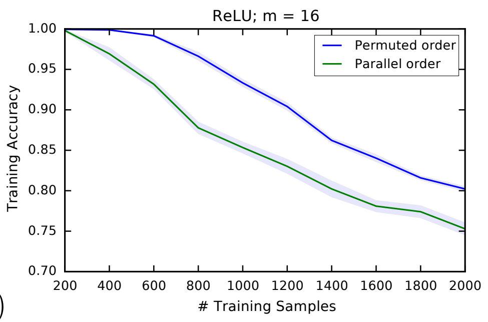
(b) 

Figure 6.1: **Fitting two random tasks.** (a) The dotted lines show that permuted ordering fits n samples as well as parallel fits $n/2$ for linear networks; (b) For ReLU networks, permuted ordering enjoys a similar advantage. Thus, permuted ordering of shared layers eases integration of information across disparate tasks.

single-task case. A similar gap in performance is seen in the nonlinear case, indicating that this behavior extends to more powerful models. Thus, the learned permuted layers are able to successfully adapt to their different orderings in different tasks.

Looking at conditions that make this result possible can shed further light on this behavior. For instance, consider T tasks $t_1, \ldots, t_T$, with input and output size both m, and optimal linear solutions $F_1, \ldots, F_T$, respectively. Let $F_1, \ldots, F_T$ be $m \times m$ matrices, and suppose there exist matrices $G_1, \ldots, G_T$ such that $F_i = G_i G_{(i+1 \bmod T)} \ldots G_{(i-1 \bmod T)}\ \forall\ i$. Then, because the matrix trace is invariant under cyclic permutations, the constraint arises that

$$\mathrm{tr}(F_1) = \mathrm{tr}(F_2) = \ldots = \mathrm{tr}(F_T). \tag{6.5}$$

In the case of random matrices induced by the random tasks above, the traces of the F_i are all equal in expectation and concentrate well as their dimensionality increases. So, the restrictive effect of Eq. 6.5 on the expressivity of permuted ordering here is negligible.

Adding a small number of task-specific scaling parameters. Of course, real world tasks are generally much more structured than random ones, so such reliable expressivity of permuted ordering might not always be expected. However, adding a small number of task-specific scaling parameters can help adapt learned layers to particular tasks. This observation has been previously exploited in the parallel ordering setting, for learning task-specific batch normalization scaling parameters (Bilen & Vedaldi, 2017) and controlling communication between columns (Misra et al., 2016). Similarly, in the permuted ordering setting, the constraint induced by Eq. 6.5 can be reduced by adding task-specific scalars $\{s_i\}_{i=2}^{T}$ such that $F_i = s_i G_i G_{(i+1 \bmod T)} \dots G_{(i-1 \bmod T)}$, and $s_1 = 1$. The constraint given by Eq. 6.5 then reduces to

$$\mathrm{tr}(F_i/s_i) = \mathrm{tr}(F_{i+1}/s_{i+1}) \;\forall\; 1 \le i < T \implies s_{i+1} = s_i(\mathrm{tr}(F_{i+1})/\mathrm{tr}(F_i)), \qquad (6.6)$$

which are defined when $\mathrm{tr}(F_i) \neq 0 \;\forall\; i < T$. Importantly, the number of task-specific parameters does not depend on m, which is useful for scalability as well as encouraging maximal sharing between tasks. The idea of using a small number of task-specific scaling parameters is incorporated in the soft ordering approach introduced in the next section.

6.3.3 Soft ordering of shared layers

Permuted ordering tests the parallel ordering assumption, but still fixes an *a priori* layer ordering for each task before training. Here, a more flexible

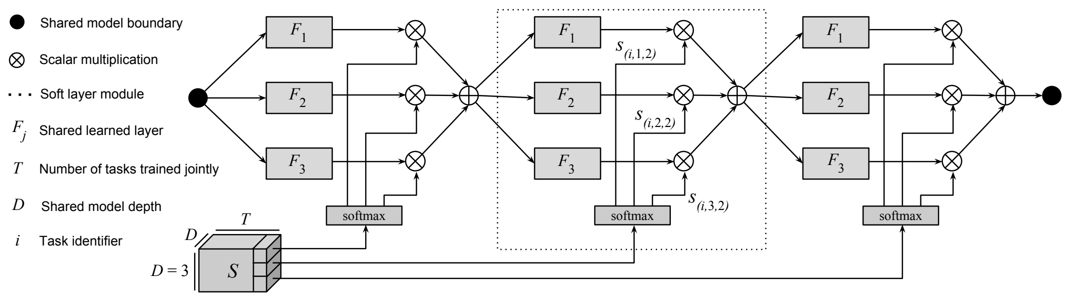


Figure 6.2: **Soft ordering of shared layers.** Sample soft ordering network with three shared layers. Soft ordering (Eq. 6.7) generalizes Eqs. 6.2 and 6.3, by learning a tensor S of task-specific scaling parameters. S is learned jointly with the F_j, to allow flexible sharing across tasks and depths. The F_j in this figure each include a shared weight layer and any nonlinearity. This architecture enables the learning of layers that are used in different ways at different depths for different tasks.

soft ordering approach is introduced, which allows jointly trained models to learn *how* layers are applied while simultaneously learning the layers themselves. Consider again a core network of depth D with layers $W_1, \dots, W_D$ learned and shared across tasks. The soft ordering model for task t_i is defined as follows:

$$y_i^k = \sum_{j=1}^{D} s_{(i,j,k)}(\phi_k[W_j(y_i^{k-1})]), \text{ with } \sum_{j=1}^{D} s_{(i,j,k)} = 1 \;\forall\; (i,k), \tag{6.7}$$

where $y_i^0 = \mathcal{E}_i(x_i)$, $y_i = \mathcal{D}_i(y_i^D)$, and each $s_{(i,j,k)}$ is drawn from S: a tensor of learned scales for each task t_i for each layer W_j at each depth k. Figure 6.2 shows an example of a resulting depth three model. Motivated by Section 6.3.2 and previous work (Misra et al., 2016), S adds only D^2 scaling parameters per task, which is notably not a function of the size of any W_j. The constraint that all $s_{(i,j,k)}$ sum to 1 for any (i,k) is implemented via softmax, and emphasizes the idea that a soft *ordering* is what is being learned; in particular, this formulation subsumes any fixed layer ordering ρ_i by $s_{(i,\rho_i(k),k)} = 1 \ \forall \ (i, k)$. S can be learned jointly with the other learnable parameters in the W_k, $\mathcal{E}_i$, and $\mathcal{D}_i$ via backpropagation. In training, all $s_{(i,j,k)}$ are initialized with equal values, to reduce initial bias of layer function across tasks. It is also helpful to apply dropout after each shared layer. Aside from its usual benefits (Srivastava et al., 2014), dropout has been shown to be useful in increasing the generalization capacity of shared representations (Devin et al., 2016). Since the trained layers in Eq. 6.7 are used for different tasks and in different locations, dropout makes them more robust to supporting different functionalities. These ideas are tested empirically on the MNIST, UCI, Omniglot, and CelebA data sets in the next section.

6.4 Empirical Evaluation of Soft Layer Ordering

These experiments evaluate soft ordering against fixed ordering MTL and single-task learning. The first experiment applies them to intuitively related MNIST tasks, the second to "superficially unrelated" UCI tasks, the third to the real-world problem of Omniglot character recognition, and the fourth to large-scale facial attribute recognition. In each experiment, single task, parallel ordering (Eq. 6.2), permuted ordering (Eq. 6.3), and soft ordering (Eq. 6.7) train an equivalent set of core layers. In permuted ordering, the order of layers were randomly generated for each task each trial.

6.4.1 Disentangling related tasks: MNIST digit$_1$-vs.-digit$_2$ binary classification

This experiment evaluates the ability of multitask methods to exploit tasks that are intuitively related, but have disparate input representations. Binary classification problems derived from the MNIST hand-written digit dataset are a common test bed for evaluating deep learning methods that require multiple tasks (Fernando et al., 2017; Kirkpatrick et al., 2017; Yang & Hospedales, 2017). Here, the goal of each task is to distinguish between two distinct randomly selected digits. To create initial dissimilarity across tasks that multitask models must disentangle, each $\mathcal{E}_i$ is a random frozen fully-connected ReLU layer with output size 64. There are four core layers, each a fully-connected ReLU layer with 64 units. Each $\mathcal{D}_i$ is an unshared dense layer with a single sigmoid classification output.

Results are shown in Figure 6.3. The relative performance of permuted ordering and soft ordering compared to parallel ordering increases with the number of tasks trained jointly (Figure 6.3a), showing how flexibility of order can help in scaling to more tasks. This result is consistent with the hypothesis that parallel ordering has increased negative effects as the number of tasks increases. Figure 6.3b-d show what soft ordering actually learns: The scalings for tasks diverge as layers specialize to different functions for different tasks.

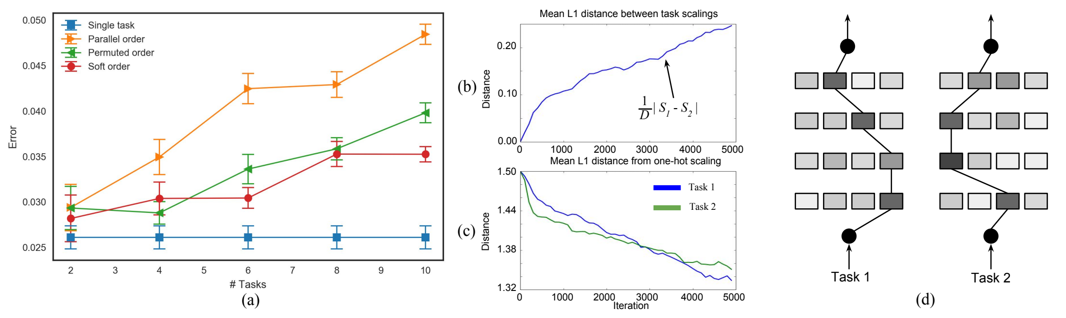


Figure 6.3: **MNIST results.** (a) Relative performance of permuted and soft ordering compared to parallel ordering improves as the number of tasks increases, showing how flexibility of order can help in scaling to more tasks. Note that cost savings of multitask over single task models in terms of number of trainable parameters scales linearly with the number of tasks. For a representative two-task soft order experiment (b) the layer-wise distance between scalings of the tasks increases by iteration, and (c) the scalings move towards a hard ordering. (d) The final learned relative scale of each shared layer at each depth for each task is indicated by shading, with the strongest path drawn, showing that a distinct soft order is learned for each task (• marks the shared model boundary).

Dataset	Input Features	Output classes	Samples
Australian credit	14	2	690
Breast cancer	30	2	569
Ecoli	7	8	336
German credit	24	2	1000
Heart disease	13	5	303
Hepatitis	19	2	155
Iris	4	3	150
Pima diabetes	8	2	768
Wine	13	3	178
Yeast	8	10	1484

Table 6.1: **UCI data set descriptions.** The ten UCI tasks used in joint training; the varying types of problems and dataset characteristics show the diversity of this set of tasks.

6.4.2 Superficially Unrelated Tasks: Joint Training of Ten Popular UCI Datasets

The next experiment evaluates the ability of soft ordering to integrate information across a diverse set of "superficially unrelated" tasks (Mahmud & Ray, 2008), i.e., tasks with no immediate intuition for how they may be related. Ten tasks are taken from some of most popular UCI classification data sets (Lichman, 2013). Descriptions of these tasks are given in Figure 6.1. Inputs and outputs have no a priori shared meaning across tasks. Each $\mathcal{E}_i$ is a learned fully-connected ReLU layer with output size 32. There are four core layers, each a fully-connected ReLU layer with 32 units. Each $\mathcal{D}_i$ is an unshared dense softmax layer for the given number of classes. The results in Figure 6.4 show that, while parallel and permuted show no improvement in error after the first 1000 iterations, soft ordering significantly outperforms the

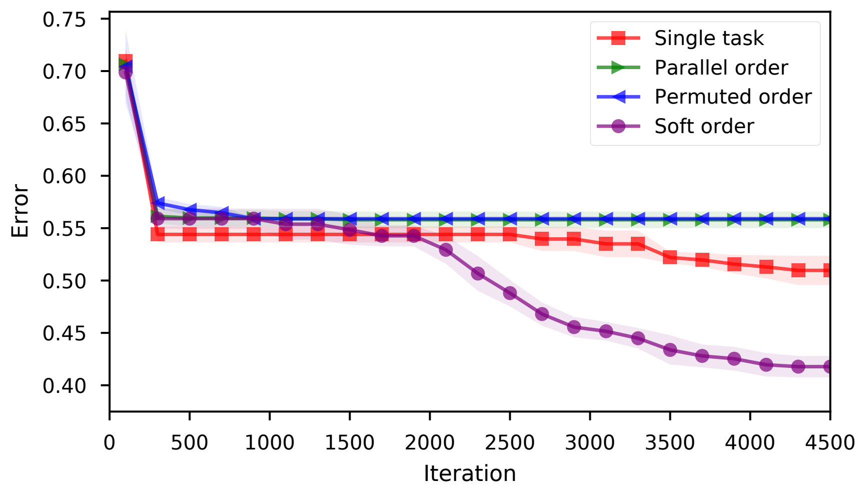


Figure 6.4: **UCI results.** Mean test error over all ten tasks by iteration. Permuted and parallel order show no improvement after the first 1000 iterations, while soft order decisively outperforms the other methods.

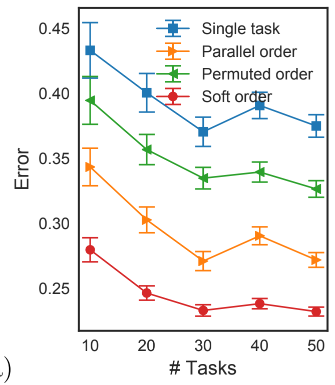
other methods. With this flexible layer ordering, the model is eventually able to exploit significant regularities underlying these seemingly disparate domains.

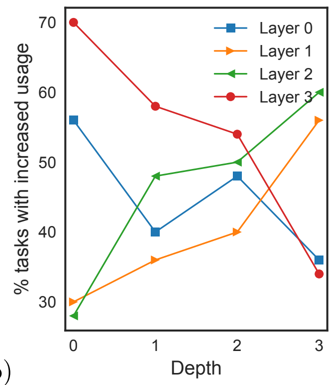
6.4.3 Extension to Convolutions: Multi-alphabet Character Recognition

The Omniglot dataset (Lake et al., 2015) consists of fifty alphabets, each of which induces a different character recognition task. Deep MTL approaches have recently shown promise on this dataset (Yang & Hospedales, 2017). It is a useful benchmark for MTL because the large number of tasks allows analysis of performance as a function of the number of tasks trained jointly, and there is clear intuition for how knowledge of some alphabets will increase the ability to learn others. Omniglot is also a good setting for evaluating the ability of soft ordering to learn how to compose layers in different ways for different tasks: it was developed as a problem with inherent composability, e.g., similar kinds of strokes are applied in different ways to draw characters from different alphabets (Lake et al., 2015). Consequently, it has been used as a test bed for deep generative models (Rezende et al., 2016). To evaluate performance for a given number of tasks T, a single random ordering of tasks was created, from which the first T tasks are considered. Train/test splits are created in the same way as previous work (Yang & Hospedales, 2017), using 10% or 20% of data for testing.

This experiment is a scale-up of the previous experiments in that it evaluates soft ordering of convolutional layers. The models are made as close as possible in architecture to previous work (Yang & Hospedales, 2017), while allowing soft ordering to be applied. There are four core layers, each convolutional followed by max pooling. $\mathcal{E}_i(x_i) = x_i \ \forall \ i$, and each $\mathcal{D}_i$ is a fully-connected softmax layer with output size equal to the number of classes. The results show that soft ordering is able to consistently outperform other deep MTL approaches (Figure 6.5). The improvements are robust to the number of tasks (Figure 6.5a) and the amount of training data (Figure 6.5c), suggesting that soft ordering, not task complexity or model complexity, is responsible for the improvement.

Permuted ordering performs significantly worse than parallel ordering in this domain. This is not surprising, as deep vision systems are known to induce a common feature hierarchy, especially within the first couple of layers (Lee et al., 2008; Lecun et al., 2015). Parallel ordering has this hierarchy built

(a) 

(b) 

(c)

Deep MTL method	10% Test Split	20% Test Split
STL	34.36 (± 0.53)	35.92 (± 0.74)
UD-MTL	29.98 (± 1.33)	29.53 (± 0.99)
DMTRL-LAF	31.08 (± 0.65)	33.37 (± 0.97)
DMTRL-Tucker	29.67 (± 1.25)	31.11 (± 1.16)
DMTRL-TT	28.78 (± 0.61)	30.61 (± 0.65)
Single task (ours)	38.49 (± 0.87)	38.10 (± 0.88)
Parallel order	27.17 (± 0.57)	28.24 (± 0.67)
Permuted order	32.64 (± 0.64)	33.18 (± 0.74)
Soft order	**23.19** (± 0.34)	**24.11** (± 0.48)

Figure 6.5: **Omniglot results.** (a) Error by number of tasks trained jointly. Soft ordering significantly outperforms single task and both fixed ordering approaches for each number of tasks; (b) Distribution of learned layer usage by depth across all 50 tasks for a soft order run. The usage of each layer is correlated (or inversely correlated) with depth. This coincides with the understanding that there is some innate hierarchy in convolutional networks, which soft ordering is able to discover. For instance, the usage of Layer 3 decreases as the depth increases, suggesting that its primary purpose is low-level feature extraction, though it is still sees substantial use in deeper contexts; (c) Errors with all 50 tasks for different training set sizes. The first five methods are previous deep MTL results (Yang & Hospedales, 2017), which use multitask tensor factorization methods in a shared parallel ordering. Soft ordering significantly outperforms the other approaches, showing the approach scales to real-world tasks requiring specialized components such as convolutional layers.

in; for permuted ordering it is more difficult to exploit. However, the existence of this feature hierarchy does not preclude the possibility that the functions (i.e., layers) used to produce the hierarchy may be useful in other contexts. Soft ordering allows the discovery of such uses. Figure 6.5b shows how each layer is used more or less at different depths. The soft ordering model learns a "soft hierarchy" of layers, in which each layer has a distribution of increased or decreased usage at each depth. In this case, the usage of each layer is correlated (or inversely correlated) with depth. For instance, the usage of Layer 3 decreases as the depth increases, suggesting that its primary purpose is low-level feature extraction, though it is still sees substantial use in deeper contexts. Section 6.5 describes an experiment that further investigates the behavior of a single layer in different contexts.

6.4.4 Large-scale Application: Facial Attribute Recognition

Although facial attributes are all high-level concepts, they do not intuitively exist at the same level of a shared hierarchy (even one that is learned; Lu et al., 2017). Rather, these concepts are related in multiple subtle and overlapping ways in semantic space. This experiment investigates how a soft ordering approach, as a component in a larger system, can exploit these relationships.

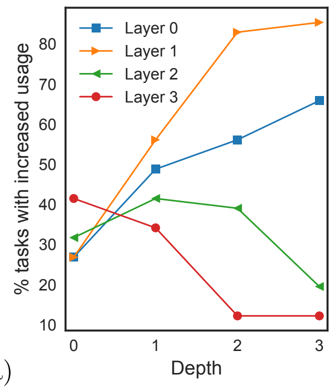
The CelebA dataset consists of ≈200K 178×218 color images, each with binary labels for 40 facial attributes (Liu et al., 2015b). In this experiment, each label defines a task, and parallel and soft order models are based on a ResNet-50 vision model (He et al., 2016), which has also been used in recent state-of-the-art approaches to CelebA (Günther et al., 2017; He et al., 2017). Let $\mathcal{E}_i$ be a ResNet-50 model truncated to the final average pooling layer, followed by a linear layer projecting the embedding to size 256. $\mathcal{E}_i$ is shared across all tasks. There are four core layers, each a dense ReLU layer with 256 units. Each $\mathcal{D}_i$ is an unshared dense sigmoid layer. Parallel ordering and soft ordering models were compared. To further test the robustness of learning, models were trained with and without the inclusion of an additional facial landmark detection regression task. Soft order models were also tested with and without the inclusion of a fixed identity layer at each depth. The identity layer can increase consistency of representation across contexts, which can ease learning of each layer, while also allowing soft ordering to tune how much total non-identity transformation to use for each individual task. This is especially relevant for the case of attributes, since different tasks can have different levels of complexity and abstraction.

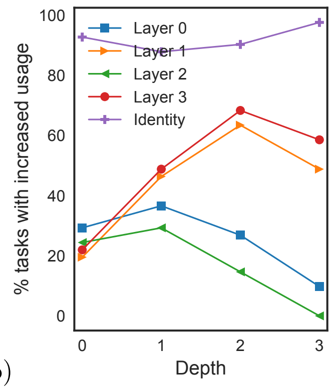
The results are given in Figure 6.6c. Existing work that used a ResNet-50 vision model showed that using a parallel order multitask model improved test error over single-task learning from 10.37 to 9.58 (He et al., 2017). With our faster training strategy and the added core layers, our parallel ordering model achieves a test error of 10.21. The soft ordering model yielded a substantial improvement beyond this to 8.79, demonstrating that soft ordering can add value to a larger deep learning system. Including landmark detection yielded a marginal improvement to 8.75, while for parallel ordering it degraded performance slightly, indicating that soft ordering is more robust to joint training of diverse kinds of tasks. Including the identity layer improved performance to 8.64, though with both the landmark detection and the identity layer this improvement was slightly diminished. One explanation for this degradation is that the added flexibility provided by the identity layer offsets the regularization provided by landmark detection. Note that previous work has shown that adaptive weighting of task loss (He et al., 2017; Rudd et al., 2016), data augmentation and ensembling (Günther et al., 2017), and a larger underlying vision model (Lu et al., 2017) each can also yield significant improvements. Aside from soft ordering, none of these improvements alter the *multitask topology*, so their benefits are expected to be complementary to that of soft ordering demonstrated in this experiment. By coupling them with soft ordering, greater improvements should be possible.

Figures 6.6a-b characterize the usage of each layer learned by soft order models. Like in the case of Omniglot, layers that are used less at lower depths are used more at higher depths, and vice versa, giving further evidence that the models learn a "soft hierarchy" of layer usage. When the identity layer is included, its usage is almost always increased through training, as it allows the model to use smaller specialized proportions of nonlinear structure for each individual task.

6.5 Visualizing the Behavior of Soft Ordering Layers

The success of soft layer ordering suggests that layers learn functional primitives with similar effects in different contexts. To explore this idea

(a) 

(b) 

(c)

Deep MTL method	Test Error %
Single Task (He et al., 2017)	10.37
MTL Baseline (He et al., 2017)	9.58
Parallel Order	10.21
Parallel Order + Landmarks	10.29
Soft Order	8.79
Soft Order + Landmarks	8.75
Soft Order + Identity	**8.64**
Soft Order + Landmarks + Identity	8.68

Figure 6.6: **CelebA results.** Layer usage by depth (a) without and (b) with inclusion of the identity layer. In both cases, layers with lower usage at lower depths have higher usage at higher depths, and vice versa. The identity layer almost always sees increased usage; its application can increase consistency of representation across contexts. (c) Soft order models achieve a significant improvement over parallel ordering, and receive a boost from including the identity layer. The first two rows are previous work with ResNet-50 that show their baseline improvement from single task to multitask.

qualitatively, the following experiment uses generative visual tasks. The goal of each task is to learn a function $(x, y) \to v$, where (x, y) is a pixel coordinate and v is a brightness value, all normalized to $[0, 1]$. Each task is defined by a single image of a "4" drawn from the MNIST dataset; all of its pixels are used as training data. Ten tasks are trained using soft ordering with four shared dense ReLU layers of 100 units each. $\mathcal{E}_i$ is a linear encoder that is shared across tasks, and $\mathcal{D}_i$ is a global average pooling decoder. Thus, task models are distinguished completely by their learned soft ordering scaling parameters s_t. To visualize the behavior of layer l at depth d for task t, the predicted image for task t is generated across varying magnitudes of $s_{(t,l,d)}$. The results for the first two tasks and the first layer are shown in Table 6.2. Similar function is observed in each of the six contexts, suggesting that the layers indeed learn functional primitives.

6.6 Discussion and Future Work

In the interest of clarity, the soft ordering approach in this chapter was developed as a relatively small step away from the parallel ordering assumption. To develop more practical and specialized methods, inspiration can be taken from recurrent architectures, the approach can be extended to layers of more general structure, and applied to training and understanding general functional building blocks.

Connections to recurrent architectures. Eq. 6.7 is defined recursively with respect to the learned layers shared across tasks. Thus, the

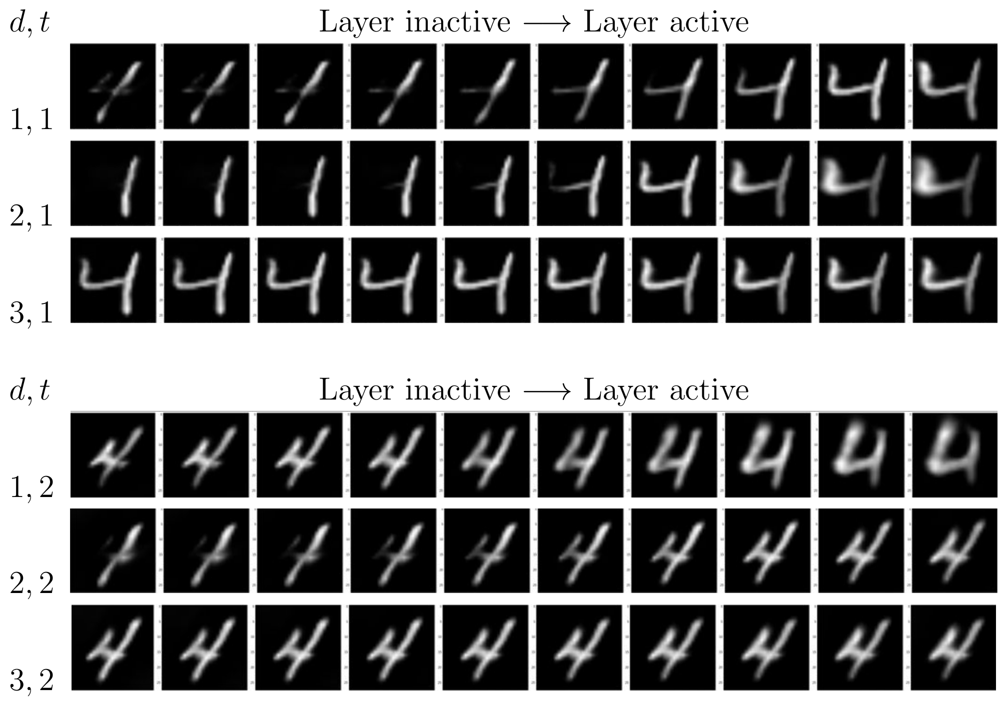

d, t
Layer inactive $\longrightarrow$ Layer active
1, 1
2, 1
3, 1
d, t
Layer inactive $\longrightarrow$ Layer active
1, 2
2, 2
3, 2

Table 6.2: **Example behavior of a soft order layer**. For each task t, and at each depth d, the effect of increasing the activation of of this particular layer is to expand the left side of the "4" in a manner appropriate to the functional context (e.g., the magnitude of the effect decreases with depth). Results for other layers are similar, suggesting that the layers implement functional primitives.

soft-ordering architecture can be viewed as a new type of recurrent architecture designed specifically for MTL. From this perspective, Figure 6.2 shows an unrolling of a *soft layer module*: Different scaling parameters are applied at different depths when unrolled for different tasks. Since the type of recurrence induced by soft ordering does not require task input or output to be sequential, methods that use recurrence in such a setting are of particular interest (Liang & Hu, 2015; Liao & Poggio, 2016; Pinheiro & Collobert, 2014; Socher et al., 2011; Zamir et al., 2016). Recurrent methods can also be used to reduce the size of S below $O(TD^2)$, e.g., via recurrent hypernetworks (Ha et al., 2017). Finally, Section 6.4 demonstrated soft ordering where shared learned layers were fully-connected or convolutional; it is also straightforward to extend soft ordering to shared layers with internal recurrence, such as LSTMs (Hochreiter & Schmidhuber, 1997). In this setting, soft ordering can be viewed as inducing a higher-level recurrence.

Generalizing the structure of shared layers. For clarity, in this chapter all core layers in a given setup had the same shape. Of course, it would be useful to have a generalization of soft ordering that could subsume any modern deep architecture with many layers of varying structure. As given by Eq. 6.7, soft ordering requires the same shape inputs to the element-wise sum at each depth. Reshapes and/or resampling can be added as adapters between tensors of different shape; alternatively, a function other than a sum could be used. For example, instead of learning a weighting across layers at each depth, a probability of applying each module could be learned in a manner similar to adaptive dropout (Ba & Frey, 2013; Li et al., 2016) or a sparsely-gated mixture of experts (Shazeer et al., 2017). Furthermore, the idea of a soft ordering of layers can be extended to soft ordering over modules with more general structure, which may more succinctly capture recurring modularity. One such approach to more general module structure is presented in Chapter 7.

Training generalizable building blocks. Because they are used in different ways at different locations for different tasks, the shared trained layers in permuted and soft ordering have learned more general functionality than layers trained in a fixed location or for a single task. A natural hypothesis is that they are then more likely to generalize to future unseen tasks, perhaps even without further training. This ability would be especially useful in the small data regime, where the number of trainable parameters should be limited. For example, given a collection of these layers trained on a previous set of tasks, a model for a new task could learn how to apply these building blocks, e.g., by learning a soft order, while keeping their internal parameters fixed. Learning an efficient set of such generalizable layers would then be akin to learning a set of *functional primitives*. Such functional modularity and repetition is evident in the natural, technological and sociological worlds, so such a set of functional primitives may align well with complex real-world models. This perspective is related to recent work in reusing modules in the parallel ordering setting (Fernando et al., 2017). The different ways in which different tasks learn to use the same set of modules can also help shed light on how tasks are related, especially those that seem superficially disparate (e.g., by extending the analysis performed for Figure 6.3d), thus assisting in the discovery of real-world regularities. Further methods for training generalizable building blocks are considered in the next two chapters.

6.7 Conclusion

This chapter extended the idea of PTA (Chapter 5) to joint training of each module (here, layers) at multiple depths in each task model, to give them another level of generality. This extension was motivated by *parallel ordering* of shared layers, which was identified as a common assumption underlying existing deep MTL approaches. This assumption restricts the kinds of shared structure that can be learned between tasks. Experiments demonstrate how direct approaches to removing this assumption can ease the integration of information across plentiful and diverse tasks. *Soft ordering* is introduced as a method for learning how to apply layers in different ways at different depths for different tasks, while simultaneously learning the layers themselves. The resulting multitask models are constructed by applying mixtures of a set of core layers at each pseudo-task location. Soft ordering is shown to outperform parallel ordering methods as well as single-task learning across a suite of domains. These results show that deep MTL can be improved while generating a compact set of multipurpose functional primitives, thus aligning more closely with our understanding of complex real-world processes. However, vanilla soft ordering does not scale well, since each layer is executed at each depth. The next two chapters automatically select which module to use at each location, in order to scale this idea to more practical deep learning systems: Chapter 7 scales it to complex automatically-discovered topologies, and Chapter 8 scales it to modules that can be shared across diverse kinds of architectures and task modalities.

Chapter 7

Multitask Architecture Search

This chapter[1] improves the scalability of soft ordering, and the generality of the learned modules, by automatically designing a multitask architecture assembled from modules, while simultaneously learning the internal parameters of the modules themselves. Designing deep neural network architectures for multitask learning is a challenge: There are many ways to tie the tasks together, and the design choices matter. The size and complexity of this problem exceeds human design ability, making it a compelling domain for evolutionary optimization. Using the soft ordering approach from Chapter 6 as the starting point, this chapter develops a method for learning sets of modules that generalize across tasks while simultaneously discovering topologically distinct ways to assemble these modules to solve each task. The approach is based on complexifying task-specific topologies, by incrementally adding new locations to apply modules. Evolution and deep training are performed on a single GPU by jointly training all candidate topologies for all tasks. This approach is then extended by using evolution to design the topological structure of the modules

[1]The content of this chapter was previously presented at GECCO (Liang, Meyerson, and Miikkulainen, 2018). Jason Liang worked on experimental design, implementation, and analysis, particularly with respect to all experiments involving CoDeepNEAT (Miikkulainen et al., 2017). Risto Miikkulainen provided guidance and feedback through discussions.

themselves. The methods significantly improve upon previous results in the Omniglot multitask, multialphabet character recognition domain. This result demonstrates how evolution can play a key role in advancing the design of deep neural networks and complex systems in general. In particular, it shows deep multitask learning can be improved by optimizing architectures explicitly for the goal of discovering and sharing generic modules.

7.1 Introduction

Much of the research in deep learning in recent years has focused on coming up with better architectures, and MTL is no exception. As a matter of fact, architecture plays possibly an even larger role in MTL because there are many ways to tie the multiple tasks together. The best network architectures are large and complex, and have become very hard for human designers to optimize (Szegedy et al., 2015, 2016; Jaderberg et al., 2017a; Zoph & Le, 2017). This chapter develops an automated, flexible approach for evolving architectures, i.e. hyperparameters, modules, and module routing topologies, of deep multitask networks.

The architecture presented in Chapter 6 is used as a starting point, in which distinct soft sequences of layers are learned by gradient descent for each task. This chapter extends this architecture in several ways. First, a novel algorithm for evolving *task specific routings* that create an unique routing between modules for each task is proposed. From the perspective of pseudo-tasks, evolution precedes by incrementally adding pseudo-task locations to each task

Algorithm 7.1 CTR (Sec. 7.2)

```
Given set of modules
Initialize topology population for each task
Randomly initialize all weights
Each meta-iteration:
    Assemble networks
    Jointly train all networks with backprop
    Assign fitnesses to topologies
    Update topology populations
```

Algorithm 7.2 CMTR (Sec. 7.3)

```
Initialize module population
Each generation:
    Assemble sets of modules
    Train sets of modules with CTR
    Assign fitnesses to modules
    Update module populations
```

Figure 7.1: High-level algorithm outlines of the two algorithms introduced in this Chapter. CTR evolves custom routings for each task on a single GPU by interleaving evolution and backpropagation; CMTR coevolves the architectures of the modules themselves, evaluating them in parallel by training with CTR in an inner loop.

model, where solving these pseudo-tasks leads to improvements for the underlying task. This approach is then extended by evolving the architectures of the modules themselves in an outer loop, taking advantage of concurrent work that developed a multitask version of the CoDeepNEAT algorithm (Miikkulainen et al., 2017). High-level descriptions of these algorithms are given in Figure 7.1.

These approaches are evaluated in the Omniglot task (Lake et al., 2015) of learning to recognize characters from many different alphabets. Evolution of task-specific topologies significantly improves upon soft ordering, and coevolution of modules and topologies together improves performance even further. The results thus demonstrate three general points: evolutionary architecture search can make a large difference in performance of deep learning networks;

MTL can improve performance of deep learning tasks; and putting these together results in a particularly powerful approach. In the future it can be applied to various problems in vision, language, and control, and in particular to domains with multimodal inputs and outputs (e.g., see Chapter 8).

The rest of this chapter is organized as follows: In Sections 7.2 and 7.3, the key contribution of this chapter, novel evolutionary algorithms for architecture search of multitask networks are described. Finally, in Section 7.4 and Section 7.5 experimental results on the Omniglot domain are presented and analyzed.

7.2 Coevolution of Task-Specific Routings (CTR)

This section introduces Coevolution of Task Routing (CTR), a multitask architecture search approach that takes advantage of the dynamics of soft ordering by evolving task-specific topologies instead of using a single fixed architecture for all tasks. The algorithm learns a set of shared generic modules while simultaneously evolving task-specific routings for each task. The algorithm begins with the soft module mixing operation, or *soft merge*, used to construct soft ordering networks in Chapter 6. A soft merge is a learnable function given by

$$\text{softmerge}(\text{in}_1, \ldots, \text{in}_M) = \sum_{m=1..M} s_m \text{in}_m, \text{ with } \sum_{m=1..M} s_m = 1\,, \tag{7.1}$$

where the in_m are a list of incoming tensors, s_m are scalars trained simultaneously with internal layer weights via backpropagation, and the constraint that all s_m sum to 1 is enforced via a softmax function.

7.2.1 Algorithm Overview

Like in soft ordering, in CTR there are K modules whose weights are shared everywhere they are used across all tasks. CTR searches for a distinct module routing scheme *for each task*, and trains a *single set of modules* throughout evolution. Having a distinct routing scheme for each task makes sense if the shared modules are seen as a set of building blocks that are assembled to meet the differing demands of different problems. Training a single set of modules throughout evolution then makes sense as well: As modules are trained in different locations for different purposes during evolution, their functionality should become increasingly general, and it should thus become easier for them to adapt to the needs of a new location. Such training is efficient since the core structure of the network need not be retrained from scratch at every generation. In other words, CTR incurs no additional iterations of backpropagation over training a single fixed-topology multitask model. Because of this feature, CTR is related to PathNet (Fernando et al., 2017), which evolves pathways through modules as those modules are being trained. However, unlike in PathNet, in CTR distinct routing schemes are coevolved across tasks, modules can be applied in any location, and module usage is adapted via the soft merge operation.

CTR operates a variant of a $(1+1)$ evolutionary algorithm ($(1+1)$-EA) for each task. Running separate EAs in parallel for each task is possible because an evaluation of a multitask network yields a performance metric *for each task*. The $(1+1)$-EA is chosen because it is efficient and sufficiently powerful in experiments, though it can potentially be replaced by any population-based method. To make it clear that a single set of modules is trained during evolution, and to disambiguate from the terminology of CoDeepNEAT, which is used in the outer loop in the next section, for CTR the term *meta-iteration* is used in place of *generation*.

7.2.2 Representation

Each individual constitutes a module routing scheme for a particular task. At any point in evolution, the ith individual for the tth task is represented by a tuple $(\mathcal{E}_{ti}, G_{ti}, \mathcal{D}_{ti})$, where $\mathcal{E}_{ti}$ is an encoder, G_{ti} is a DAG, which specifies the module routing scheme, and $\mathcal{D}_{ti}$ is a decoder. The complete model for an individual is then given by

$$\boldsymbol{y}_t = \big(\mathcal{D}_{ti} \circ \mathcal{R}\big(G_{ti}, \{f_k\}_{k=1}^{K}\big) \circ \mathcal{E}_{ti}\big)(\boldsymbol{x}_t)\,, \tag{7.2}$$

where $\mathcal{R}$ is the canonical routing function, which contains any learned soft merge parameters, and indicates the execution of the computational graph of shared modules f_k based on the DAG G_{ti}. Note that $\mathcal{E}_{ti}$, and $\mathcal{D}_{ti}$ can be any neural network functions that are compatible with the set of shared modules. In the experiments in this chapter, each $\mathcal{E}_{ti}$ is an identity transformation layer, and each $\mathcal{D}_{ti}$ is a fully connected classification layer.

G_{ti} is a DAG, whose single source node represents the input layer for that task, and whose single sink node represents the output layer, e.g., a classification layer. All other nodes either point to a module f_k to be applied at that location, or a parameterless adapter layer that ensures adjacent modules are technically compatible. In the experiments in this chapter, all adapters are 2×2 max-pooling layers. Whenever a node of G_{ti} has multiple incoming edges, their contents are combined in a learned soft merge (Eq. 7.1).

7.2.3 Initialization

The algorithm begins by initializing the shared modules $\{f_k\}_{k=1}^K$ with random weights. Then, each champion $(\mathcal{E}_{t1}, G_{t1}, \mathcal{D}_{t1})$ is initialized, with $\mathcal{E}_{t1}$ and $\mathcal{D}_{t1}$ initialized with random weights, and G_{t1} according to some graph initialization policy. For example, the initialization of G_{t1} can be minimal or random. In the experiments in this chapter, G_{t1} is initialized to reflect the classical deep multitask learning approach, i.e., the graph is given by

$$\mathcal{E}_{t1} \to f_1 \to f_2 \to \ldots \to f_K \to \mathcal{D}_{t1} \tag{7.3}$$

with adapters added as needed.

7.2.4 Variation

At the start of each meta-iteration, a challenger $(\mathcal{E}_{t2}, G_{t2}, \mathcal{D}_{t2})$ is generated by mutating the tth champion as follows (the insertion of adapters is omitted for clarity):

1. The challenger starts as a copy of the champion, including learned weights, i.e., $(\mathcal{E}_{t2}, G_{t2}, \mathcal{D}_{t2}) := (\mathcal{E}_{t1}, G_{t1}, \mathcal{D}_{t1})$.

2. A pair of nodes (u, v) is randomly selected from G_{t2} such that v is an ancestor of u.
3. A module $\mathcal{M}_k$ is randomly selected from $\{f_k\}_{k=1}^{K}$.
4. A new node w is added to G_{t2} with f_k as its function.
5. New edges (u, w) and (w, v) are added to G_{t2}.
6. The scalar weight of (w, v) is set such that its value after the softmax is some $\alpha \in (0, 1)$. To initially preserve champion behavior, α is set to be small. I.e., if $s_1, \ldots, s_m$ are the scales of the existing inbound edges to v, s_{m+1} is the initial scale of the new edge, and $s_{\max} = \max(s_1, \ldots, s_m)$ then
$$s_{m+1} = \ln\big(\frac{\alpha}{1-\alpha} \sum_{j=1..m} e^{s_j - s_{\max}}\big) + s_{\max} \,. \tag{7.4}$$

7.2.5 Training

After challengers are generated, all champions and challengers are trained jointly for M iterations with a gradient-based optimizer. Note that the scales of G_{t1} and G_{t2} diverge during training, as do the weights of $\mathcal{D}_{t1}$ and $\mathcal{D}_{t2}$. After training, all champions and challengers are evaluated on a validation set that is disjoint from the training data. The fitness for each individual is its performance for its task on the validation set. In this chapter, accuracy is the performance metric. If the challenger has higher fitness than the champion, then the champion is replaced, i.e.,$(\mathcal{E}_{t1}, G_{t1}, \mathcal{D}_{t1}) := (\mathcal{E}_{t2}, G_{t2}, \mathcal{D}_{t2})$.

After selection, if the average accuracy across all champions is the best achieved so far, the entire system is checkpointed, including the states of the modules. After evolution, the champions and modules from the last checkpoint constitute the final trained model, and are evaluated on a held out test set.

7.2.6 An Ecological Perspective

More than most evolutionary methods, this algorithm reflects an artificial ecology. The shared modules can be viewed as a shared finite set of environmental resources that is constantly exploited and altered by the actions of different tasks, which can correspond to different species in an environment. Within each task, individuals compete and cooperate to develop mutualistic relationships with the other tasks via their interaction with this shared environment. A visualization of CTR under this perspective is shown in Figure 7.2. Importantly, even if a challenger does not outperform its champion, its developmental (learning) process still affects the shared resources. This perspective suggests a more optimistic view of evolution, in which individuals can have substantial positive effects on the future of the ecosystem even without reproducing.

7.3 Coevolution of Modules and Task Routing (CMTR)

Concurrently to this work, the soft ordering approach has been extended to evolve the module architectures themselves using CoDeepNEAT (Miikkulainen et al., 2017). Though it also considers evolving multitask topologies,

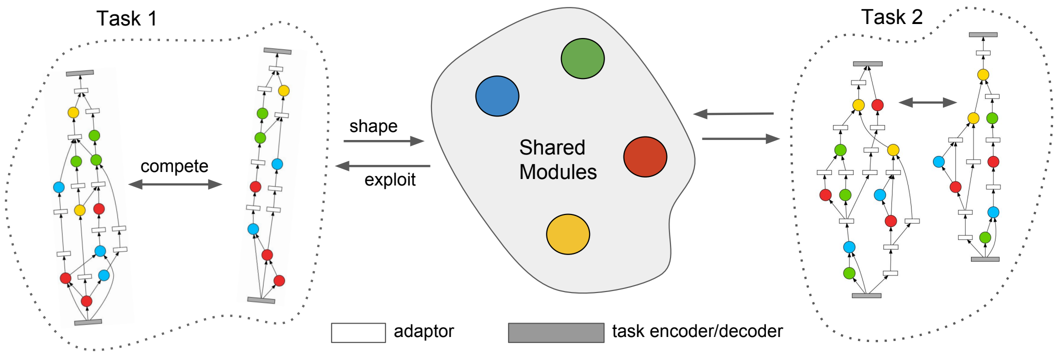


Figure 7.2: This figure shows an instance of CTR with three tasks and four modules that are shared across all tasks. Each individual assembles the modules in different ways. Through gradient-based training, individuals exploit the shared resources to compete within a task, and over time must develop mutualistic relationships with other tasks via their use of the shared modules.

the improvements achieved with that approach are largely orthogonal to those of CTR, and they can be combined to form an even more powerful algorithm called Coevolution of Modules and Task Routing (CMTR). Since evolution in CTR occurs during training and is highly computational efficient, it is feasible to use CoDeepNEAT as an outer evolutionary loop to evolve modules. To evaluate and assign fitness to the modules, they are passed on to CTR (the inner evolutionary loop) for evolving and assembling the task specific routings. The performance of the final task-specific routings is returned to CoDeepNEAT and attributed to the modules in the standard way, i.e., by averaging over their performance in independent runs of CTR. The resulting algorithm searches for multitask solutions in a highly general space aligning with the grand vision of deep multitask learning: functional modules can take on arbitrary topological form and are assembled in arbitrarily diverse topologies for each task. CMTR's evolutionary loop works as follows:

1. CoDeepNEAT initializes a population of modules. The blueprints that are normally specify module connectivity in CoDeepNEAT are not used.
2. Modules are randomly chosen from automatically-determined species and grouped together into sets of K modules each.
3. Each set of modules is fed to CTR, which assembles the modules by evolving task-specific routings. The performance of the evolved routings on a task is returned as fitness.
4. Fitness is attributed to the modules, and NEAT's evolutionary operators applied to evolve the modules.
5. The process repeats from step 1 until CoDeepNEAT terminates, i.e. no improvement for a given number of generations.

Like in CTR, the weights between modules are always shared in CMTR across all locations they are applied. Thus, even though the module architectures are evolved to for the specific set of tasks, their learned behavior of each module is forced to be general through the CTR training process.

7.4 Experiments

This section details experiments comparing the methods in the Omniglot MTL domain.

7.4.1 Omniglot Character Recognition

The Omniglot dataset is the same one used in Chapter 6. Previous deep MTL approaches used random training/testing splits for evaluation (Bilen & Vedaldi, 2017; Meyerson & Miikkulainen, 2018a; Yang & Hospedales, 2017). However, with model search (i.e. when the model architecture is learned as well), a validation set separate from the training and testing sets is needed. Therefore, in the experiments in this chapter, a fixed training/validation/testing split of 50%/20%/30% is introduced for each task. Because training is slow and increases linearly with the number of tasks, a subset of 20 tasks out of the 50 possible is used in the current experiments. These tasks are trained in the same fixed random order used in (Meyerson & Miikkulainen, 2018a).

7.4.2 Experimental Setup

All networks are trained using Adam (?). For CTR, the network is trained for 120 meta-iterations of 3,000 backprop iterations each. For CMTR, for efficiency, each network is only trained for 12 meta-iterations before its fitness is reported. Each iteration is equivalent to one full forward and backward pass through the network with a single example image and label chosen randomly from each task. The fitness assigned to each network is the average validation accuracy across the 20 tasks. All setups use a total of four modules, as in previous work.

Since CTR uses a (1+1) evolutionary algorithm and trains all candidates jointly, it is run on a single GPU. CMTR uses a module population of 25 with two species. During each generation, 100 module sets are assembled from this population. The evaluation of these module sets via CTR is distributed over 100 separate EC2 instances with a K80 GPU in AWS. The average time for training is around 1-2 hours depending on the size of the modules in the set.

Since each multitask model generated with CMTR is only trained for 12 meta-iterations during evolution, to find the best module set, the top 50 sets from the run are retrained for the full 120 meta-iterations. During training, a snapshot of the network is taken at the point of highest validation accuracy. This snapshot is then evaluated and the average test accuracy over all tasks returned.

7.4.3 Results

Figure 7.3 compares how fitness (i.e. average validation accuracy) improves for CTR (using the default modules) and CMTR (using the best evolved modules discovered by CMTR) during training, averaged over 10 runs. Interestingly, while CTR improves faster in the first 10 meta-iterations, it is soon overtaken by CMTR, demonstrating how evolution discovers modules that leverage the available training better. Table 7.1 shows the validation and test errors for the best evolved network produced by each method, averaged over 10 runs. The best-performing methods are highlighted in bold and standard error for the 10 runs is shown in parenthesis. In addition, performance of the baseline methods are shown, namely (1) a single-task architecture, i.e. where each task is trained and evaluated separately, and (2) the soft ordering network

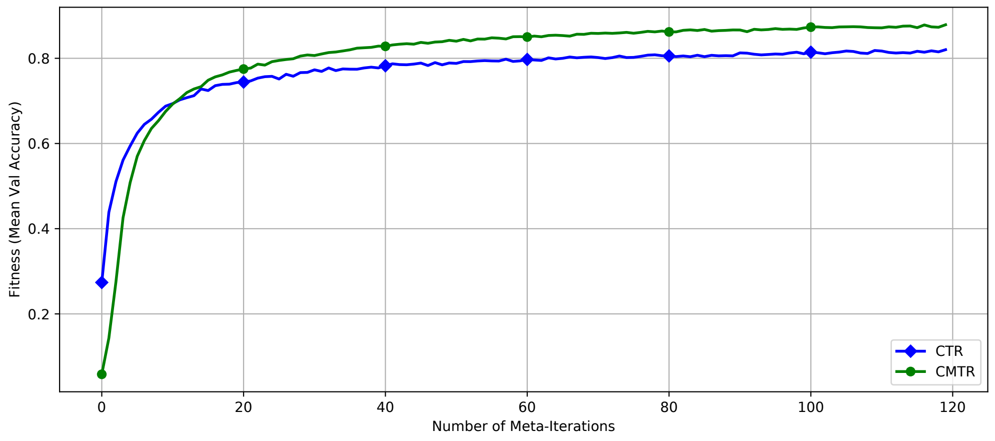


Figure 7.3: Comparison of fitness over number of meta-iterations of training for CTR and CMTR. Evolution discovers modules that leverage the available training better, forming a synergy of the two processes.

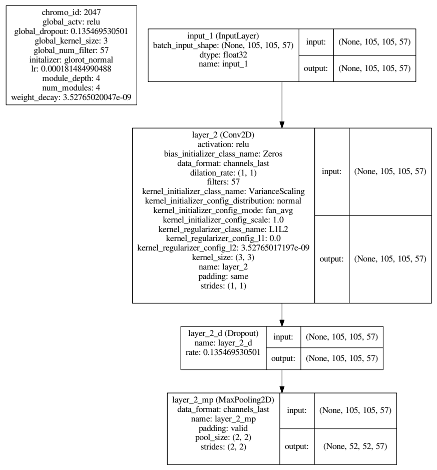
Method	Best Val Error (%)	Test Error (%)
Single Task	36.41 (0.53)	39.19 (0.50)
Soft Ordering	32.33 (0.74)	33.41 (0.71)
CTR	17.52 (0.21)	17.64 (0.19)
CMTR	**11.80** (1.02)	**12.18** (1.02)

Table 7.1: Average validation and test errors over all 20 tasks for each algorithm. CMTR performs the best as it combines both module and routing evolution. Pairwise t-tests show all differences are statistically significant with $p < 0.05$.

architecture (Meyerson & Miikkulainen, 2018a). The evolutionary methods substantially improve upon the baselines: Evolving task-specific routings is better than using fixed routings, and evolving module architectures gives an additional boost, as CMTR performs the best.

The best networks have approximately three million parameters. Figure 7.4 visualizes one of the best performing sets of modules from the CMTR experiment; Figure 7.5 visualizes sample routing topologies evolved for the different alphabets for this same CMTR experiment. Because the CoDeepNEAT outer loop is based on two species, the four modules passed to the CTR inner loop consist of two different designs (but still separate weights). Thus, evolution has discovered that a combination of simple and complex modules is beneficial. Similarly, while the routing topologies for some alphabets are simple, others are very complex. Moreover, similar topologies emerge for similar alphabets (such as those that contain prominent horizontal lines, like Gurmukhi and Manipuri). Also, when evolution is run multiple times, similar topologies for the same alphabet result. Such useful diversity in modules and routing topologies, i.e. structures that complement each other and work well together, would be remarkably difficult to develop by hand. However, evolution discovers them consistently and effectively, demonstrating the power of the approach.

To further analyze the distribution of evolved architectures, the topologies are embedded as vectors in a 29-dimensional hand-designed feature space. For each module, there is a feature indicating how many times that module is used in the topology. The rest of the features are based on the depth of



chromo_id: 2047
global_actv: relu
global_dropout: 0.135469530501
global_kernel_size: 3
global_num_filter: 57
initalizer: glorot_normal
lr: 0.000181484990488
module_depth: 4
num_modules: 4
weight_decay: 3.52765020047e-09

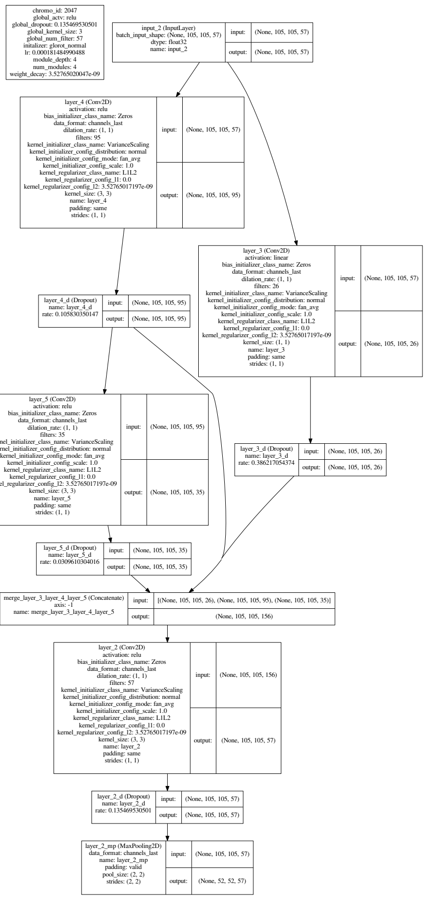
input_1 (InputLayer)
batch_input_shape: (None, 105, 105, 57)
dtype: float32
name: input_1
input: (None, 105, 105, 57)
output: (None, 105, 105, 57)

layer_2 (Conv2D)
activation: relu
bias_initializer_class_name: Zeros
data_format: channels_last
dilation_rate: (1, 1)
filters: 57
kernel_initializer_class_name: VarianceScaling
kernel_initializer_config_distribution: normal
kernel_initializer_config_mode: fan_avg
kernel_initializer_config_scale: 1.0
kernel_regularizer_class_name: L1L2
kernel_regularizer_config_l1: 0.0
kernel_regularizer_config_l2: 3.52765017197e-09
kernel_size: (3, 3)
name: layer_2
padding: same
strides: (1, 1)
input: (None, 105, 105, 57)
output: (None, 105, 105, 57)

layer_2_d (Dropout)
name: layer_2_d
rate: 0.135469530501
input: (None, 105, 105, 57)
output: (None, 105, 105, 57)

layer_2_mp (MaxPooling2D)
data_format: channels_last
name: layer_2_mp
padding: valid
pool_size: (2, 2)
strides: (2, 2)
input: (None, 105, 105, 57)
output: (None, 52, 52, 57)

(a) Topology of modules one and three.



chromo_id: 2047
global_actv: relu
global_dropout: 0.135469530501
global_kernel_size: 3
global_num_filter: 57
initalizer: glorot_normal
lr: 0.000181484990488
module_depth: 4
num_modules: 4
weight_decay: 3.52765020047e-09

input_2 (InputLayer)
batch_input_shape: (None, 105, 105, 57)
dtype: float32
name: input_2
input: (None, 105, 105, 57)
output: (None, 105, 105, 57)

layer_4 (Conv2D)
activation: relu
bias_initializer_class_name: Zeros
data_format: channels_last
dilation_rate: (1, 1)
filters: 95
kernel_initializer_class_name: VarianceScaling
kernel_initializer_config_distribution: normal
kernel_initializer_config_mode: fan_avg
kernel_initializer_config_scale: 1.0
kernel_regularizer_class_name: L1L2
kernel_regularizer_config_l1: 0.0
kernel_regularizer_config_l2: 3.52765017197e-09
kernel_size: (3, 3)
name: layer_4
padding: same
strides: (1, 1)
input: (None, 105, 105, 57)
output: (None, 105, 105, 95)

layer_3 (Conv2D)
activation: linear
bias_initializer_class_name: Zeros
data_format: channels_last
dilation_rate: (1, 1)
filters: 26
kernel_initializer_class_name: VarianceScaling
kernel_initializer_config_distribution: normal
kernel_initializer_config_mode: fan_avg
kernel_initializer_config_scale: 1.0
kernel_regularizer_class_name: L1L2
kernel_regularizer_config_l1: 0.0
kernel_regularizer_config_l2: 3.52765017197e-09
kernel_size: (1, 1)
name: layer_3
padding: same
strides: (1, 1)
input: (None, 105, 105, 57)
output: (None, 105, 105, 26)

layer_4_d (Dropout)
name: layer_4_d
rate: 0.105830350147
input: (None, 105, 105, 95)
output: (None, 105, 105, 95)

layer_5 (Conv2D)
activation: relu
bias_initializer_class_name: Zeros
data_format: channels_last
dilation_rate: (1, 1)
filters: 35
kernel_initializer_class_name: VarianceScaling
kernel_initializer_config_distribution: normal
kernel_initializer_config_mode: fan_avg
kernel_initializer_config_scale: 1.0
kernel_regularizer_class_name: L1L2
kernel_regularizer_config_l1: 0.0
kernel_regularizer_config_l2: 3.52765017197e-09
kernel_size: (3, 3)
name: layer_5
padding: same
strides: (1, 1)
input: (None, 105, 105, 95)
output: (None, 105, 105, 35)

layer_3_d (Dropout)
name: layer_3_d
rate: 0.386217054374
input: (None, 105, 105, 26)
output: (None, 105, 105, 26)

layer_5_d (Dropout)
name: layer_5_d
rate: 0.0309610304016
input: (None, 105, 105, 35)
output: (None, 105, 105, 35)

merge_layer_3_layer_4_layer_5 (Concatenate)
axis: -1
name: merge_layer_3_layer_4_layer_5
input: [(None, 105, 105, 26), (None, 105, 105, 95), (None, 105, 105, 35)]
output: (None, 105, 105, 156)

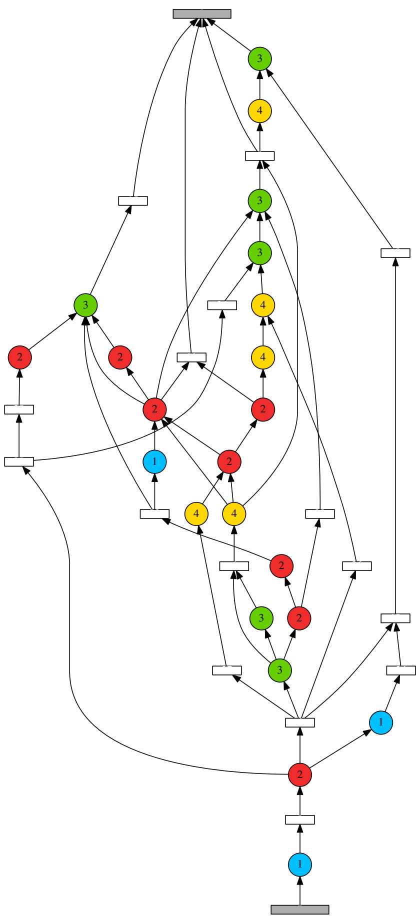
layer_2 (Conv2D)
activation: relu
bias_initializer_class_name: Zeros
data_format: channels_last
dilation_rate: (1, 1)
filters: 57
kernel_initializer_class_name: VarianceScaling
kernel_initializer_config_distribution: normal
kernel_initializer_config_mode: fan_avg
kernel_initializer_config_scale: 1.0
kernel_regularizer_class_name: L1L2
kernel_regularizer_config_l1: 0.0
kernel_regularizer_config_l2: 3.52765017197e-09
kernel_size: (3, 3)
name: layer_2
padding: same
strides: (1, 1)
input: (None, 105, 105, 156)
output: (None, 105, 105, 57)

layer_2_d (Dropout)
name: layer_2_d
rate: 0.135469530501
input: (None, 105, 105, 57)
output: (None, 105, 105, 57)

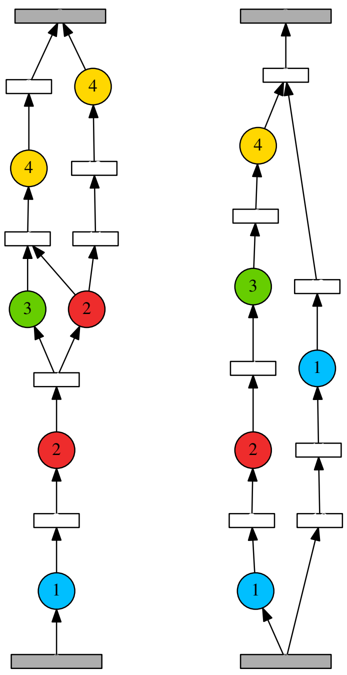
layer_2_mp (MaxPooling2D)
data_format: channels_last
name: layer_2_mp
padding: valid
pool_size: (2, 2)
strides: (2, 2)
input: (None, 105, 105, 57)
output: (None, 52, 52, 57)

(b) Topology of modules two and four.

Figure 7.4: Structures of the best module set discovered by CMTR. The two species in CMTR evolve very different modules: one simple and one complex. The thick boxes represent convolutional, medium max pooling, and thin dropout layers, with hyperparameters listed on the left. This complementarity would be difficult to develop by hand, demonstrating the power of evolution in designing complex systems.



(a) Routing for Angelic, an invented alphabet.



(b) Routings for Gurmukhi and Mujarati.

Figure 7.5: Structures of the best routing topologies discovered by evolution. The routing topologies represent a range from simple to complex; similar alphabets have similar topologies, and the structure is consistently found. Again, such useful diversity would be difficult to develop by hand, demonstrating the power of evolution in designing complex systems.

a module, defined as the length of the largest subsequence of pooling layers between the input and that module, i.e., how much downsampling has been performed. For each depth, there is a feature indicating how many modules are used at that depth. For each module-depth pair, there is a feature indicating how many times that module is used at that depth. Since there are four modules and five depths, i.e., zero to four pooling layers can precede each module, there are a total of $4 + 5 + 4 \times 5 = 29$ dimensions in the feature space. Each dimension is then normalized across all topologies to have a mean of zero and a variance of one. Although this featurization ignores much of the graph structure of each topology, it incorporates the idea that different depths induce qualitatively different pseudo-tasks.

A two-dimensional t-SNE (van der Maaten & Hinton, 2008) visualization of the final topologies from a fifty-task run of CTR is shown in Figure 7.6. Although the evolutionary process does incur a substantial amount of noise, there is still evident structure. For example, the North American alphabets are off to the left, the Invented languages are towards the top, while the Asian languages are towards the bottom. Such structure is intriguing, since alphabets in the real world tend to develop incrementally by region and across regions (Daniels & Bright, 1996), while CTR can incrementally exploit knowledge accumulated across alphabets.

The topology embeddings were also compared to visual embeddings of the alphabets themselves. Such embeddings of dimensionality 1024 were generated using the MobileNet vision architecture pretrained on ImageNet

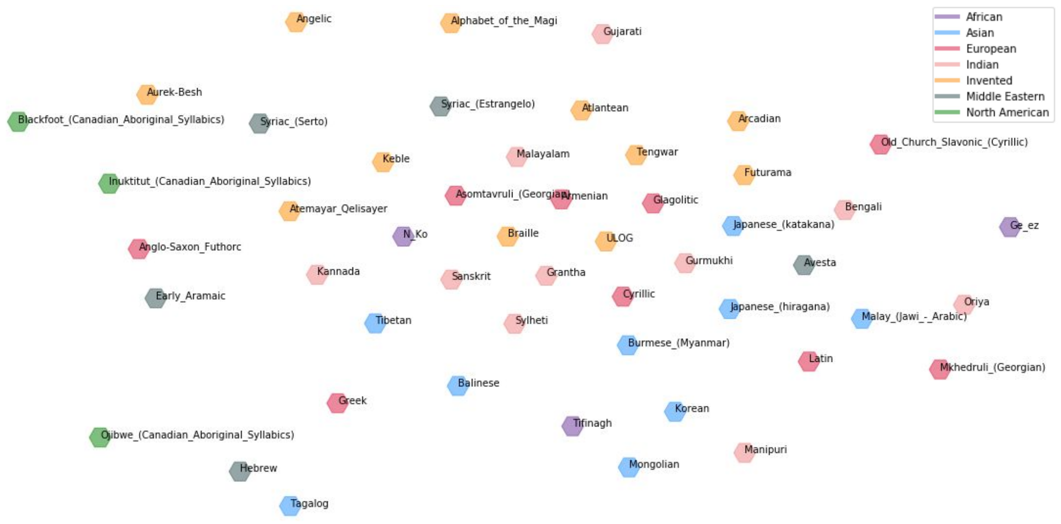


Figure 7.6: **Visualization of CTR topology embeddings.** This figure shows a t-SNE projection (van der Maaten & Hinton, 2008) of the featurization of evolved topologies from a run of CTR on all fifty Omniglot tasks, colored according to geography (including Invented alphabets in orange). Although the evolutionary process does incur significant noise, there is still evident structure. For example, the North American alphabets are off to the left, the Invented languages are towards the top, while the Asian languages are towards the bottom. Such structure is intriguing, since alphabets tend to develop incrementally by region and across regions, while CTR can incrementally exploit knowledge accumulated across alphabets.

(Howard et al., 2017). The visual embedding for each alphabet was the mean over that of all of its characters. The t-SNE projection of these visual alphabet embeddings is given in Figure 7.7.

In order to compare the topologies and alphabets quantitatively, for each alphabet, the nearest-neighbor alphabet was computed both respect to topology and visual embedding using L1 distance. Out of the fifty alphabets, there were three cases where the nearest neighbor was the same for

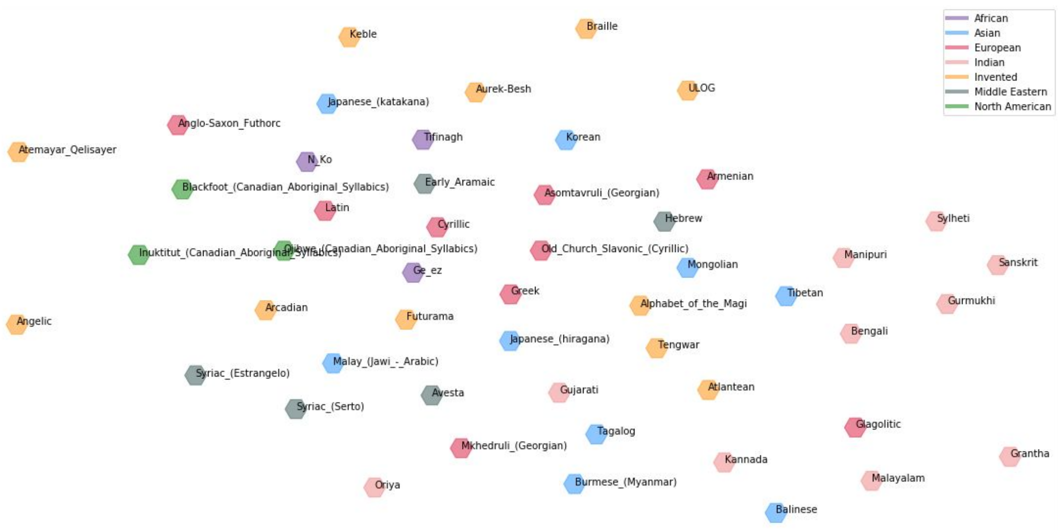


Figure 7.7: **Visualization of CTR visual embeddings.** This figure shows a t-SNE projection of alphabets as in Figure 7.6, but using visual embeddings for each alphabet produced via MobileNet, instead of using embedding derived from evolved topologies. This projection has a structure similar to that in Figure 7.6, although both European and Indian alphabets are more tightly grouped. Overall, the organization is similar to that of 7.6, demonstrating that similar topologies are discovered for similar alphabets.

both kinds of embedding: the nearest neighbor of Sanskrit was Gurmukhi, a related Indian alphabet; that of Anglo-Saxon Futhorc was Early Aramaic, both ancient cuniform-looking alphabets; and that of Atlantean was Futurama, both alphabets invented for use in animated worlds. Note that there is less than a 6% chance of seeing at least this number of matches (permutation test by permuting alphabet labels). These connections between the topological and visual representations suggest that CTR indeed captures intuitive visual regularities.

7.5 Discussion and Future Work

The experiments show that MTL can improve performance significantly across tasks, and that the architecture used for it matters a lot. Multiple ways of optimizing the architecture are proposed in this chapter and the results lead to several insights. First, module architectures used in the joint multitask model can be optimized, and their designs end up diverging in a systematic way. Unlike in the original soft ordering architecture, evolution in CMTR results in discovery of a wide variety of simple and complex modules, and which are reused many times. Evolution thus discovers a useful set of building blocks that are diverse in structure. Second, the routing of the modules matters. The power of CTR is from evolving different topologies for different tasks, and tying the tasks together by sharing their constituent modules. In addition, sharing components (including learned parameter values) in CMTR is crucial to its performance. If indeed the power from multitask learning comes from integrating requirements of multiple tasks, this integration will happen in the core functions that modules encode, so it makes sense that sharing plays a central role.

There are several directions for future work. The proposed algorithms can be extended to many applications that lend themselves to a multitask setting. For instance, it will be interesting to see how it can be used to find synergies in different tasks in vision, and in language. Further, as has been shown in related work, the tasks do not even have to be closely related to gain the benefit from MTL. For instance, object recognition can be paired with caption generation. It is possible that the need to express the contents of an image in words will help object recognition, and vice versa. Discovering ways to tie such multimodal tasks together should be a good opportunity for evolutionary optimization, and constitutes a most interesting direction for future work. One such approach to this problem is considered in Chapter 8.

7.6 Conclusion

This chapter presented EAs for optimizing the architectures of deep multitask networks constructed from multi-purpose modules. They extend upon previous work which has shown that carefully designed routing and sharing of modules can significantly help multitask learning. The power of the proposed algorithms is shown by dramatically improving performance on an existing multitask learning benchmark problem. By discovering new ways to use modules, and training them in these new locations, these architecture search approaches increase their breadth of applicability. The next chapter takes an orthogonal approach by enabling modules to be shared across diverse kinds of architectures and problem areas.

Chapter 8

Modular Universal Reparameterization

This chapter presents a final system for learning sets of generic functional modules. This system solves two problems that limit the systems in previous chapters: It scales many-module systems to complex modern architectures, and it shares modules across diverse architectures and problem areas. Unlike the systems in Chapter 7, the one in this chapter makes no changes to the functional form of the underlying predictive model. Instead, it breaks the parameter set for a model into parameter blocks, each of which is parameterized by a module. As a result, the modules that are learned are fully generic, in that they can be applied to any kind of architecture whose parameters can be chunked into blocks of the given size. This generality enables sharing across problems of different modalities, e.g., from vision to text to genomics, and different layer types, e.g., from convolutions to LSTMs to fully-connected layers. The results indicate that sharing can be beneficial in this setting, which opens the door to future methods that accumulate vast knowledge bases over highly diverse problems and indefinite lifetimes.

8.1 Motivation

One of the grand promises of multitask learning is to provide a means for sharing knowledge across seemingly disparate settings. The simple fact that all tasks of interest have been generated from our world and formulated in a way that makes sense to us is evidence that any set of real world tasks could be practical to share across: They share these physical and human biases. The idea that sharing should always be useful has been formalized from a theoretical perspective (Mahmud & Ray, 2008; Mahmud, 2009). From a more practical perspective, it is already possible to see how this knowledge is shared when the same machine learning method is applied across a wide range of problems. Knowledge is accumulated from one or more domains as the method is developed, and the generalizability of this knowledge is demonstrated when the method is successfully applied to further domains.

An especially striking instance of this generalizability has been the recent dominance of broad swathes of machine learning problems by deep neural networks. A standard deep neural network has many (often millions of) free parameters, often allowing it to memorize training data exactly if trained to complete convergence. However, the fact that, architecturally, different variants of the same underlying components are successfully applied across the board shows that this same "stuff" that deep neural networks are made of is very general. From this perspective, all standard deep learning approaches are successfully performing multitask learning where the knowledge shared takes the form of the topological components used to construct the models for each task. But is there more that can be shared beyond topological components and training methods? This chapter investigates how we can learn more about where the values of these many parameters come from, rather than simply saying they can take on any values, which are separately determined for each single-task model.

The perspective starts with the observation that the parameterization of the vast majority of deep models is encoded as linear transformations. A given model's behavior is given by the the assembly of many of these linear mappings into its architecture. Functional modules can parameterize these linear mappings, and to make each module universally applicable to all pseudo-tasks, a joint model is decomposed into equally-sized blocks of parameters $\boldsymbol{B}_\ell \in \mathbb{R}^{m \times n}$, each inducing a pseudo-task. Modules generate these parameters by mapping a context associated with each location to the parameters for the block at that location. Importantly, this reparameterization neither depends on nor affects the architecture of the underlying model. Thus, it provides a clear framework for discovering modules that can be used across any problem, in particular, across problems requiring qualitatively diverse architectures. The remainder of this chapter describes the reparameterization in more detail, then introduces and evaluates an algorithm for mapping modules to pseudo-task locations across diverse tasks.

8.2 Reparameterization by Hypermodules

Suppose you have T tasks $\{\{x_i, y_i\}_{i=1}^{N_t}\}_{t=1}^{T}$, with corresponding model architectures $\{\mathcal{M}_t\}_{i=1}^{T}$, each parameterized by a set of parameters $\theta_{\mathcal{M}_t}$. Suppose each $\theta_{\mathcal{M}_t}$ can be decomposed into equally-sized parameter blocks $\boldsymbol{B}_\ell$ of size $m \times n$, and there are L such blocks total across all $\theta_{\mathcal{M}_t}$. Then, the parameterization for the entire joint model can be rewritten as:

$$\bigcup_{t=1}^{T} \theta_{\mathcal{M}_t} = (\boldsymbol{B}_1, \ldots, \boldsymbol{B}_L) \tag{8.1}$$

That is, the entire parameter set can be regarded as a single tensor $\boldsymbol{B} \in \mathbb{R}^{L \times m \times n}$. In the standard deep learning setup, each $\boldsymbol{B}_\ell$ is parameterized separately. In another approach, taken in this chapter, each $\boldsymbol{B}_\ell$ is generated by a *hypermodule.* This perspective draws from previous work on using hypernetworks to reparameterize neural networks (Stanley et al., 2009; Ha et al., 2017).

8.2.1 Hypermodules

Associate with the ℓth block location a context vector $\boldsymbol{z}_\ell \in \mathbb{R}^c$. These contexts contain the location-specific parameters that are not shared across locations, analogous to the task-specific factors found in factorization-based MTL methods (Argyriou et al., 2008; Kang et al., 2011; Yang & Hospedales, 2015, 2017), and task-specific parameters more generally. Suppose that we also have a collection of K hypermodules $U = \{\boldsymbol{H}_k\}_{k=1}^{K}$, where $\boldsymbol{H}_k \in \mathbb{R}^{c \times m \times n}$. Finally, let $\psi : \{1, \ldots, L\} \to U$ be a hypermodule mapping function that tells us which hypermodule generates the parameters of the ℓth block. Then, the

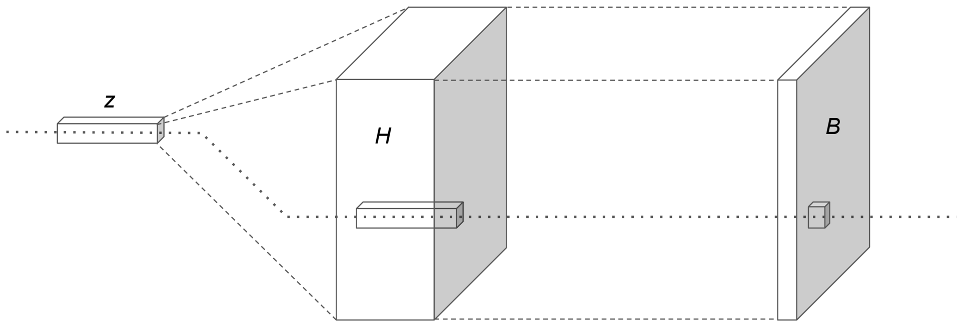


Figure 8.1: **Hypermodule Parameter Generation.** The parameters of a parameter block $\boldsymbol{B}_\ell$ are generated by applying a hypermodule $\boldsymbol{H}_k$ to the block's context vector $\boldsymbol{z}_\ell$. The blocks parameters are generated as the 1-mode (vector) product of the hypermodule and the context. That is, instead of learning all of its parameters independently, the block gets its parameters by tuning a generic module to this particular location.

parameters of the model are generated by

$$\boldsymbol{B}_\ell = \psi(\ell) \ \bar{\times}_1 \ \boldsymbol{z}_\ell \tag{8.2}$$

where $\bar{\times}_1$ denotes the *1-mode (vector) product* of a tensor and a vector (Kolda & Bader, 2009). Elementwise, this is written as

$$\boldsymbol{B}_{\ell ij} = \langle \psi(\ell)_{:ij}, \boldsymbol{z}_\ell \rangle \tag{8.3}$$

In other words, the value at $\boldsymbol{B}_{\ell ij}$ is the dot product between $\boldsymbol{z}_\ell$ and the fiber in $\psi(\ell)$ associated with the (i,j)th element of $\boldsymbol{B}_\ell$. A visualization of this generation of block parameters via a hypermodule is shown in Figure 8.1. Notice that the 1-mode multiplication could also be implemented as a transposed 2D convolution when considering $\boldsymbol{z}_\ell$ to be a single spatial pixel with c channels. However, in practice it is more convenient and computationally efficient to generate all $\boldsymbol{B}_\ell$ simultaneously as a batch matrix multiplication between $(\boldsymbol{z}_1, \dots, \boldsymbol{z}_L)$ and the mode-1 matricizations of $(\psi(1)_{(1)}, \ldots, \psi(L)_{(1)})$. This operation coupled with an H_ℓ defines a functional module for this system.

Since each pseudo-task is associated with a single hypermodule, the generation of parameters is block-sparse with respect to which parameters are used at each location. This sparsity mirrors the desirable quality of discovering effective block-sparse matrices for grouping related tasks in the case of linear multitask feature learning (Kang et al., 2011; Kumar & Daumé, 2012). Of course, with this reparameterization, since all newly introduced operations are differentiable, the joint model can still be trained end-to-end with gradient descent.

Through this reparameterization by hypermodules, the block decomposition (Equation 8.1) can now be written as

$$\bigcup_{t=1}^{T} \theta_{\mathbb{M}_t} = [(\boldsymbol{H}_1, \ldots, \boldsymbol{H}_K), (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_L)] \tag{8.4}$$

where $\theta_{\mathbb{M}_t}$ is the original parameter set for the tth task, $\boldsymbol{H}_k$ are hypermodules, and $\boldsymbol{z}_\ell$ are contexts, one of which is associated with each pseudo-task.

8.2.2 Decomposition of Common Neural Network Layers

This section gives concrete examples of how the parameters of a neural network layer can be broken into equally-sized blocks, so that each block parameterizes a linear transformation. The blocks for an entire network are then just the concatenation of the blocks for each of its constituent layers. The decomposition of the three layer types below are used in the experiments in subsequent sections in this chapter.

Fully-Connected Layers. The weight matrix of a fully-connected layer with pm inputs and qn outputs characterizes a linear transformation $\boldsymbol{W} \in \mathbb{R}^{pm \times qn}$. This matrix $\boldsymbol{W}$ can be broken into pq blocks of size $m \times n$. The (i, j)th block then defines a linear transformation between the im to $(i+1)m$ units of the input space and the jn to $(j+1)n$ units of the output space. The final output of the layer for each subset of output units jn to $(j+1)n$ is the sum over all i of the output of the (i, j)th block. Since these locations all join together to solve a single problem, i.e., the problem the complete fully-connected layer solves, it is understandable that they may be able to share information effectively. Indeed, previous work has shown that fully-connected layers often contain redundant information that can be reduced or reorganized via other kinds of reparameterizations (Stanley et al., 2009; Denil et al., 2013; Chen et al., 2015; Cheng et al., 2015; Li et al., 2018).

Convolutional Layers. The case of convolutional layers is similar to the case of fully-connected layers, with additional dimensions included, based on the shape of the convolutional kernel defining the spatial field of the layer. A convolutional weight tensor applies a distinct dense linear transformation to each element in the spatial field, e.g., to each word token in the 1D case, or to each pixel in the 2D case. The outputs of these linear transformations are then aggregated over the entire spatial field. Each of the transformations can be viewed as solving the distinct but related task of: What outputs do I want given that I am looking at this particular element of the spatial field? Thus, the most natural way to decompose a convolutional weight tensor with F elements in its spatial field is to decompose each of the F linear transformations independently as in the fully-connected case. If the convolutional layer has pm input channels and qn output channels, the resulting decomposition will have Fpq blocks.

LSTMs. The parameters of an LSTM layer can be viewed as a tuple of fully-connected layers, each of which solves a distinct but related purpose, e.g., how to process new input, update cell state, and produce updated output. Each of these linear transformations can be broken down into blocks independently, as in the case of the fully-connected layer.

From the three above examples, it should be easy to see how this decomposition can be extended to other types of layers whose parameters are grounded in linear transformations. The key idea is that the decomposition does not care about the network overall or even individual layers, but works at the level of decomposing the linear transformations that constitute the core of the knowledge learned by and stored in the model. Of course, these decompositions rely on the fact that for each linear transformation in the layer, its input size is divisible by m and its output size is divisible by n. This assumption is not unreasonable, since layer sizes in deep models often have many common factors, and can be adjusted to a minor extent with negligible effects on performance. In all experiments in this chapter this assumption is met; however, it would even be reasonable to simply truncate any generated parameters that do not fit into blocks of size less than $m \times n$. Other potential solutions to this question are proposed in Chapter 9.

8.2.3 Parameter Initialization

In order to take advantage of the desirable features of the underlying model, these features should be preserved when possible. One of these features is parameter initialization. Historically, neural networks trained with backpropagration have been very sensitive to initialization, and it has taken several recent advances in theory and experimentation to develop simple, efficient and reliable initialization schemes (Glorot & Bengio, 2010; He et al., 2016).

Currently, the most popular method is to initialize each layer by determining analytically an ideal variance σ^2 of parameters in that layer and initializing the parameters from either a uniform or normal distribution with variance σ^2 and mean 0. Suppose each block location is associated with a distinct hypermodule $\boldsymbol{H}_\ell$. To enable reparameterization to strictly subsume the original model parameterization, the parameters of $\boldsymbol{H}_\ell \in \mathbb{R}^{c \times m \times n}$ and $\boldsymbol{z}_\ell \in \mathbb{R}^c$ should be initialized so that $\boldsymbol{B}_{\ell ij} = \langle \boldsymbol{H}_{\ell:ij}, \boldsymbol{z}_\ell \rangle$ (from Equation 8.3) is drawn from the intended distribution. To avoid introducing unintended bias into the models, $\boldsymbol{H}_\ell$ and $\boldsymbol{z}_\ell$ much each have i.i.d. initialization.

Notice that whenever $c > 1$, $\boldsymbol{B}_{\ell ij}$ is the sum of two or more random variables. This fact immediately rules out the possibility of initializing $\boldsymbol{B}_{\ell ij}$ from a uniform distribution. However, it is possible to initialize from a normal distribution, which is the standard initialization for the models used in the experiment sections in this chapter. One solution is to initialize $\boldsymbol{H}_\ell$ from a normal distribution $\mathcal{N}(0, \sigma_o^2)$, and initialize $\boldsymbol{z}_\ell$ to have constant magnitude $|z|$. Specifically,

$$\boldsymbol{B}_{\ell ij} = \langle \boldsymbol{H}_{\ell :ij}, \boldsymbol{z}_\ell \rangle \sim |z| c \mathcal{N}(0, \sigma_o^2) = \mathcal{N}(0, z^2 c^2 \sigma_o^2) = \mathcal{N}(0, \sigma^2) \\ \implies |z| = \frac{\sigma}{c \sigma_o} \quad (8.5)$$

Given c and σ, there is no unique satisfying choice for $|z|$ and σ_o. However, hypermodules would ideally be initialized independently of the locations in which they are used: hypermodules should only contain generic information, and all location-specific information should be contained in the context vector associated with that location. So, all H_ℓ are initialized from the same normal distribution $\mathcal{N}(0, \sigma_o^2)$, and each $\boldsymbol{z}_\ell$ initialized accordingly with a potentially distinct $|z|$.

Now, in the above formulation, $\boldsymbol{z}_\ell$ can either be initialized with a constant value z or uniformly at random from $\{-z, z\}^c$. If ψ is initialized with the same hypermodule used at multiple locations in the same layer, initializing $\boldsymbol{z}_\ell$ to be constant will cause some parameters of the layer to be initialized to the exact same values, locking them into a symmetry that cannot be broken through training, effectively destroying some fraction of the model. For example, if the model is initialized with a single hypermodule, similar to previous work on hypernetworks (Ha et al., 2017), the effect on the model will be unrecoverable. Initializing $\boldsymbol{z}_\ell$ uniformly at random from $\{-z, z\}^c$ successfully avoids this pitfall if c is large enough so that the chance of collisions is sufficiently low.

That said, in the experiments in this chapter, each location is always initialized with a distinct hypermodule, so all $\boldsymbol{z}_\ell$ can be safely initialized to be constant, which avoids any unintended bias as to the a priori semantics of each location. In particular, in the experiments in this chapter, all σ^2 and σ_o^2 are determined by *He normal* initialization (He et al., 2016), i.e., they are computed based on the fan-in of the layer in which they are initialized. Although it may seem pessimistic to initialize each location with its own hypermodule, the analysis in Section 8.3.2 shows that this initialization need not induce slower convergence to the optimal hypermodule mapping.

8.2.4 Ensuring Consistent I/O Semantics across Pseudo-tasks

For the same module to be successfully applied to different locations in qualitatively distinct areas of the joint model, it is important that its inputs always have the same *meaning*. For example, consider the possible pitfall that, if there are two tasks that are equivalent except that their input spaces are permuted, a model that works for one will not work for the other. This issue is avoided by ensuring each task model has an *input adapter* and *output adapter*. The input adapter maps the raw task input space to a space consistent with the module set; the output adapter maps a space consistent with the module set to the task output. These adapters are task or domain specific, so, to enable maximal sharing, they should have limited capacity. In the experiments in this chapter, for vision tasks the input adapter is the initial convolutional layer with m output channels; for language and genomic tasks, the input adapter is the initial embedding layer with embedding dimension m. Similarly, the output adapter for each task is simply the final fully-connected layer whose output is the predictions for the task. Since these adapters are linear transformations, they can capture any necessary permutations and scaling of inputs that enable module cross-compatibility. Finally, note that between the input and output adapters, the modules themselves are trusted to ensure input and output are meaningful at each location. By evaluating the success of a model parameterized by modules, it is possible to verify that the adapters and modules have indeed avoided this pitfall successfully.

Input and output adapters contain domain-specific parameters that are not included in $\theta_{\mathcal{M}_t}$. In addition, there may be other model parameters that are not convenient or practical to include in the reparameterization. For clarity, these are left out of the formalization above. They can be considered as additional adapters that make up a relatively small portion of the architecture and parameter set. They help the system run smoothly, preserving structural properties of the underlying model.

8.2.5 Model Complexity

Note that the parameter generation method given by Equation 8.2 makes no additional assumptions about the values of the parameters, aside from the fact that they are each some linear combination of the context. This generality is in contrast to other tensor factorization approaches whose goal is to show that a particular structure of tensor factorization can yield more parsimonious representations (Long et al., 2017; Yang & Hospedales, 2017), and means that the reparameterized model strictly subsumes the original (standard) parameterization. If reparameterization is to achieve parsimony, it must do so autonomously by reusing hypermodules efficiently through optimizing ψ. By reusing hypermodules, the system can reduce the number of total parameters in the model. That is, the system may decide to reuse the same hypermodule in multiple locations, which has the side-benefit of reducing the model parameter complexity.

When rewritten as $\boldsymbol{B}$ in Equation 8.1, it is easy to see that the original joint model has Lmn trainable parameters. When reparameterized as hypermodules, the model has $Lc + Kcmn$ trainable parameters. In particular, the reparameterized model has fewer parameters than the original only when

$$K < \frac{L(mn - c)}{cmn} < \frac{L}{c} \tag{8.6}$$

which means that on average each hypermodule is used in c different locations. However, when the model has been trained, any hypermodules used fewer than c times can be replaced with the parameters they generate, thus never incurring any increase of parameters due to redundant hypermodule capacity. Thus, the model complexity at inference is never greater than that of the original model:

$$(L - L_o)c + Kcmn + L_o mn \leq Lmn \tag{8.7}$$

where L_o is the number of block locations parameters by hypermodules used fewer than c times. Intuitively, more parameters should never be needed at inference, since the reparameterization only provides another way of generating the parameters $\theta_{\mathbb{M}_t}$. The expressivity and functional form of the model at inference is exactly the same as that of the original.

8.3 Theoretical Perspective on Optimizing Module Mappings

The previous sections described how joint models can be decomposed into blocks that are reparameterized by sets of hypermodules, and how the parameters in the reparameterization can be learned through gradient descent. There remains the question: How should hypermodules be mapped to block locations? In other words, how can an optimal ψ be found? This section presents an analysis of a simplified version of the problem from the perspective of the $(1 + \lambda)$ evolutionary algorithm (*EA*), which is a relatively simple evolutionary algorithm (stochastic global search) that is amenable to theoretical analysis (Droste et al., 2002; Doerr et al., 2008; Doerr & Auger, 2011; Durrett et al., 2011; Witt, 2013). The analysis extends existing convergence results of the $(1 + \lambda)$-EA on the Onemax problem (Doerr & Auger, 2011) to the case where there are more than two options for each location, where multiple EAs are run simultaneously, and where the number of possible values for each location is not known a priori.

Section 8.3.1 analyzes the effect that decomposing the problem into subproblems has on expected time to convergence to the optimal ψ. Section 8.3.2 demonstrates that the same asymptotic convergence rate can be achieved without knowing the optimal number of hypermodules a priori. Together, these results motivate the design of the practical algorithm presented in Section 8.4.

8.3.1 The Advantage of Problem Decomposition

Given that task architectures are fixed, the joint model is decomposed into a fixed number of blocks L. Suppose that there are K available hypermodules $\{\boldsymbol{H}_k\}_{k=1}^K$. Suppose there is a unique optimal $\psi_\star$. This optimal solution can be represented as a vector $(\psi_\star(0), \dots, \psi_\star(L))$, where there are K possible choices for each element of this vector. Suppose also that an ideal scoring function h is known, where $h(\psi)$ returns the Hamming distance between the vector representations of ψ and $\psi_\star$. Then, the problem of optimizing ψ is equivalent to Onemax with K choices for every bit.

On a problem of optimizing a vector of length L with K possible values at each location, with h a function that gives the utility (fitness) of a solution, the $(1 + \lambda)$-EA works as follow:

1. Initialize the current solution ψ_0 uniformly at random over all L^K possible solutions.
2. Generate λ new candidate solutions $\psi_1, \dots, \psi_\lambda$. Each candidate solution is a copy of the current solution, but, for each location, set its value to a new random value with probability p.

3. Set $\psi_0 := \underset{\psi_i}{\operatorname{argmax}}(h(\psi_i))$, with $0 \leq i \leq \lambda$.
4. Go to step 2.

In other words, the current best is replaced whenever a better solution is found, i.e., the $h(\psi_0)$ improves monotonically over time. So, this algorithm is inherently a greedy algorithm. However, notice that compared to random local search, in which each iteration alters a single location and checks for improvement, the $(1 + \lambda)$-EA is able to escape local optima by simultaneously altering multiple locations. Further differences between random local search and the $(1 + \lambda)$-EA have been documented in previous work (Doerr et al., 2008).

The optimal value for p is $\frac{1}{L}$, and with it, the $(1 + 1)$-EA solves One-max with binary values in $O(L \log L)$ time (number of iterations) with high probability (Witt, 2013). This result can be straightforwardly extended to show that in the case of K possible values for each location, the algorithm converges in $O(KL \log L)$ time with high probability. In the cases of $\lambda > 1$, such convergence times are generally reduced by a factor of λ.

The above analysis assumes that there is a single function h that gives the score of the entire mapping function, i.e., a single score for the entire multitask model. However, in the systems presented in Chapters 7 and 5, the best performers were selected on a *per task* basis, using variations of the $(1 + 1)$-EA. This is a natural choice, because each task produces a distinct validation loss that indicates how well the model performed on that task.

Intuitively, this per-task decomposition should make optimization easier, since the system is taking advantage of more detailed information, and is assigning performance more precisely based on per-task performance than if a single value were assigned to the joint model based on aggregated multitask performance.

But how much should per-task decomposition be expected to help in theory? Suppose the model for each task is decomposed into L_t blocks, so that $\sum_{t=1}^{T} L_t = L$. For simplicity, assume $L_t = \frac{L}{T} \; \forall \; t$. Similarly, ψ can be decomposed into per-task subsolutions $\psi_1, \dots, \psi_T$, and h into per-task scoring functions $h_1, \dots, h_T$, such that $h_t(\psi_t)$ gives the hamming distance between ψ_t and optimal subsolution $\psi_{t\star}$. Then, each ψ_t can be optimized with an independent instance of the $(1+1)$-EA, all of which are run in parallel. Each of these instances converges in $O(KL_t \log L_t) = O(\frac{KL(\log L - \log T)}{T})$ with high probability, and so we can show that the expected time for all of them to complete (equivalent to the time for the slowest of them to complete) is $O(\frac{KL(\log L - \log T)\log T}{T})$, when each location is mutated with probability $\frac{T}{L}$. This is because the CDF of the runtime for the $(1+1)$-EA is dominated by the CDF of an exponential random variable with mean $O(L \log L)$ (Witt, 2013), and the maximum of T i.i.d. exponential random variables with mean $\frac{1}{\rho}$ is $\frac{H_T}{\rho}$, where H_T is the Tth harmonic number $\sum_{t=1}^{T} \frac{1}{t} = O(\log T)$ (Eisenberg, 2008). The factor of $\frac{T}{\log T}$ speedup compared to the non-decomposed runtime of $O(KL \log L)$ yields tremendous speedups for large T. In particular, when the number of locations per task $\frac{L}{T}$ is relatively small, as in the experiments in Chapters 7 and 5, this convergence rate is satisfactory and practical.

However, in the block decomposition considered in this chapter (Equation 8.1), $\frac{L}{T}$ can grow large as the approach is scaled. For example, in the experiments in Section 8.7, $T = 3$ and each task is decomposed into thousands of blocks. In this case, the factor of $\frac{L}{T}$ becomes the bottleneck: The algorithm will be impractically slow even when K is small, i.e., even when the set of hypermodules is sufficiently compact. How can this bottleneck be addressed?

Note that each location in the block decomposition of the joint model defines a pseudo-task. So, if the problem were decomposed into L pseudo-tasks, each of length 1, the expected runtime would be reduced to $O(K \log L)$, when each location is mutated at every step. Similar to the case with T tasks, this is because the expected value of maximum of L geometric random variables with mean $\frac{1}{\rho}$ is also $O(\rho \log L)$. Importantly, now L only shows up as a log factor, making it highly scalable to large models with relatively small blocks. The differences in convergence time between the non-decomposed, task-decomposed, and block-decomposed algorithms are summarized in Table 8.1.

Thus, decomposing the optimization problem by pseudo-task can make a big difference The question is how to come up with functions that provide an accurate assessment of the utility of hypermodules at each location, i.e., approximations of ideal location-specific score functions h_ℓ. The Section 8.4 presents a practical solution to this problem.

Decomposition Level	Convergence Time	Speedup over None
None (Multitask)	$\Theta(KL\log L)$	1
Per-task (Single-task)	$\Theta(\frac{KL(\log L - \log T)\log T}{T})$	$\Theta(\frac{T}{(\log L - \log T)\log T}) = \Omega(\frac{T}{\log T})$
Per-block (Pseudo-task)	$\Theta(K\log L)$	$\Theta(L)$

Table 8.1: **Runtime Complexity of Optimizing Hypermodule Mapping.** This table gives the expected time to convergence of the $(1+1)$-EA on the problem of finding the optimal mapping of L block locations to K hypermodules, in a joint model containing T tasks. These theoretical results assume an ideal scoring function described in Section 8.3. The runtime of per-block decomposition (which assigns fitness to hypermodule mappings at the granularity of single-block pseudo-tasks) scales logarithmically as the number of blocks in the joint model increases, providing significant speedup.

8.3.2 Determining the Optimal Number of Hypermodules

Decomposing the optimization problem by location as described above dramatically reduces the dependence of the runtime on L. However, when the number of hypermodules K is large, the probability of a location selecting the correct hypermodule in a mutation becomes small, and this becomes the major bottleneck in optimization.

In general, the optimal number of hypermodules is not known a priori, so it is necessary to allow for the worst-case possibility that L hypermodules are required, i.e., the system is initialized with a distinct hypermodule at every location. Initializing with a distinct hypermodule at each location also satisfies the parameter initialization conditions discussed in Section 8.2.3. To enable this initialization feature, while preserving convergence rates, new candidate hypermodules are selected from a dynamic distribution at each location. This distribution corresponds to the likelihood that each hypermodule would be correct for a new location.

Consider again the case where the problem is fully decomposed into pseudo-tasks by location, so that L instances of the $(1+1)$-EA are run in parallel. Suppose ψ is initialized such that $\psi(\ell) = \boldsymbol{H}_\ell$, with all $\boldsymbol{H}_\ell$ distinct, and that $\psi_\star(\ell) = \boldsymbol{H}_1 \; \forall \ell$. Now, suppose that during a mutation each hypermodule is selected with probability proportional to the number of times it is used in the current ψ. Then, the expected time for the algorithm to converge to $\psi_\star$ is $O(\log L)$, i.e., it is asymptotically equivalent to the case from Section 8.3.1 where K is fixed to be constant.

The proof proceeds as follows. Let W_i be a variable tracking the number of locations whose hypermodule is wrong at iteration i. Now, $W_0 = L-1$, since the first location has its hypermodule initialized correctly. Let W_{i+1} be the expected number of locations whose hypermodule is incorrect at iteration $i+1$ given that W_i are incorrect at iteration i. Then,

$$W_{i+1} = W_i\left(1 - \frac{L - W_i}{L}\right) = \frac{W_i^2}{L}, \tag{8.8}$$

which yields a closed form for W_t:

$$W_t = \frac{1}{L}(\dots(\frac{1}{L}(\frac{1}{L}(L-1)^2)^2)\dots)^2 = \frac{(L-1)^{2^t}}{L^{2^t-1}}. \tag{8.9}$$

If there is at most 1 incorrect location, optimizing this final location clearly takes constant time. The goal is then to find t such that $W_t < 1$:

$$W_t = \frac{(L-1)^{2^t}}{L^{2^t-1}} = L\left(\frac{L-1}{L}\right)^{2^t} < 1 \tag{8.10}$$

$$\implies (\frac{L-1}{L})^{2^t} < \frac{1}{L}$$
$$\implies 2^t < \log_{\frac{L-1}{L}} \frac{1}{L} = \frac{\ln \frac{1}{L}}{\ln \frac{L-1}{L}} < \frac{\ln \frac{1}{L}}{-\frac{1}{L-1}} = L \ln L - \ln L < L \ln L$$
$$\implies t < \log{(L \ln L)} < \log{(L \log L)} = \log L + \log \log L$$
$$\implies t = O(\log L) \ . \qquad (8.11)$$

Since the expected time for the algorithm to get from W_i to W_{i+1} is one iteration, and it converges faster when there are fewer errors, the sum of the expected times to cross each of the t thresholds, is an upper bound on the expected runtime of the algorithm.

This observation that the number of necessary hypermodules can be discovered automatically during optimization lends itself to more realistic cases considered in the remainder of this chapter, where very little about the underlying modular structure of a problem may be known a priori.

8.4 Integrated Algorithm for Hypermodule Mapping and Learning

This section presents a practical implementation of the system based on the observations above. First, a method of selection at the location level is presented. Then, this selection method is integrated into a complete stochastic algorithm for optimizing ψ while simultaneously learning the trainable parameters. The complete algorithm is then applied to a suite of problems in the subsequent three sections.

8.4.1 The Softmax Surrogate Fitness for Hypermodules

Section 8.3 showed that evaluating the performance of ψ at the granularity of particular block locations could in theory give a huge advantage in terms of convergence speed, especially when the number of locations per task is large. Evaluating performance at each location requires a utility metric, or *fitness*, associated with each location, and using metrics based on validation performance can only be applied at the granularity of tasks. The solution adopted is to have the model indicate its hypermodule preference directly through backpropagation, by learning a softmax distribution over hypermodules at each location. The approach is based on the same soft-merge module-mixing mechanism used in Chapters 6 and 7.

Say that at a given point in time, there are $Q + 1$ active mapping functions $\{\psi_q\}_{q=0}^{Q}$, each associating with each location a corresponding potential hypermodule $\psi_q(\ell)$. Through backpropagation, these hypermodules compete at each location by generalizing the parameterization of $\boldsymbol{B}_\ell$ (Equation 8.2) to include a soft-merge operation over hypermodules:

$$\boldsymbol{B}_\ell = \sum_{q=0}^{Q} \psi_q(\ell) \ \bar{\times}_1 \ \boldsymbol{z}_\ell \cdot \mathrm{softmax}(\boldsymbol{s}_\ell)_q \tag{8.12}$$

where $\boldsymbol{s}_\ell \in \mathbb{R}^{Q+1}$ is the vector of soft weights associated with the ℓth location that induces the probability distribution over hypermodules. Through training, when interpreted as a probability distribution, the learned value of $\mathrm{softmax}(\boldsymbol{s}_\ell)_q$ is the model's belief that $\psi_q(\ell)$ is the best option for location ℓ out of the current choices $\{\psi_q(\ell)\}_{q=0}^{Q}$. A visual depiction of this competition between

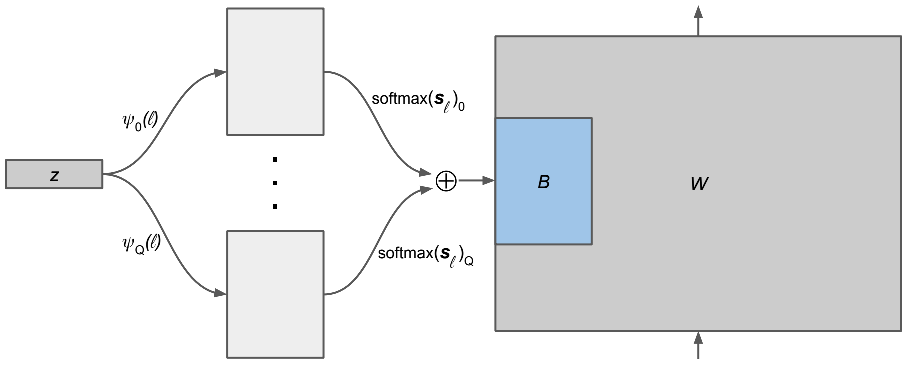

$\psi_0(\ell)$
$\psi_Q(\ell)$
z
softmax($\boldsymbol{s}_\ell$)$_0$
softmax($\boldsymbol{s}_\ell$)$_Q$
B
W

Figure 8.2: **Competition Between Hypermodules**. This figure depicts the competition between hypermodules for being selected for a particular location, i.e., to parameterize a particular block $\boldsymbol{B}$ within weight matrix $\boldsymbol{W}$ in the original model. Here, $\boldsymbol{z}$ is the context vector associated with $\boldsymbol{B}$ which is mapped to candidate parameter blocks by hypermodules $\psi_q(\ell)$. These candidate blocks are mixed by a soft sum based on the model's belief that each hypermodule is the best for this location out of the current $Q + 1$ options.

hypermodules is given in Figure 8.2. In particular, the estimate of the best hypermodule for the ℓth location at any given time is $\psi_{\text{argmax}\,\boldsymbol{s}_\ell}(\ell)$. Identifying this current best hypermodule at each location is a key component of the complete iterative algorithm introduced in the next section.

In order for the model to learn its hypermodule preferences reliably and quickly, a special learning rate α_s is associated with the learning of these soft weights. This rate is distinct from the learning rate α associated with the rest of the parameters in the model. In the experiments in this chapter, setting α_s two orders of magnitudes larger than the learning rate of the rest of the model yields reliable and satisfactory results.

8.4.2 Evolutionary Optimization of Hypermodule Mappings

Using the soft-merge hypermodule-mixing selection function described above, a complete algorithm can be constructed for optimizing ψ while simultaneously learning the model parameters. Every iteration the algorithm will sample a set of possible alternative hypermodules to use at some fraction of block locations and then train with backpropagation until the best of these alternatives is identified. At a high level, the algorithm is described as follows:

1. Initialize $U = \{\boldsymbol{H}_\ell\}_{\ell=1}^{L}$, $\{\boldsymbol{z}_\ell\}_{\ell=1}^{L}$, and $\psi = \psi_0$ with $\psi(\ell) = \boldsymbol{H}_\ell$, and initialize any other non-sharable model parameters.
2. Set $\psi_q := \psi_0 \ \forall \ q = 1, \dots, Q$, and set $\boldsymbol{s}_\ell = \boldsymbol{0} \ \forall \ \ell$.
3. Select $\lceil pL \rceil$ locations from $\{1, \dots, L\}$ without replacement ($p \in (0, 1]$).
4. For the ith of these $\lceil pL \rceil$ locations, select Q hypermodules at random as the new values of $\psi_1(i), \dots, \psi_Q(i)$. To randomly select a hypermodule, with probability ϵ create a new hypermodule, and with probability $1 - \epsilon$ select an existing hypermodule with probability proportional to the number of times it is currently used.
5. Train the joint model for a fixed number of steps, based on Equation 8.12.
6. Evaluate the model on the validation set for each task.
7. Set $\psi_0(\ell) := \psi_{\operatorname{argmax} \boldsymbol{s}_\ell}(\ell) \ \forall \ \ell$.
8. If not the final iteration, go to step 2.

9. Revert the state of the system (including parameters) to the state when the best validation performance was achieved, and continue training from there until convergence.

Additional details for these steps are given below:

Soft Weight Reinitialization. At every iteration, the soft weights S are all reinitialized to 0. This reinitialization associates with each hypermodule at each location the same initial softmax probability of success, and avoids aggregating bias over iterations.

More generally, to control the smoothness of evolution and reduce the shock of mutations, as in Chapter 7, we can set the initial probability of new candidates to something small. Then, our bias is towards not accepting new candidates, since the current winner has already demonstrated its relative value. Say, we would like to allocate probability α evenly over all Q new candidates for each location ℓ. Fix $\boldsymbol{s}_{\ell 0} = 0$. Then, for $0 < q < Q$,

$$\text{softmax}(s_\ell)_q = \frac{\alpha}{Q} \implies \boldsymbol{s}_{\ell q} = \ln \alpha - \ln Q - \ln(1 - \alpha) \tag{8.13}$$

So, by initialization $\boldsymbol{s}_{\ell q}$ to this value, we can achieve the desired behavior.

Selecting $\lceil pL \rceil$ Locations. In the highly simplified theoretical case discussed in Section 8.3, it is clear that all locations should be selected at every iteration, since that allows the fastest exploration and optimization of the static, noiseless problem. However, in the real problem, there may be noise in evaluation, and the changing use of hypermodules in other locations will have a dynamic effect on the performance of a particular hypermodule at the ℓth location. Therefore, in the experiments in this chapter, p is set to be less than one, in order to give more stability to the system to make the optimization process smoother. Although setting $p = 1$ did yield interesting and acceptable results in some cases, setting $p = 0.5$ yields reliably strong performance across all problems tried.

Selecting Hypermodules at Random. The distribution for selecting new hypermodules for a location is motivated by the analysis given in Section 8.3.2. Selecting hypermodules at random according to the distribution specified in Step 4 ensures that hypermodules that work well in many places can have their use spread quickly through the model. In other words, the distribution corresponds to the likelihood that each module is best for a given location about which there is no prior knowledge. This bias makes it possible to start with a parameterization equivalent to the original model, i.e., where each location is associated with a unique hypermodule, and then automatically collapse and group the set of hypermodules as regularities are discovered which can be exploited. The creation and selection of a new hypermodule with probability ϵ allows the model to avoid getting stuck, since the noisy and dynamic evaluation cannot guarantee that the current set of actively used modules is a superset of the optimal set of modules. In the experiments in this chapter, ϵ is always set to 10^{-4}. Taking a random action with probability ϵ to avoid local optima is related to the ϵ-greedy approach in reinforcement learning (Sutton & Barto, 1998).

Notice also that in this algorithm, if there is a time when a hypermodule is not actively used at any location, then it will have zero probability of ever being used again, i.e., it has been permanently deleted. Deletion makes sense as long as all potentially useful task models are considered when tabulating hypermodule usage: If a hypermodule is not useful anywhere, it should not be preferred over a new randomly initialized hypermodule. The maximum number of distinct hypermodules that can be in use at any time is L. By initializing the system with L modules, and allowing these deletions, the system can automatically determine the optimal number of modules during optimization.

Training. In all experiments in this chapter, backpropagation training is performed using Adam with the default learning rate of 0.001 (Kingma & Ba, 2014). The models were implemented using the PyTorch framework (Paske et al., 2017).

Evaluation Step. Notice that the evaluation Step 6 plays no role in the optimization of ψ. It is only used to track training progress, and for early stopping when returning the final model, i.e., parameters for all hypermodules, contexts, and $\psi = \psi_0$. This disregard for validation performance is in stark contrast to other methods that combine deep learning and evolution by making validation a crucial point of the optimization process: performing back propagation with the training data and basing fitness for evolution on validation performance.

Updating ψ_0. Note that the method for generating new hypermodule options at each locations does not forbid multiple copies of the same hypermodule to be used as candidates at a single location. If such duplication occurs, through training the model will converge to assigning equal probability to each copy. So, the score for this hypermodule is the sum of these equal probabilities, since this sum is the total belief that this hypermodule is best.

Final Training. The final training performed at Step 9 is designed to allow the model to perform final tuning without the shocks and noise of adding and removing hypermodules. This final tuning may or may not yield improved performance over that achieved during evolution; it is always useful to include for completeness.

8.5 Experiments: Synthetic Dataset

The first set of experiments is on a synthetic dataset on which it is easy to illustrate and analyze the behavior and performance of the approach. Once the intended behavior is confirmed in this dataset, the approach can be applied to real world problems. Summary characteristics of the problems considered in this and subsequent sections are given in Table 8.2.

Section	Dataset	Architecture	Core Layer	Raw Params.	Blocks
8.5	Synthetic	Linear	Dense	20×30	1×30
8.6	MNIST	Small Conv	Conv-2D	15K	49
8.6	IMDB	Small Conv	Conv-1D	171K	40
8.6	CRISPR	DeepBind	Conv-1D	7K	26
8.7	CIFAR-10	WideResNet	Conv-2D	0.59M	2268
8.7	Wikitext-2	Stacked RNN	LSTM	9.60M	4096
8.7	CRISPR	1layer_256motif	Conv-1D	1.64M	6400

Table 8.2: **Problem Summaries for Universal Reparameterization.** This table summarizes the problems considered in experiments for reparameterization by hypermodules. The experiments begin with small models, and are scaled up in two steps. Importantly, the problems cover qualitatively different model architectures (Fully-connected vs. 1D Convolutional vs. 2D Convolutional vs. LSTM), input domains (Vision, Text, Genomic, and arbitrary Unstructured Features), and target metrics (Accuracy vs. Perplexity vs. MSE). With these diverse characteristics, the generality of the method introduced in Section 8.4 can be demonstrated. Note that models with non-dense core layer type may also contain reparameterized dense layers. Also note that a big source of the difference in size between the NLP models and the other models is the high number of parameters in the word embedding (input adapter) layers.

8.5.1 Domain Descriptions

The synthetic dataset was first introduced as a toy problem for linear multitask learning approaches (Kang et al., 2011), and was used in subsequent work on modeling task overlap (Kumar & Daumé, 2012). The dataset is designed to capture an ideal situation in multitask feature learning, that is, methods that take a factorization approach. Thus, it is a perfect problem for our approach of factoring linear blocks into contexts and hypermodules.

The dataset consists of 30 linear regression tasks. Each task has the same 20-dimensional input space, and 1-dimensional output, so no input or output adapters are needed. Each task has 15 randomly generated training samples, and 50 randomly generated test samples. The outputs for these samples are generated by multiplying a random input vector by the true underlying parameter vector for the task. The goal for each task is to minimize the root mean squared error (RMSE) on the test set. The overall goal for the problem is to minimize the average RMSE aggregated over all tasks. The tasks are grouped into three groups of ten tasks each. The underlying parameter vector for tasks within a group differ only by a scalar factor. Since the number of training samples for each task is less than the input dimensionality, each task cannot be solved accurately without drawing the connection between its parameters and the parameters of other tasks in its group. The experiments in this chapter implement early stopping, and a random five of the 15 training samples for each task is withheld as validation data. This withholding of validation data was not done in previous work, and makes the setup slightly more difficult.

In this section, two versions of the problem are considered, one in which Gaussian noise has been added to all sample outputs, and one in which no noise is added. The noisy case corresponds exactly to the dataset introduced by Kang et al. (2011). The noiseless case uses exactly the same sample inputs and underlying parameter vectors, but does not add Gaussian noise to the outputs. Clearly, the noisy case is more difficult. The noiseless case is included to demonstrate the behavior of the approach in an ideal setting.

8.5.2 Model Description

As in previous work, the model for each task is a linear regression model parameterized by a single 20-dimensional vector. In the baseline single-task case, these vectors are parameterized and trained independently. In the reparameterization case, $c = 1$, and each task is reparameterized using a single hypermodule of dimension $1 \times 20 \times 1$, which is sufficient to capture the structural regularity within a group of tasks. So, per the algorithm described in the previous section, the system is initialized with 30 hypermodules, and over iterations it should converge to using only three, i.e., one for all tasks within each group. As an upperbound on performance, an Oracle comparison is included, in which ψ is fixed to the perfect mapping corresponding to the true groups.

8.5.3 Noiseless Domain Results

The convergence of the mapping to the true underlying grouping is summarized in Figure 8.3. Evolution quickly converges to the true underlying grouping. A more detailed visualization of how the mapping converges is given in Figure 8.4. This optimal convergence yields optimal performance with respect to test loss. The test losses for evolution along with comparisons are given in Table 8.3. Along with Oracle, Evolution achieves a perfect test loss, showing dramatic improvement over the other baselines. These results all show that the algorithm works as expected in this ideal domain. Importantly, these results show that the softmax surrogate fitness function is effectively serving

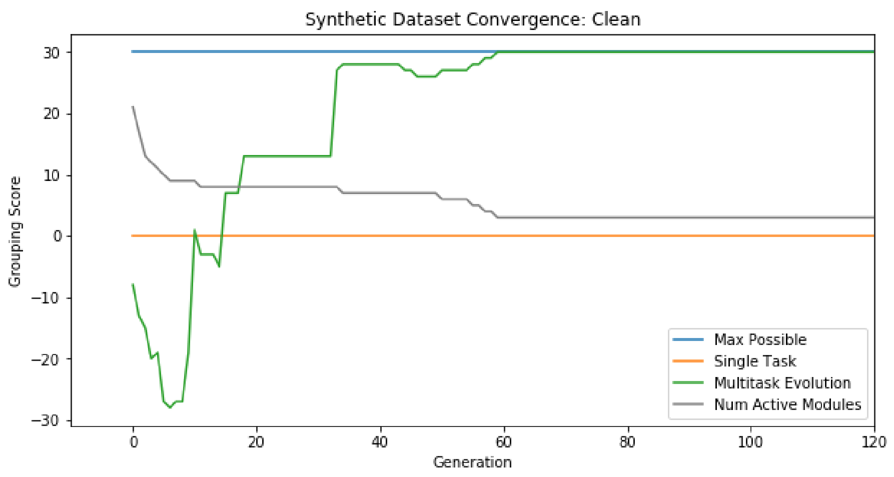


Figure 8.3: **Grouping Convergence.** This figure shows the convergence of the mapping to the correct underlying grouping for the noiseless version of the synthetic dataset. The group score for a task is 0 if its hypermodule is used by no other tasks; 1 if it is used by at least one other task and all such tasks are in its underlying group; and -1 otherwise, i.e., if the task uses the same hypermodule as a task not in its true group. The total grouping score is the sum over all task scores. Thus, the maximum possible score is 30. In this problem, evolution quickly converges to the optimal grouping and remains there indefinitely. Likewise, the number of hypermodules used by the system quickly converges to 3, the true number needed.

Method	Test MSE
Single Task	2.873
Random Search	1.537
Oracle	**0.000**
Evolution	**0.000**

Table 8.3: **Synthetic Noiseless Results.** Evolution achieves a perfect test loss equivalent to that of the perfect fixed (Oracle) grouping. This is a dramatic improvement over Single Task (i.e., separate hypermodule for each task), and Random Search, which is equivalent to evolution, except every iteration each hypermodule receives a random fitness instead of the softmax surrogate fitness.

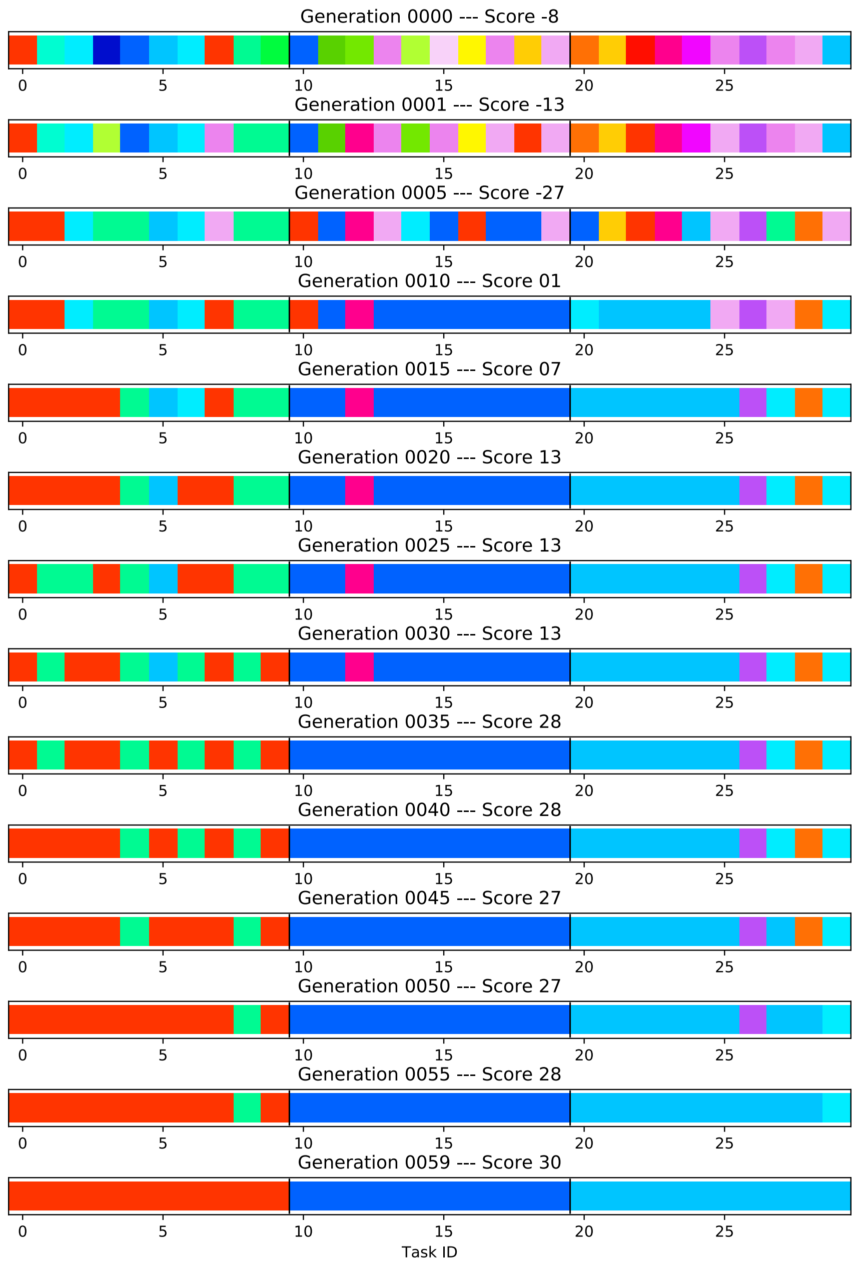


Figure 8.4: **Visualizing Convergence.** This series of images shows the progression of convergence of ψ on the synthetic dataset. Each color corresponds to a distinct hypermodule. The color shown at each location is the hypermodule currently in use for that task. At generation 59 and beyond, the model remains at the optimal solution indefinitely.

Method	Test RMSE
STL (Kang et al., 2011)	0.97
DG-MTL (Kang et al., 2011)	0.42
GO-MTL (Kumar & Daumé, 2012)	0.36
STL (ours)	1.31
Evolution (ours)	0.38
Oracle (ours)	0.37

Table 8.4: **Synthetic Noisy Results.** Evolution provides a substantial boost over the baselines in this domain, with results on par with the best known from the linear MTL literature. The fact that our Single Task Learning setup performs poorly and Oracle does not do quite as well as the best previous work, suggests the difference is due to a feature in the setup, e.g., using a validation set, or lack of L2 regularization. Note also that Evolution learns the number of groups automatically, whereas in the previous work it was fixed.

its purpose of determining the value of hypermodules at each location.

8.5.4 Noisy Domain Results

Now that the algorithm is shown to work as expected in the noiseless case, it will be tested with noise, which is more in line with real-world problems. The performance results for the noisy case are shown in Table 8.4. Again, evolution gives a substantial improvement over the baselines, on par with the best known results in the linear MTL literature for this domain. Note that this performance is achieved despite differences in the setup that make generalization more difficult: withholding data for validation and absence of L2 regularization.

8.6 Experiments: Cross-modal Multitask Learning (Scale-up 1)

In this section, the algorithm is applied to a set of standard benchmark problems with highly diverse modalities: The system jointly trains a vision model, an NLP model, and a genomics model. The models used for each task are relatively small, permitting rapid testing and analysis. However, by consisting of tasks from these disparate modalities, these experiments demonstrate the ability of the system to train across highly-diverse tasks and architectures.

8.6.1 Domain Descriptions

In this section, models are jointly trained for three tasks, each of a distinct modality.

MNIST. The dataset for MNIST is the same one used for experiments in Chapter 6. The main difference is that the problem is taken as a single multi-class classification task with ten classes, one for each digit, which is the standard setup for MNIST. Five thousand random samples are withheld from the training set for validation. The goal is to minimize multi-class error. Also, for consistency with previous work, and to prevent overfitting, the dataset is augmented by padding images to 30×30, and performing random crops during training (Ha et al., 2017).

IMDB. The IMDB movie review sentiment classification dataset is the same one used for experiments in Chapter 5 (Maas et al., 2011). The main difference is that the vocab size is capped at 5,000 words, and the maximum input sequence length is 400 instead of 80. The sequence length is longer because a convolutional model is used instead of an LSTM model, so increasing the input length has a negligible affect on computational cost on GPU, and there is less of problem of overfitting to long reviews.

CRISPR. The third dataset is a private genomics regression dataset. The dataset consists of the propensity of a CRISPR protein complex with a given guide RNA molecule to bind to specific locations in the reference human genome (Jung et al., 2017). The regression problem is to predict this *binding affinity* at each location. This is an important problem in genetic engineering and personalized medicine, since it gives the risk of using a particular protein complex and guide RNA on a particular human. When using the technology, there is one particular (target) location that is intended to be cut out by the CRISPR complex, so that this location can be edited. If the complex makes other (off-target) cuts, there may be unintended consequences. Predicting the binding affinity at off-target locations gives an assessment of the risk of the procedure. More interestingly, having an accurate regressor for this problem would enable a method of selecting an optimal guide RNA that minimizes risk for a particular genome. The experiments in this chapter focus on building an accurate regression model; the guide RNA design problem is left as future work, which will require further physical experiments.

In the dataset there are reported binding affinities for around 30 million base pair (bp) locations in the genome. From these raw binding affinities, a dataset is constructed by taking windows of length 200bp centered around sequences for which there is data. The data at each bp location is represented as a one-hot vector of length five: four for the four bases (A, C, T, G) and one for the null base (N), which indicates that there was no base pair reported for this location in the reference human genome. The goal of the model is to predict the binding affinity at the central location. With the motivation that big errors for this problem are much more significant than small ones, the loss to minimize is mean squared error (MSE). The samples for the 30 million total base pairs are split into training, validation, and test sets so that no sets have any of their input windows overlapping with another split. With this constraint, the samples are split uniformly at random. Approximately one million samples are withheld for validation, and one million for testing.

8.6.2 Model Description

Hypermodules. The context size for each location is set to $c = 4$, and the block size is set to $m \times n = 16 \times 16$, so each hypermodule has dimension $4 \times 16 \times 16$. These values were chosen to coincide with previous known satisfactory settings used in the case of hypernetworks (Ha et al., 2017). Also, for the task models described below, 16×16 is the largest block size that cleanly divides the weight kernels of all sharable layers.

MNIST: 2D Convolutional Vision Classifier. The model for MNIST is one used in previous work to validate the behavior of hypernetworks (Ha et al., 2017). The model has three layers of weights. The first is a 2D convolutional layer that maps the single-channel gray-scale image to a space with 16 channels. This layer is the input adapter (as described in Section 8.2.4). The second is a 2D convolutional layer with 16 output channels and a 7×7 spatial kernel. This layer is the only one that is decomposed into blocks for reparameterization; the kernel is decomposed into 49 blocks, each of which yields a mapping to all output channels that depends only on a view of a single spatial location. The third is a fully-connected layer that maps the output of the second to a softmax over digit classes. This layer is the output adapter.

IMDB: 1D Convolutional Text Classifier. The task model for the IMDB sentiment classification problem is based on the off-the-shelf Keras example convolutional model for this problem (Chollet et al., 2015). The model begins with a standard word embedding layer, which maps words in the vocabulary to embeddings of size 32. This first layer is the input adapter. This layer is followed by a 1D convolutional layer with kernel size 3 and 64 output channels. This convolutional layer is followed by a fully-connected layer with output size 64. Finally, a fully connected layer maps the output of the third layer to a single sigmoidal unit that indicates the probability of a review being positive or negative. This final layer is the output adapter. All layers aside from the input and output adapters have their weight kernels reparameterized by hypermodules. The reparameterization results in a total of 40 blocks for this model.

CRISPR: DeepBind Genomics Regressor. The model for the CRISPR binding regression problem is the DeepBind model, which was introduced as a deep learning solution to protein binding problems (Alipanahi et al., 2015). In form, it is similar to the convolutional model used for IMDB. The model begins with an embedding layer, that matches each of the five elements of the vocabulary (A, C, T, G, N) to 16-dimensional embeddings. The second layer is a 1D convolution with kernel size 24, and 16 output channels. The third layer is fully-connected with 32 hidden units. The final layer is fully-connected with a single output that outputs the predicted binding affinity. The middle two layers can be reparameterized by hypermodules, while the outer two are adapters. Thus, when reparameterized, the model is decomposed into 26 blocks. Notice that the 1D kernel size of 24 is much larger than is regularly seen in NLP, thus making this model qualitatively distinct from 1D convolutional text classifiers, although the topological components are the same.

8.6.3 Results

Performance results and comparisons for the tasks and models used in this section are given in Table 8.5. Although the single task baseline does best for MNIST, both versions of evolution outperform the baseline on IMDB, and multitask evolution outperforms the baseline on CRISPR. This indicates that

Method	MNIST % Err.	IMDB % Err.	CRISPR MSE
Single Task Baseline	**0.77**	14.34	0.1549
Intra-task Evolution	1.13	**13.90**	0.1557
Multitask Evolution	0.91	14.15	**0.1545**

Table 8.5: **Scale-up 1 Performance Summary.** This table reports the test set results for each of the three tasks and models used in the experiments in this section. Although the single task baseline does best for MNIST, both versions of evolution outperform the baseline on IMDB, and multitask evolution outperforms the baseline on CRISPR.

by sharing modules with other modalities, the CRISPR model learns functions that generalize better.

8.7 Experiments: Cross-modal Multitask Learning (Scale-up 2)

This section applies the algorithm to a larger cross-modal multitask learning problem. There is still one vision task-model pair, one text, and one genomic, but with thousands of blocks per task instead of tens. The domains and models are described below, followed by experimental results and analysis.

8.7.1 Domain Descriptions

CIFAR-10. The standard CIFAR-10 multiclass image classification benchmark problem is used (Krizhevsky, 2009). The dataset consists of 50,000 training images and 10,000 test images. Of the training images, 5,000 are randomly withheld for validation. Data augmentation is performed as in previous work, with random crops and horizontal flips (Ha et al., 2017).

Wikitext-2. Wikitext-2 is a standard language modeling problem (Merity et al., 2016). Preprocessing is performed as in previous work (Zaremba et al., 2014). The dataset consists of 2,551,843 total word tokens, 2,088,628 reserved for training, 217,646 for validation, and 245,569 for testing. Since the dataset is relatively clean in terms of language quality, all words are included in the vocabulary, for a total of 33,278 words. As is standard for language modeling problems, the target metric for the model to minimize is perplexity, which is the exponentiation of cross-entropy loss averaged over all test samples.

CRISPR. The CRISPR regression dataset is the same one used in the previous section. Because this is the first work that approaches this dataset with deep learning, it is included in both the small and large cross-modality experiments. This inclusion helps to give a more complete picture of what is possible on this problem.

8.7.2 Model Description

Hypermodules. The hypermodules used in this section have equivalent form to those used in the previous section, i.e., they have dimension $4 \times 16 \times 16$. Since the task models used in this section are larger, they will each be parameterized by many more hypermodule usages (around two orders of magnitude more overall).

CIFAR-10: WideResNet Vision Classifier. WideResNet has been established as an effective, relatively simple, and computationally efficient vision model in the last couple of years (Zagoruyko & Komodakis, 2016). It was developed specifically for CIFAR and ImageNet. Due to its relative simplicity, it is a good underlying model for experimentation, and has been used for previous work on hypernetworks (Ha et al., 2017). WideResNet defines a family of vision models, each defined by a depth parameter N and a width parameter k. The model considered in this section has $N = 6$ and $k = 1$, which has 40 layers and is known as WideResNet-40-1. This model is the smallest (in terms of parameters) high-performing model in the standard WideResNet family. This model was used in previous work on hypernetworks as well. As in the case of the 2D convolutional vision model for MNIST in the previous section, the first and last layers of the model are reserved as input and output adapter layers, respectively. All remaining intermediate convolutional layers can be reparameterized by hypermodules, yielding a total of 2268 blocks.

Wikitext-2: Stacked LSTM Language Model. The model for Wikitext-2 language modeling is the standard stacked LSTM language model (Zaremba et al., 2014). This standard model has one main parameter, LSTM size. In general, increasing the size improves performance. The standard LSTM sizes are 200, 650, and 1000. In order to make the LSTM weight kernels divisible by the output dimension of hypermodules, the experiments in this section use an LSTM size of 256. As in the case of IMDB, the model begins with a word embedding layer, mapping each word in the vocabulary to a vector of size 256. The embedding layer is followed by a stack of two LSTM layers, each of size 256. The second LSTM layer is followed by a fully connected layer mapping its output to a softmax over the vocabulary. The LSTM layers can be reparameterized by hypermodules, yielding a total of 4096 blocks.

CRISPR: Large 1D Convolutional Genomics Regressor. The model used for the CRISPR dataset in this section was developed as an extension of DeepBind as part of a more exhaustive architecture search for deep models for genomics problems (Zeng et al., 2016). That work found that simply widening the DeepBind model to 128 filters instead of 16 gave the best results; the id for the model is "1layer_128motif". Widening this model even further to 256 in the experiments in this section, and increasing the number of units in the subsequent fully-connected layer to 256 improved performance even further in the case of the CRISPR dataset, because of its scale and noise. So, the model, called "1layer_256motif", has equivalent topology to DeepBind, but with larger layer sizes. Specifically, the output size of the embedding layer, the number of output kernels in the convolutional layer, and the output size of the first fully-connected layer are all increased to 256. The increased capacity is useful for fitting this large and noisy dataset. There are 6400 blocks in the decomposition of this model.

8.7.3 Results

For each of the task-architecture pairs, a chain of comparisons were run, with increasing generality: a baseline run that simply trained the original architecture; an intratask run that applied MUiR evolution via block decomposition of a single model; a cross-modal run across a pair of tasks for each of the two remaining tasks; and a cross-modal run across all three tasks. The performance in the tasks considered in this section is given in Figure 8.5. The results are similar to those in the previous section. In particular, evolution outperforms the baselines except on the vision problem. Although cross-modal sharing does not improve performance on CIFAR, the performance in the other tasks improve whenever they are trained jointly with CIFAR. This may be because the pseudo-tasks in the CIFAR WideResNet model are more diverse than those in the other models. That is, the DNA and text models consist of relatively few parameter tensors, within which blocks are interchangeable a priori, whereas WideResNet has more qualitatively separate tensors due to its substantial depth. Such diversity can create more opportunities for regularities to be exploited by the other models, but can make it more difficult to find the correct regularities for improving WideResNet.

Another possible reason for performance degradation on vision problems is that the models used are relatively small versions of the vision models that are best known for the tasks. That is, WideResNet-40-1 is already the most compact high-performing models in the WideResNet family, which means all of the parameters in the original model are most likely critical, making it more difficult to make progress in optimizing ψ without destroying critical information that is not replicated elsewhere in the model.

Figure 8.6 shows the number of modules used by each combination of

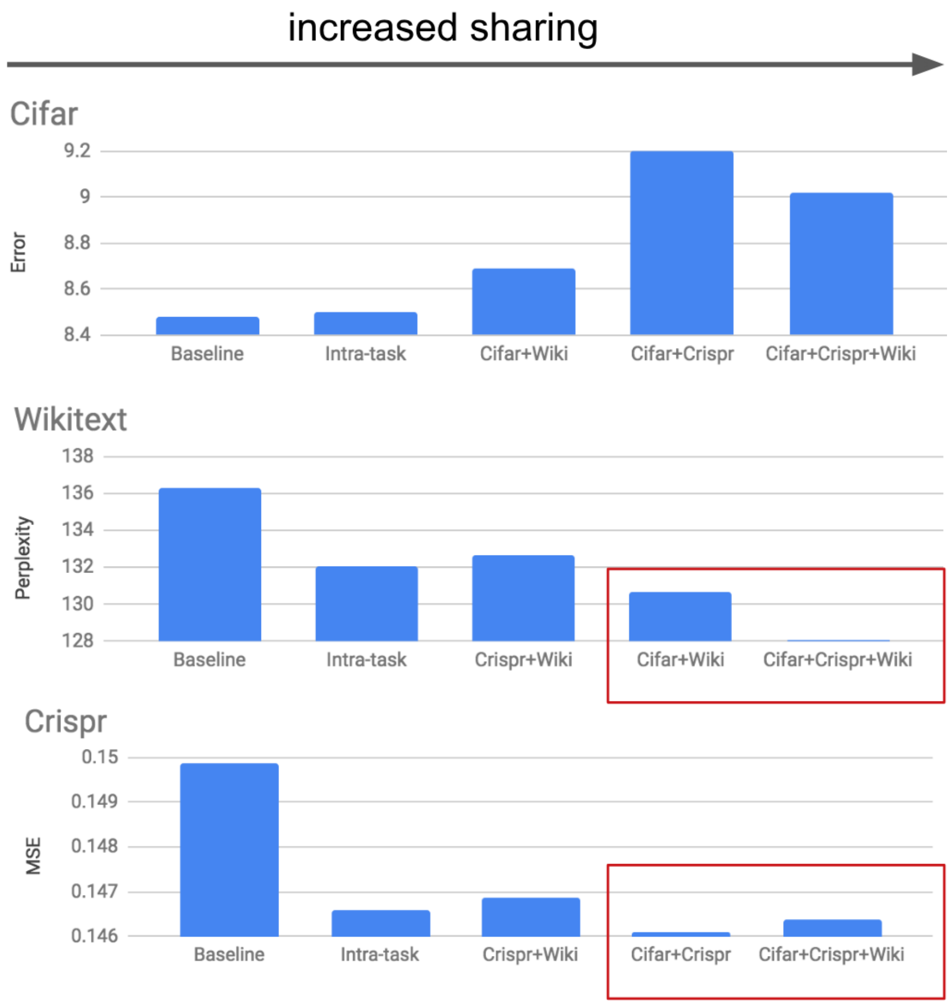


Figure 8.5: **Scale-up 2 Performance Summary.** This figure shows the performance in each task across a chain of comparisons with increased module sharing. For each plot, a lower score is better, i.e., classification error, perplexity and mean squared error (MSE). Interestingly, CIFAR performs worse during cross-modal sharing, while both Wikitext and CRISPR perform better whenever they share with CIFAR. This may be because WideResNet contains a higher diversity of pseudo-tasks than the other two models, which provides greater opportunity for improving the other tasks, but also is more difficult to optimize. Overall, the ability of MUiR to improve performance, even in the intra-task case, indicates that it is able to exploit underlying regularities across pseudo-tasks.

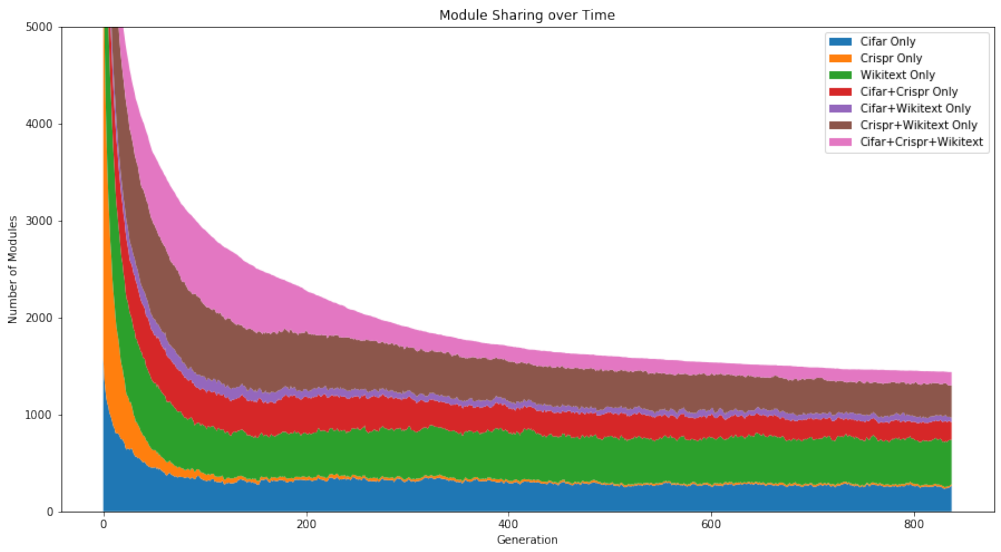


Figure 8.6: **Module Sharing Over Time.** The number of modules shared by each subset of the tasks is shown for a single run of multitask evolution. The total number of modules used is the sum over all these sets. Interestingly, the number of modules used by the CRISPR module alone quickly becomes very small, while the number used by CIFAR and Wikitext stays relatively large. Overall, the number of distinct active modules in each of subsets stabilizes as ψ is optimized.

tasks over time. The proportion of distinct hypermodules in each of the subsets converges as ψ is optimized. Interestingly, the number of modules used by the CRISPR model along quickly becomes small, while the performance of the model improves.

Looking more closely at how module usage evolves over time, an interesting phenomena was uncovered: *The emergence of a supermodule.* That is, there is a single module whose number of usages grows to dominate a significant fraction of all of the tasks. The progression of usages of this single module

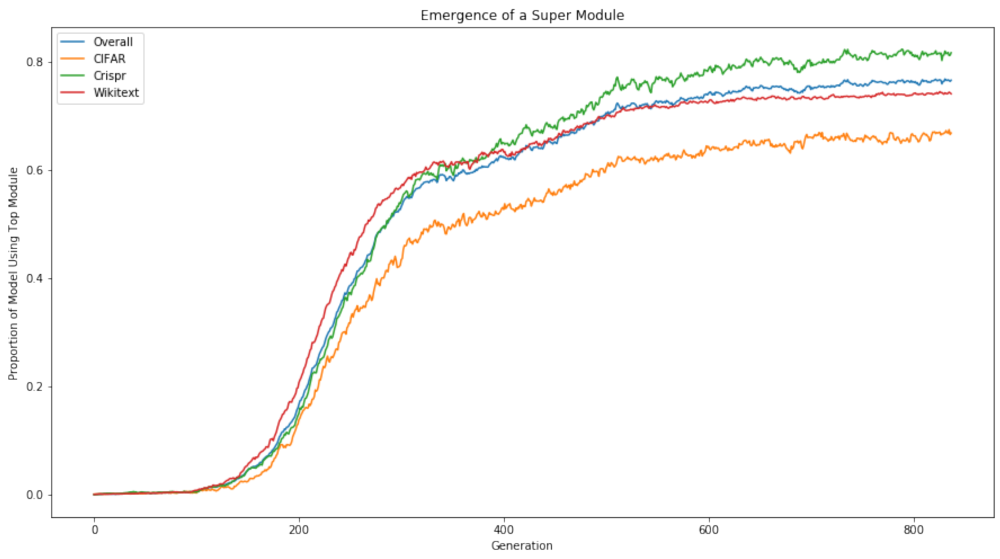


Figure 8.7: **Emergence of a Supermodule.** The number of locations in which the most used hypermodule is used is shown over time for a single evolutionary run. Remarkably, a supermodule emerges that quickly spreads to dominate a significant fraction of the locations in the entire model. The fact that so much of the joint model can be parameterized by this single hypermodule suggests that there is a surprisingly regular structure across block pseudo-tasks. The reparameterization only requires assistance from a relatively small number of more specific hypermodules to yield good performance on each task.

is shown in Figure 8.7. Since each of these locations where the supermodule is used share the same hypermodule parameters, they only incur additional parameters in their contexts, i.e., they yield a 64$\times$ reduction in parameters at their location. Having this substantial portion of the model yield this reduction implies that the model is substantially compressed. Aside from the parsimony and structure it implies, it could give practical advantages in cases where models need to be small, e.g., on mobile devices.

One important direction for future work is to analyze what kinds of structure models end up capturing. In the case of MUiR, each module can be viewed as a set of c matrices. Thus, standard matrix analysis techniques, such as spectral methods, would be a good starting point for this analysis. It will be interesting to see if, through training towards many diverse pseudo-tasks, the modules converge towards crisp properties like those hand-coded into convolutional and LSTM layers.

8.8 Conclusion

This chapter introduced a general framework for sharing information across highly diverse architectures and tasks through block decomposition and reparameterization by hypermodules. A stochastic algorithm was introduced for optimizing the mapping of hypermodules to pseudo-tasks given such a decomposition. The behavior of the algorithm was explored in a suite of domains. The conclusion is that sharing between qualitatively distinct settings is possible, and can be facilitated by generic decomposition and evolution. This is a first step towards future multitask and lifelong learning systems that accumulate and refine large libraries of useful reusable knowledge over a range of diverse problems.

Chapter 9

Discussion and Future Work

This chapter discusses some of the core themes that were presented throughout this work, along with directions for future work. First, the contrasting advantages of the various systems are reviewed and their most promising areas of application identified. Second, tradeoffs that emerged across the various systems are reviewed, and ideas presented for how to harness their complementary advantages better. Third, an ecological perspective arising from the systems is discussed, including some directions for improving these based on these perspectives. Fourth, the potential benefits of offline model analysis are discussed. Fifth, possible extensions to lifelong learning are described, with ideas for how to address the new issues that arise in this more dynamic setting.

9.1 Outline of Systems and Applications

The systems presented in this dissertation were developed in a process that explored methods for discovering deep multitask modules by increasing the general applicability of modules step-by-step. A high-level characterization of this progression based on technical mechanisms is given in the taxonomy of Table 10.1. Though the taxonomy indicates that MUiR (Chapter 8) is indeed the technical capstone of this work, MUiR does not strictly subsume the other systems. In fact, for each system there is some space of applications for which it would be preferred over the others.

GRUSM (Chapter 4) improved performance by transferring knowledge across diverse sets of video games, achieving results on par with humans and DQN (Mnih et al., 2015a), the state-of-the-art at the time. Since its implementation is derived from ESP (Gomez & Miikkulainen, 1997), GRUSM is a natural fit for sequential decision-making problems with relatively low intrinsic dimensionality. It is therefore a good choice for tasks of this form.

For problems where larger and more complex models are needed, subsequent systems are a better choice. The first one of these, soft ordering (Chapter 6), improved performance over previous state-of-the-art deep multitask learning approaches when tasks were drawn from the same high-dimensional domain. However, the dramatic improvement of CTR (Chapter 7) over soft ordering, while preserving efficiency, suggests that CTR should always be preferred over soft ordering, if the only goal is performance. Similarly, if an order of magnitude more compute is available, CMTR (Chapter 7) should always be preferred over CTR, as it demonstrated the value of optimizing module topologies. That said, soft ordering still has practical value in illuminating how information can be shared across depths and tasks in a joint model; its simpler uniform architecture is amenable to such analysis.

CTR and CMTR are powerful design systems especially in the case where a near-optimal architecture is not known *a priori*. However, like soft ordering, they can only provide value when each subnetwork module can be applied in many locations. This constraint discludes the exploitation of some state-of-the-art design patterns, such as deep vision models in which the number of filters increases with depth. PTA (Chapter 5) skirts this issue; it can be applied directly to any deep learning architecture, and is especially valuable when a single architecture that contains such design subtleties is already known to be near-optimal across a tasks. This value was demonstrated by achieving state-of-the-art performance on the CelebA dataset, using the highly-tuned and complex InceptionResNet-V3 architecture (Szegedy et al., 2015) for the underlying model. PTA was also shown to improve performance of natural-language LSTM models. In contrast, it is unknown whether the methodology of CTR can be practically extended to recurrent layers.

Still, PTA, CTR, and CMTR can only be applied to sets of tasks drawn from the same domain. Their fundamental building blocks are modules whose functional specification (e.g., input-output shapes and spatial semantics) are highly dependent on the problems being solved, e.g., graphs of fully-formed convolutional layers or LSTMs. MUiR (Chapter 8) provides a mechanism for sharing modules across such diverse architecture types, and thus across tasks from different domains and different modalities. The value of this mechanism is demonstrated by improving performance through sharing across vision, natural language, and genomics tasks. Thus, MUiR is the go-to system if this level of general sharing is desired. In practice, such generality may be required when a problem with a completely new modality arises, e.g., from a newly designed geosensor that collects a unique kind of climate data. Even without auxiliary datasets of the same modality, MUiR can be used to provide this new problem with a prior for what successful solutions to real world problems look like, i.e., they are composed of the modules MUiR has collected.

Note also that, in principle, MUiR should be preferred over PTA, even when tasks are drawn from the same domain. Like PTA, MUiR can directly take advantage of state-of-the-art architectures, and has much greater flexibility in how knowledge can be shared. However, PTA is a much simpler system to apply effectively, as shown by its off-the-shelf application to a broad array of problems. Investigating the application of MUiR to intradomain multitask learning, and generally working to make MUiR a more robust tool, is an important area of future work. Similarly, a direct comparison of the systems across a comprehensive set of applications would help to clarify the assessment of their advantages.

9.2 Tradeoffs

In the development and evaluation of the systems presented in Chapters 4-8, several underlying tradeoffs became apparent. In particular, two umbrella tradeoff themes were identified: *mixing* vs. *matching* and *matching* vs. *restructuring*, along with a third cross-cutting theme: *computational cost.*

Mixing vs. *matching* is the question of whether a combination of modules or a single module should be applied at each pseudo-task location. In soft ordering (Chapter 6), CTR (Chapter 7), and CMTR (Chapter 7), the models include learned mixing of module outputs. In contrast, in GRUSM (Chapter 4) and PTA (Chapter 5), single modules are applied in fixed locations. Interestingly, in MUiR (Chapter 8), the inclusion of a task-specific context vector at each location in MUiR can be interpreted as a compromise between mixing and matching. Mixtures are learned in order to select for the best module at each location, but once the best is selected, the rest are discarded. Thus, the context vector at each location effectively defines a mixture of linear components *within* each hypermodule.

Allowing for module mixtures can make the required set of modules much more compact, since mixtures add a level of combinatorial expressivity to the model. This expressivity can capture any inherent overlap between the functionality required across multiple locations. Such overlap has been exploited similarly in the setting of linear multitask learning by representing each task by a mixture of shared bases (Kumar & Daumé, 2012), yielding substantial improvements over approaches that find strict task groupings (Kang et al., 2011). This baseline approach that is analogous to the case of matching a single module to each pseudo-task location. However, additional issues arise in the deep learning setting. For one, there is a computational overhead to applying multiple modules at each location. In the case of soft ordering, the cost scaled linearly with the number of modules. This cost may be negligible in the case of a linear model, or acceptable in the case of ensembling already-trained deep models (Dietterich, 2000), but during the training of large deep models it can quickly become overwhelming. Another issue is possible leakage of information from shared modules to unshared task-specific parameters. Though there may only be one additional task-specific parameter per module application, when the number of modules applied at a location becomes large, care must be taken to ensure that the model is taking advantage of the shared structure, and not simply exploiting the task-specific scaffolding. On top of the complexity incurred by task-specific parameters, mixtures may make results more difficult to interpret than a simple module-to-location mapping. Of course, if mixtures are indeed fundamentally useful, their utility could be discovered automatically, by a system that supports sufficiently expressive topological exploration.

Matching vs. *restructuring* is the question of whether such topological alterations will be necessary for future systems that improve upon the work in this dissertation. GRUSM expanded the topology of the target task model by adding a module that the model could choose to exploit if it could find a beneficial way to do so. Similarly, CTR allowed the incremental expansion of task models, by iteratively adding new module applications and letting the model decide how much to use the new module, via a learned soft mixture. CMTR generalized CTR by evolving the topologies of the modules themselves. On the other hand, soft ordering, PTA, and MUiR are all based on fixed topologies, derived from the underlying model architecture for each task. When architectures can be optimized as well, better performance is possible, as shown by the dramatic performance gains of CTR and CMTR over soft ordering. Because they are used in a greater diversity of locations, the modules learned in CTR can even be said to be more general than those in soft ordering. However, architecture search adds additional layers of complexity, both computationally and with respect to system dynamics. The improvements induced by this complexification may be orthogonal to progress on other system aspects. For example, PTA focused on one key training mechanism of the more complicated CTR and CMTR methods, and was still able to yield state-of-the-art performance on a well-established benchmark. Similarly, MUiR assumes the problem is specified by a set of tasks and corresponding architectures, which the system then reparameterizes without making topological alterations. One benefit of using fixed architectures is that orthogonal improvements to the state-of-the-art can be directly incorporated into the system.

One drawback of MUiR is that the given set of architectures must all be decomposable into equally-sized chunks of parameters. Such decomposability is not an unreasonable assumption for deep models, but allowing more flexible module shapes, perhaps facilitated by structural search methods, could performance further. It remains to be seen whether future methods can exploit this kind of decomposed modularity within existing architectures, or whether the most natural form of generic modules is not something that humans have incidentally designed for individual tasks. Fixed architectures may be fine for now, but eventually architectural adaptation will be necessary to push performance. Further, to achieve this performance without incurring unacceptable computational overhead, it will be necessary to include modern search methods that complement those used in this dissertation. There is a wealth of recent evolutionary techniques that could be used to improve these methods. For example, asynchronous evolutionary approaches (De Jong, 2006) can be used to accelerate coevolution across pseudo-tasks, online hyperparameter adaptation (Eiben et al., 1999) can be used to increase the robustness of long-running systems, and multiobjective (Deb, 2015) approaches can be used to handle architectural bloat and enforce sufficient module generality, by treating each of these as separate objectives.

Overall, by looking at where tradeoffs had to be made in the design of the systems in this dissertation, the biggest pain points are identified, each of which sets a direction for future work.

9.3 An Ecological Perspective

The systems developed in this dissertation interleave gradient descent and evolution, and thereby capture properties of an artificial ecology more than most other machine learning methods. An ecology is defined by how independently-motivated organisms interact and adapt within an environment, with a particular focus on how these behaviors affect the distribution of resources and the success of competing organisms over time. From an ecological perspective, shared modules represent a set of limited environmental resources that are constantly shaped and exploited by the optimization for each pseudo-task through joint training. Pseudo-tasks with a shared fitness objective constitute a species, and compete within their species by trying to exploit the resources they have access to as well as possible. To increase their competitive advantage, pseudo-tasks can develop either mutualistic or competitive relationships with other species, e.g., by harmoniously improving shared resources or by fighting to monopolize resources for their own needs. Furthermore, the fact that candidates optimize the shared resources through joint training points to a more optimistic view of evolution than “survival of the fittest”. In particular, it suggests that all individuals can have an important impact on the state of the world, even if they never reproduce.

The ecological perspective has already been noted in Chapter 7 with respect to CTR. In CTR, candidate models compete to be the best in each task. These candidates must also effectively share a fixed and relatively small number of modules with the models from the other tasks. As task models complexify through evolution, the environmental relationships within and across tasks also become more complex. By optimizing module architectures in an outer loop, CMTR extends CTR by searching for a class of resources that will lead to the most fruitful ecological development. PTA also has similar ecological dynamics to CTR, but with the shared resources constrained to a single shared encoder. MUiR takes speciation to the extreme, with competition within a species occurring at a sublayer level. Thus, the systems capture a range of ecological dynamics.

It may then be possible to apply models from theoretical ecology (Gurney & Nisbet, 1998; May & McLean, 2007), to see if they can help to model the dynamics of such systems, since combinatorial interactions can make them difficult to scrutinize. Whether from ecology or somewhere else, adapting methods from other disciplines that analyze interaction networks is an interesting direction to pursue. Insights from such systems may help develop methods that support more healthy and robust ecologies. For example, they may help in interpreting the significance, advantages, and drawbacks of the emergence of a supermodule that was observed in the experiments in Chapter 8, e.g., does it represent the well-deserved incremental domination by a superior species, or the epidemic spread of a destructive virus?

Closer to home, there are other systems within evolutionary computation that also include some of these ecological properties explicitly. For example, several neuroevolution systems have been developed to model coevolution of agents that share resources in a single environment, and to investigate the competitive or cooperative properties that emerge (Rawal et al., 2010; Nitschke et al., 2012). The emergence of specialization is particularly indicative of functional modularity (Nitschke, 2005). For example, the specialization of robots in a construction task resembles an organic assembly line (Nitschke et al., 2012). This specialization can be interpreted at a semantic level, with distinct agents taking on complementary roles.

Going in the other direction, when interpreted from an ecological perspective, the systems developed in this dissertation could be used to advance the study of the effects of modularity on ecological dynamics. For example, in the cases of PTA and MUiR, any set of tasks and corresponding fixed architectures instantiate a unique environment for investigating ecological dynamics. Since tasks and architectures are plentiful, general applicability yields a near-limitless supply of distinct environments.

9.4 Offline Analysis of Single-task Models

The systems developed in this dissertation discovered modularity while learning the structure of the modules themselves. By learning modules from scratch, the systems avoid bias towards the kind of modularity that may be found in trained single-task models. However, introspecting such models may lead to better understanding of what kinds of modularity can be expected to naturally emerge from these systems, and at what scales modularity can be expected to be most useful.

The systems in this dissertation operate under the assumption that significant sharing via modularity is possible across a given set of tasks. Estimates of the theoretical benefit of such sharing could be computed through offline analysis that attempts to modularize trained single-task models, including indications of how modules can be shared across tasks. For example, as in MUiR, the parameters of each model could be broken into equally-sized linear maps, which are then clustered into K groups based on their functionality, e.g., using a Schatten norm. One result of such analysis would be a lower bound on the amount of compression that sharing is able to provide without degrading performance. Such analysis could also be used to initialize a pool of modules from this already learned structure, which may provide a computational boost by skipping the initial online module specialization process. Such analysis techniques would rely on methods for accurately and efficiently comparing the functionality of high-dimensional nonlinear functions.

Offline analysis has already yielded positive results in identifying modularity in neural networks (Velez & Clune, 2016; Huizinga et al., 2018). In that work, subnetworks are shown to exhibit modular behavior within the networks in which they are trained. These methodologies must be extended to analyze how much such modules have the potential to be used effectively for problems for which they were not trained. One challenge for such techniques is the *permutation problem* in neural networks, where models represent the exact same function by combinatorially many distinct parameterizations (Yao, 1999). One approach to overcoming this issue could be to compare functions in the space of behaviors, i.e., based on what the functions actually do during evaluation. The power of this approach has been demonstrated through behavior-based evolutionary techniques (Lehman & Stanley, 2011a; Mouret & Doncieux, 2012; Pugh et al., 2016; Meyerson & Miikkulainen, 2017). For example, in the computer vision domains used in experiments in this dissertation, modules can be compared via the vectors of predicted classes they induce across different pseudo-tasks.

The systems in this dissertation avoided the permutation problem by initializing modules from scratch and jointly training them across all tasks, so that their semantics were preserved in each of their applications (e.g., see Section 8.2.4). Aside from the theoretical insights and computational advantages of offline analysis, there may be scenarios in which we do not have the luxury of training everything from scratch, e.g., when attempting to join sets of tasks and modules into a unified system. Such a scenario is related to the setting of lifelong learning discussed in the next section.

9.5 Towards Lifelong Learning of Highly Diverse Tasks

For simplicity, the systems introduced in this dissertation focused on multitask learning, where the data or simulators for training each task are always available. Once the discovery of functional modules is reliable and well-understood, it can be extended to lifelong learning systems (Thrun & Pratt, 2012; Brunskill & Li, 2014), in which new tasks may appear over the lifetime of an agent that can always keep learning and improving itself. Lifelong learning has a well-trod history in classical machine learning methods, has recently been extended to the deep learning realm (Silver et al., 2013; Tessler et al., 2017). It is ripe for innovation via an approach of modular ecologies.

Because they are used in different ways at different locations for different tasks, the shared modules trained in the systems in this dissertation have learned more general functionality than layers trained in a fixed location or for a single task. A natural hypothesis is that they are then more likely to generalize to future unseen tasks, perhaps even without further training. This ability would be especially useful in the small data regime, where the number of trainable parameters should be limited. For example, given a collection of these modules created from a previous set of tasks, a model for a new task could learn how to assemble these building blocks effectively while keeping their internal parameters fixed.

Maintaining module diversity is also important in the lifelong learning setting. If all modules are too similar in some important respect, e.g., they all include some functionality that is overfit to previous tasks, they may not generalize well to new problems. Such diversity can be critical when the distribution from which tasks are drawn over time can change drastically. If the system can maintain sufficient module diversity and functional coverage, it will be in a position to quickly adapt to such changes. One approach to achieving this robustness could be to apply recent advantages in behavior-based evolutionary methods, i.e., novelty search (Lehman & Stanley, 2011a) and related methods (Mouret & Doncieux, 2012; Pugh et al., 2016), which balance performance with functional diversity of solutions. Such diversity mechanisms require a metric for computing the distance between individual behaviors. As suggested in the previous section, in the case of functional modules, this metric could be based on some Schatten norm or vectors of model predictions.

It is also in this lifelong setting that the modular approach can dramatically set itself apart from the shared feature extractor approach. Consider, for example, the most ambitious alternative approach to MUiR for deep multitask learning across diverse tasks with vastly different modalities: One Model to Learn them All (Kaiser et al., 2017). In this alternative approach, there is a single shared model with separate encoders and decoders for each modality. To ensure the information is maximally sharable across tasks, the model is autoregressive. In particular, to get a prediction for a particular task, produces output for all modalities and reintegrates that output back into the core model recurrently. This process proceeds over several recurrent steps until the output for the target modality is finally read off as a prediction. By reintegrating knowledge from all modalities, using a single giant model of this form is indeed an elegant solution to the problem of preventing information from leaking into task-specific components. However, such an approach has inherent scalability issues: As more tasks are added and the number of modalities increases, the model becomes more and more computationally expensive to operate, incurring additional computational overhead for each task. In other words, because the system is monolithic, it is expensive to adapt it to new purposes. The modular approach is especially attractive in light of these drawbacks. By collecting a compact set of modules that can be assembled in different ways to solve different problems, the assembled model for each task can remain at a size appropriate for that task, and yet the combinatorial applicability of modules can enable a broad array of problems to be solved.

9.6 Conclusion

This Chapter reviewed the systems presented in this dissertation in various contexts. The systems were contrasted based on their areas of application, leading to the identification of core tradeoffs that must be considered for future systems. Avenues for better understanding of modularity and pseudo-task interactions were then discussed from the angles of ecological analysis and offline analysis of already-trained models. Based on the potential of this future work and the success of the existing system, an extension to lifelong learning was suggested, where the modular deep learning approach may find even greater success. Overall, the systems embody a rich set of interaction dynamics that yield endless streams of data If this data is properly analyzed and understood, the insights can be used to improve system performance and reliability.

Chapter 10

Conclusion

This chapter reviews the main contributions of this dissertation, and then concludes with a big picture perspective on where we now stand with respect to using deep multitask learning to discover multi-purpose modules.

10.1 Contributions

Chapter 3 introduced a multitask learning framework for situating methods that discover multi-purpose functional modules. This framework clarifies what we mean by multi-*purpose* module, by defining the notion of a *pseudo-task*. Systems can then be evaluated and compared based on what pseudo-tasks their modules solve, and how they solve them. Within this framework, a progression of six systems was introduced, with increasing levels of module sharing. Table 10.1 compares these systems across five qualitative dimensions: Whether discovered modules are universal, how many modules the system optimized, how the system is optimized, the functional form of the modules, and whether the system modifies the underlying model topologies. Considering the systems along these dimensions yields a simplified but informative view of the progression and interconnectedness of the differing approaches. In particular,

Approach	Universal	Count	Optimizer	Module Form	Topology
GRUSM	Yes	One	Evolution	Layer	Altered
PTA	No	One	Interleaved	Subnetwork	Fixed
Soft Ordering	No	Multiple	Gradient-based	Layer	Fixed
CTR	No	Multiple	Interleaved	Layer	Altered
CMTR	No	Multiple	Interleaved	Subnetwork	Altered
MUiR	Yes	Many	Interleaved	Projective	Fixed

Table 10.1: **Taxonomy of Systems.** This table displays the six systems developed in this work in a simplified taxonomy along five salient high-level dimensions: *Universal* indicates whether the discovered modules are intended for application across arbitrary tasks; *Count* indicates the quantity of modules; *Module Form* gives the functional form of modules in the system; *Optimizer* describes how the system is trained, i.e., by evolution, by gradient descent, or by interleaving the two; *Topology* indicates whether the system alters the topology of task models when incorporating new modules. In particular, notice that GRUSM constituted an initial attempt at uncover universally applicable modules; an ambition that was finally revisited by MUiR, once more refined mechanisms were developed.

from GRUSM to MUiR mechanisms were designed and joined for creating a suite of module discovery systems.

To investigate the inherent generality of neural network modules, Chapter 4 developed GRUSM, a general evolutionary method for reusing trained neural network modules. This system was implemented as a generalization of ESP, by adding subpopulations for evolving *how* to use previous structure in the new task. Since it is derived from ESP, the system is a natural fit for sequential decision-making domains. The performance of the system was exemplified in general video game playing, where it was shown that reuse helped more when the source was more complex, with results on par with the concurrent state-of-the-art. The observed generality potential of modules motivated the development of systems that explicitly aim to make modules more general. The value of complexity also motivated a turn towards larger, deeper models and intrinsically higher-dimensional domains.

Chapter 5 developed a system, PTA, for making a single large encoder module more general, by including multiple decoders for each task, thereby training this module towards more pseudo-tasks. To improve generalization, pseudo-tasks were optimized by an evolutionary process interleaved with gradient descent. This mechanism was theoretically shown to create more expressive training dynamics. In experiments, this method alone was able to yield performance gains for LSTM and convolutional models, in the case of multitask learning and when only a single task was available, including achieving state-of-the-art performance on the CelebA dataset. In this simplified system, the trajectories of pseudo-tasks could be visualized, and were shown to coincide with the intuition behind their behavior. Overall, the system showed broad value and applicability in its ability to use any underlying deep architecture. The success of PTA motivated the development of systems that further increase the general applicability of shared modules.

In Chapter 6, soft ordering, took on this motivation by training multiple modules (here, layers) each at all possible depths for each task in an end-to-end deep learning system. While learning module parameters, the system simultaneously learns *how* to assemble modules for each task by learning a soft mixture of modules at each pseudo-task location. This mechanism is theoretically motivated by the increase in expressivity achieved by assembling modules in different ways for different tasks. The value of this flexibility was demonstrated by improving performance across seemingly unrelated UCI tasks. The power of the method was then demonstrated in deep convolutional networks, giving improvements over existing deep multitask learning methods. Qualitative evidence that the resulting modules do indeed learn functional primitives was observed.

Chapter 7 addressed inherent scalability issues of the approach developed in Chapter 6 by combining its advantages with those of evolutionary architecture search. First, CTR was introduced as a method that incrementally grows a distinct topology of soft module mixtures for each task, so that the correct complexity for each task can be discovered automatically. This architecture search was made practical by *coevolving* task topologies while *interleaving* evolution with joint gradient descent *across all candidates*. CTR was then generalized to CMTR, which increases flexibility dramatically by evolving the topology of the modules themselves using CoDeepNEAT in an outer loop around CTR. The power of CTR was demonstrated by achieving state-of-the-art results on the Omniglot multitask dataset, and CMTR provided an additional significant improvement. Evolution reliably discovered diverse topologies, and these topologies were similar across multiple runs, matching intuition about how specialization can simultaneously manifest at different complexities.

Chapter 8 introduced a final system, MUiR, that scales the system ideas in Chapter 6 to support a much more general form of sharing, returning to the ambition that was investigated in Chapter 4. In MUiR, an entire set of multitask models is decomposed into an assembly of hypermodules. Each hypermodule can be used at any location in this decomposition, so sharing can occur across highly diverse architectures. As in the preceding systems, the assembly of hypermodules is optimized by interleaving evolution and gradient descent. To accelerate optimization, coevolution is performed at the module level, using learned soft mixtures as a surrogate fitness function. This speed-up was confirmed theoretically, and the expected behavior of the complete system demonstrated on a classical synthetic multitask learning dataset. The ability of the system to improve performance by sharing across vastly different deep architectures and tasks was then shown in multitask experiments that include vision, natural language, and genomics tasks, whose underlying models together include fully-connected, 1D convolutional, 2D convolutional, and LSTM layers. Through the optimization process, the number of parameters in the joint model was decreased, and intriguing sharing dynamics emerged, such as a *supermodule* that proliferates to parameterize the majority of pseudo-task locations. Overall, MUiR encodes a rich set of dynamics for optimizing modular sharing across arbitrary architecture-task pairs, and thus provides a promising starting point for practical methods that share across any and all available datasets.

10.2 Big Picture

Modern machine learning models are so complex that humans can no longer manually discern the functional regularities that exist across models for different tasks. Instead, deep multitask learning can be used to improve generalization by discovering such subsymbolic regularities automatically. This dissertation formalized a modular approach to deep multitask learning, which provides increased flexibility, scalability, and interpretability over using monolithic deep multitask models, and is more aligned with how functionality is organized in human intelligence and nature. Within this framework, a progression of modular deep multitask learning systems were developed, whose capabilities were demonstrated across a range of problem areas, including video game playing, vision, natural language, and genomics; and across a range of multitask scenarios, from intuitively related tasks to seemingly disparate tasks, and even when only a single task was available. With the suite of techniques developed for these systems, it is now practical to discover and share multi-purpose modules in arbitrary multitask scenarios. These advances point towards a future class of ecological machine learning systems that recycle modularized subsymbolic knowledge over indefinite universes and timescales.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Alipanahi, B., Delong, A., Weirauch, M. T., and Frey, B. J. Predicting the sequence specificities of dna-and rna-binding proteins by deep learning. *Nature biotechnology*, 33(8):831, 2015.

Ammar, H. B., Eaton, E., Luna, J. M., and Ruvolo, P. Autonomous cross-domain knowledge transfer in lifelong policy gradient reinforcement learning. In *Proceedings of IJCAI*, 2015a.

Ammar, H. B., Eaton, E., Ruvolo, P., and Taylor, M. E. Unsupervised cross-domain transfer in policy gradient reinforcement learning via manifold alignment. In *Proceedings of AAAI*, 2015b.

Anderson, M. L. Neural reuse: A fundamental organizational principle of the brain. *Behavioral and Brain Sciences*, 33:245–266, 2010.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, 5(1):54–65, 1994.

Argyriou, A., Evgeniou, T., and Pontil, M. Convex multi-task feature learning. *Machine Learning*, 73(3):243–272, Dec 2008.

Ba, J. and Frey, B. Adaptive dropout for training deep neural networks. In *NIPS*, pp. 3084–3092. 2013.

Bachman, P., Alsharif, O., and Precup, D. Learning with pseudo-ensembles. In *NIPS*, pp. 3365–3373. 2014.

Bäck, T., Fogel, D. B., and Michalewicz, Z. *Handbook of evolutionary computation*. CRC Press, 1997.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artifical Intelligence Research*, 47:253–279, 2013.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24*, pp. 2546–2554. 2011.

Bilen, H. and Vedaldi, A. Integrated perception with recurrent multi-task neural networks. In *NIPS*, pp. 235–243. 2016.

Bilen, H. and Vedaldi, A. Universal representations: The missing link between faces, text, planktons, and cat breeds. *CoRR*, abs/1701.07275, 2017.

Bonilla, E. V., Chai, K. M., and Williams, C. Multi-task gaussian process prediction. In *Advances in neural information processing systems*, pp. 153–160, 2008.

Bonner, J. T. *The evolution of complexity by means of natural selection*. Princeton University Press, 1988.

Braylan, A. and Miikkulainen, R. Object-model transfer in the general video game playing domain. In *Proceedings of AIIDE*, 2016.

Braylan, A., Hollenbeck, M., Meyerson, E., and Miikkulainen, R. Frame skip is a powerful parameter for learning to play atari. In *Workshops at AAAI*, 2015.

Braylan, A., Hollenbeck, M., Meyerson, E., and Miikkulainen, R. Reuse of neural modules for general video game playing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Brunskill, E. and Li, L. Pac-inspired option discovery in lifelong reinforcement learning. In *Proceedings of ICML*, pp. 316–324, 2014.

Bryant, B. D. and Miikkulainen, R. A neuroevolutionary approach to adaptive multi-agent teams. In Abbass, H. A., Scholz, J., and Reid, D. J. (eds.), *Foundations of Trusted Autonomy*, pp. 87–114. Springer, New York, 2018.

Brys, T., Harutyunyan, A., Taylor, M. E., and Nowé, A. Policy transfer using reward shaping. In *Proceedings of AAMAS*, pp. 181–188, 2015.

Caruana, R. Multitask learning. In *Learning to learn*, pp. 95–133. Springer US, 1998.

Chan, W., Jaitly, N., Le, Q. V., and Vinyals, O. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *Proceedings of ICASSP*, pp. 4960–4964, 2016.

Chen, T., Goodfellow, I., and Shlens, J. Net2net: Accelerating learning via knowledge transfer. In *Proc. of ICLR*, 2016.

Chen, W., Wilson, J., Tyree, S., Weinberger, K., and Chen, Y. Compressing neural networks with the hashing trick. In *Proceedings of ICML*, pp. 2285–2294, 2015.

Cheng, Y., Yu, F. X., Feris, R. S., Kumar, S., Choudhary, A., and Chang, S. F. An exploration of parameter redundancy in deep networks with circulant projections. In *Proceedings of ICCV*, pp. 2857–2865, 2015.

Chetlur, S. et al. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

Chollet, F. et al. Keras, 2015.

Christodoulou, C. and Georgiopoulos, M. *Applications of neural networks in electromagnetics*. Artech House, 2000.

Collobert, R. and Weston, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proc. of ICML*, pp. 160–167, 2008.

D'Ambrosio, D. B., Lehman, J., Risi, S., and Stanley, K. O. Evolving policy geometry for scalable multiagent learning. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, pp. 731–738, 2010.

Daniels, P. T. and Bright, W. *The world's writing systems*. Oxford University Press, 1996.

De Jong, K. A. *Evolutionary computation: a unified approach*. MIT Press, 2006.

Deb, K. Multi-objective evolutionary algorithms. In *Springer Handbook of Computational Intelligence*, pp. 995–1015. Springer, Berlin, Heidelberg, 2015.

Denil, M., Shakibi, B., Dinh, L., and De Freitas, N. Predicting parameters in deep learning. In *Advances in neural information processing systems*, pp. 2148–2156, 2013.

Devin, C., Gupta, A., Darrell, T., Abbeel, P., and Levine, S. Learning modular neural network policies for multi-task and multi-robot transfer. *CoRR*, abs/1609.07088, 2016.

Devries, T. and Taylor, G. W. Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552, 2017.

Dietterich, T. G. Ensemble methods in machine learning. *International workshop on multiple classifier systems*, pp. 1–15, 2000.

Doerr, B. and Auger, A. *Theory of randomized search heuristics: Foundations and recent developments*. World Scientific, 2011.

Doerr, B., Jansen, T., and Klein, C. Comparing global and local mutations on bit strings. In *Proceedings of GECCO*, pp. 929–936, 2008.

Doersch, C. and Zisserman, A. Multi-task self-supervised visual learning. In *Proceedings of ICCV*, 2017.

Dong, D., Wu, H., He, W., Yu, D., and Wang, H. Multi-task learning for multiple language translation. In *Proc. of ACL*, pp. 1723–1732, 2015.

Droste, S., Jansen, T., and Wegener, I. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1-2):51–81, 2002.

Durrett, G. and Klein, D. A joint model for entity analysis: Coreference, typing and linking. *Transactions of the ACL*, 2:477–490, 2014.

Durrett, G., Neumann, F., and O'Reilly, U.-M. Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics. In *Workshop Proceedings on Foundations of Genetic Algorithms*, pp. 69–80, 2011.

Eiben, A. E. and Smith, J. E. *Introduction to evolutionary computing.* 2003.

Eiben, A. E., Hinterding, R., and Michalewicz, Z. Parameter control in evolutionary algorithms. *IEEE Transactions on evolutionary computation*, 3 (2), 1999.

Eisenberg, B. On the expectation of the maximum of iid geometric random variables. *Statistics & Probability Letters*, 78(2):135–143, 2008.

Evgeniou, T. and Pontil, M. Regularized multi–task learning. In *Proc. of KDD*, pp. 109–117, 2004.

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. Pathnet: Evolution channels gradient descent in super neural networks. *CoRR*, abs/1701.08734, 2017.

Floreano, D., Durr, P., and Mattiussi, C. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, pp. 47–62, 2008.

Fogel, D. B., Fogel, L. J., and Porto, V. W. Evolving neural networks. *Biological Cybernetics*, 63(6):487–493, 1990.

Fukuda, T. and Shibata, T. Theory and applications of neural networks for industrial control systems. *IEEE Transactions on industrial electronics*, 39 (6):472–489, 1992.

Gaier, A., Asteroth, A., and Mouret, J.-B. Data-efficient neuroevolution with kernel-based surrogate models. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO)*, 2018.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of AISTATS*, pp. 249–256, 2010.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of AISTATS*, pp. 315–323, 2011.

Gomez, F. and Miikkulainen, R. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, 1997.

Gomez, F. and Miikkulainen, R. Solving non-markovian control tasks with neuroevolution. In *Proceedings of IJCAI*, pp. 1356–1361, 1999.

Gomez, F. and Miikkulainen, R. Active guidance for a finless rocket using neuroevolution. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO)*, pp. 2084–2095, 2003.

Gomez, F. J. *Robust non-linear control through neuroevolution*. Technical report, UT Austin, 2003.

Gomez, F. J. and Schmidhuber, J. Co-evolving recurrent neurons learn deep memory pomdps. In *Proceedings of GECCO*, pp. 491–498, 2005.

Graves, A., Mohamed, A., and Hinton, G. E. Speech recognition with deep recurrent neural networks. In *Proceedings of ICASSP*, pp. 6645–6649, 2013.

Greff, K., Srivastava, R. K., and Schmidhuber, J. Highway and residual networks learn unrolled iterative estimation. abs/1612.07771, 2016.

Günther, M., Rozsa, A., and Boult, T. E. AFFACT - alignment free facial attribute classification technique. *CoRR*, abs/1611.06158v2, 2017.

Gurney, W. and Nisbet, R. M. *Ecological Dynamics*. Oxford University Press, 1998.

Ha, D., Dai, A. M., and Le, Q. V. Hypernetworks. In *Proceedings of ICLR*, 2017.

Hand, E. M. and Chellappa, R. Attributes for improved attributes: A multi-task network utilizing implicit and explicit relationships for facial attribute classification. In *Proc. of AAAI*, pp. 4068–4074, 2017.

Hashimoto, K., Xiong, C., Tsuruoka, Y., and Socher, R. A joint many-task model: Growing a neural network for multiple NLP tasks. In *Proc. of EMNLP*, pp. 1923–1933, 2017.

Hausknecht, M. and Stone, P. The impact of determinism on learning atari 2600 games. In *Workshops at AAAI*, 2015.

Hausknecht, M., Lehman, J., Miikkulainen, R., and Stone, P. A neuroevolution approach to general atari game playing. *IEEE Trans. on Comp. Intelligence in AI in Games*, 6(4):355–366, 2013.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proc. of CVPR*, pp. 770–778, 2016.

He, K., Wang, Z., Fu, Y., Feng, R., Jiang, Y.-G., and Xue, X. Adaptively weighted multi-task deep network for person attribute classification. 2017.

Hinton, G., Vinyals, O., and Dean, J. Distilling the Knowledge in a Neural Network. *ArXiv e-prints*, 2015.

Hinton, G. et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. In *IEEE Signal processing magazine*, pp. 82–97, 2012.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. ISSN 0899-7667.

Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Howard, A. G., Zhu, M., Chen, B., et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

Huang, J. T., Li, J., Yu, D., Deng, L., and Gong, Y. Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers. In *Proc. of ICASSP*, pp. 7304–7308, 2013.

Huang, Z., Li, J., Siniscalchi, S. M., Chen, I.-F., Wu, J., and Lee, C.-H. Rapid adaptation for deep neural networks through multi-task learning. In *Proc. of Interspeech*, 2015.

Huizinga, J. and Clune, J. Evolving multimodal robot behavior via many stepping stones with the combinatorial multi-objective evolutionary algorithm. *CoRR*, abs/1807.03392, 2018.

Huizinga, J., Stanley, K. O., and Clune, J. The emergence of canalization and evolvability in an open-ended, interactive evolutionary system. In *Artificial Life*, 2018.

Hussain, M. A. Review of the application of neural networks in chemical process control—simulation and online implementation. *Artificial intelligence in engineering*, 13(1):55–68, 1999.

Ioffe, S. and Szegedy, C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of ICML*, pp. 448–456, 2015.

Iqbal, M., Browne, W. N., and Zhang, M. Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems. *IEEE Transactions on Evolutionary Computation*, 18(4):465–480, 2014.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., and Kavukcuoglu, K. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017a.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. Reinforcement learning with unsupervised auxiliary tasks. In *Proc. of ICLR*, 2017b.

Jaśkowski, W., Krawiec, K., and Wieloch, B. Multitask visual learning using genetic programming. *Evolutionary Computation*, 16(4):439–459, 2008.

Jayaraman, D., Gao, R., and Grauman, K. Shapecodes: Self-supervised feature learning by lifting view to viewgrids. In *Proceedings of ECCV*, 2018.

Jia, Y. et al. Caffe: Convolutional architecture for fast feature embedding. In *ACM International Conference on Multimedia*, pp. 675–678, 2014.

Jou, B. and Chang, S.-F. Deep cross residual learning for multitask visual recognition. In *Proc. of MM*, pp. 998–1007, 2016.

Jung, C., Hawkins, J. A., Jones, S. K., et al. Massively parallel biophysical analysis of crispr-cas complexes on next generation sequencing chips. *Cell*, 170(1):35–47, 2017.

Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., and Uszkoreit, J. One model to learn them all. *CoRR*, abs/1706.05137, 2017.

Kang, Z., Grauman, K., and Sha, F. Learning with whom to share in multi-task feature learning. In *Proc. of ICML*, pp. 521–528, 2011.

Khare, V. R., Yao, X., Sendhoff, B., Jin, Y., and Wersing, H. Co-evolutionary modular neural networks for automatic problem decomposition. In *Proceedings of CEC*, pp. 2691–2698, 2005.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., et al. Overcoming catastrophic forgetting in neural networks. *PNAS*, 114(13):3521–3526, 2017.

Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. Self-normalizing neural networks. In *Proceedings of NIPS*, pp. 971–980, 2017.

Kolda, T. G. and Bader, B. W. Tensor decompositions and applications. *SIAM Review*, 51:455–500, 2009.

Konidaris, G., Scheidwasser, I., and Barto, A. G. Transfer in reinforcement learning via shared features. *JMLR*, pp. 1333–1371, 2012.

Koutník, J., Schmidhuber, J., and Gomez, F. J. Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In *Proceedings of GECCO*, pp. 541–548, 2014.

Krizhevsky, A. *Learning Multiple Layers of Features from Tiny Images*. 2009.

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

Kumar, A. and Daumé, III, H. Learning task grouping and overlap in multi-task learning. In *Proc. of ICML*, pp. 1723–1730, 2012.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Lecun, Y. and Bengio, Y. Convolutional networks for images, speech and time series. *The handbook of brain theory and neural networks*, 3361(10), 1995.

Lecun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature*, 521(7553): 436–444, 2015.

Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z., and Tu, Z. Deeply-Supervised Nets. In *Proc. of AISTATS*, pp. 562–570, 2015.

Lee, H., Ekanadham, C., and Ng, A. Y. Sparse deep belief net model for visual area v2. In *NIPS*, pp. 873–880. 2008.

Lehman, J. and Stanley, K. O. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011a.

Lehman, J. and Stanley, K. O. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 211–218, 2011b.

Lehman, J., Clune, J., Misevic, D., Adami, C., Beaulieu, J., et al. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *CoRR*, abs/1803.03453, 2018.

Li, C., Farkhoor, H., Liu, R., and Yosinski, J. Measuring the intrinsic dimension of objective landscapes. In *Proceddings of ICLR*, 2018.

Li, P., Wang, Z., Lam, W., Ren, Z., and Bing, L. Salience estimation via variational auto-encoders for multi-document summarization. In *Proceedings of AAAI*, pp. 3497–3503, 2017.

Li, X. and Miikkulainen, R. Evolving multimodal behavior through subtask and switch neural networks. In *Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems*, 2014.

Li, Z., Gong, B., and Yang, T. Improved dropout for shallow and deep learning. In *NIPS*, pp. 2523–2531. 2016.

Liang, J., Meyerson, E., and Miikkulainen, R. Evolutionary architecture search for deep multitask networks. In *Proceedings of GECCO*, 2018.

Liang, M. and Hu, X. Recurrent convolutional neural network for object recognition. In *Proc. of CVPR*, 2015.

Liao, Q. and Poggio, T. A. Bridging the gaps between residual learning, recurrent neural networks and visual cortex. *CoRR*, abs/1604.03640, 2016.

Lichman, M. UCI machine learning repository, 2013.

Lillicrap, T. P. et al. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. Hierarchical representations for efficient architecture search. In *Proceedings of ICLR*, 2018.

Liu, X., Gao, J., He, X., Deng, L., Duh, K., and Wang, Y. Y. Representation learning using multi-task deep neural networks for semantic classification and information retrieval. In *Proc. of NAACL*, pp. 912–921, 2015a.

Liu, Z., Luo, P., Wang, X., and Tang, X. Deep learning face attributes in the wild. In *Proc. of ICCV*, 2015b.

Long, M., Cao, Z., Wang, J., and Yu, P. S. Learning multiple tasks with multilinear relationship networks. In *NIPS*, pp. 1593–1602. 2017.

Loshchilov, I. and Hutter, F. CMA-ES for hyperparameter optimization of deep neural networks. *CoRR*, abs/1604.07269, 2016.

Lu, Y., Kumar, A., Zhai, S., Cheng, Y., Javidi, T., and Feris, R. S. Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. *Proc. of CVPR*, 2017.

Lukoševičius, M. and Jaeger, H. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

Luong, M. T., Le, Q. V., Sutskever, I., Vinyals, O., and Kaiser, L. Multi-task sequence to sequence learning. In *Proc. ICLR*, 2016.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning word vectors for sentiment analysis. In *Proc. of ACL: HLT*, pp. 142–150, 2011.

Maclaurin, D., Duvenaud, D., and Adams, R. Gradient-based hyperparameter optimization through reversible learning. In *Proc. of ICML*, pp. 2113–2122, 2015.

Mahmud, M. M. and Ray, S. Transfer learning using Kolmogorov complexity: Basic theory and empirical evaluations. In *NIPS*, pp. 985–992. 2008.

Mahmud, M. M. H. On universal transfer learning. *Theoretical Computer Science*, 410(19):1826 – 1846, 2009.

May, R. and McLean, A. R. *Theoretical ecology: principles and applications.* Oxford University Press on Demand, 2007.

McCann, B., Bradbury, J., Xiong, C., and Socher, R. Learned in translation: Contextualized word vectors. In *NIPS*, pp. 6297–6308. 2017.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016.

Meyerson, E. and Miikkulainen, R. Discovering evolutionary stepping stones through behavior domination. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 139–146, 2017.

Meyerson, E. and Miikkulainen, R. Beyond shared hierarchies: Deep multitask learning through soft layer ordering. In *Proc. of ICLR*, 2018a.

Meyerson, E. and Miikkulainen, R. Pseudo-task augmentation: From deep multitask learning to intratask sharing—and back. In *Proc. of ICML*, 2018b.

Meyerson, E., Lehman, J., and Miikkulainen, R. Learning behavior characterizations for novelty search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 149–156, 2016.

Miikkulainen, R. Evolving neural networks. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. ACM, 2016.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., and Hodjat, B. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. Network motifs: Simple building blocks of complex networks. *Science*, 298 (5594):824–827, 2002.

Misra, I., Shrivastava, A., Gupta, A., and Hebert, M. Cross-stitch networks for multi-task learning. In *Proc. of CVPR*, 2016.

Mnih, V. et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015a.

Mnih, V. et al. Asynchronous methods for deep reinforcement learning. In *Proceedings of ICML*, pp. 1928–1937, 2015b.

Moriarty, D. E. and Miikkulainen, R. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1–3):11–32, 1996.

Moriarty, D. E. and Miikkulainen, R. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5(4):373–399, 1997.

Moriarty, D. E. and Miikkulainen, R. Hierarchical evolution of neural networks. In *Proc. of IEEE World Congress on Computational Intelligence*, pp. 428–433, 1998.

Mouret, J.-B. and Doncieux, S. Encouraging behavioral diversity in evolutionary robotics: An empirical study. *Evolutionary computation*, 20(1):91–133, 2012.

Nguyen, A. M., Yosinski, J., and Clune, J. Innovation engines: Automated creativity and improved stochastic optimization via deep learning. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 959–966, 2015.

Nitschke, G. Emergence of cooperation: State of the art. *Artificial Life*, 11(3): 367–396, 2005.

Nitschke, G. S., Schut, M. C., and Eiben, A. E. Evolving behavioral specialization in robot teams to solve a collective construction task. *Swarm and Evolutionary Computation*, 2:25–38, 2012.

Pan, S. J. and Yang, Q. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

Parker, M. and Bryant, B. D. Lamarckian neuroevolution for visual control in the quake ii environment. In *IEEE Congress on Evolutionary Computation*, pp. 2630–2637, 2009.

Paske, A. et al. Automatic differentiation in pytorch. 2017.

Perez, D., Samothrakis, S., Togelius, J., Schaul, T., et al. The 2014 general video game playing competition. *IEEE Transactiona on Computational Intelligence and AI in Games*, 99, 2015.

Pinheiro, P. and Collobert, R. Recurrent convolutional neural networks for scene labeling. In *ICML*, pp. 82–90, 2014.

Pugh, J. K. and Stanley, K. O. Evolving multimodal controllers with hyperneat. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pp. 735–742, 2013.

Pugh, J. K., Soros, L. B., and Stanley, K. O. Quality diversity: a new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 2016.

Ranjan, R., Patel, V. M., and Chellappa, R. Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition. *CoRR*, abs/1603.01249, 2016.

Rawal, A. and Miikkulainen, R. From nodes to networks: Evolving recurrent neural networks. *CoRR*, abs/1803.04439, 2018.

Rawal, A., Rajagopalan, P., and Miikkulainen, R. Constructing competitive and cooperative agent behavior using coevolution. In *Proceedings of CIG*, pp. 107–114, 2010.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. Large-scale evolution of image classifiers. In *Proc. of ICML*, pp. 2902–2911, 2017.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018.

Rebuffi, S.-A., Bilen, H., and Vedaldi, A. Learning multiple visual domains with residual adapters. In *NIPS*, pp. 506–516. 2017.

Reisinger, J., Stanley, K. O., and Miikkulainen, R. Evolving reusable neural modules. In *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 69–81, 2004.

Rezende, D., Mohamed, S., Danihelka, I., Gregor, K., and Wierstra, D. One-shot generalization in deep generative models. In *Proc. of ICML*, pp. 1521–1529, 2016.

Richards, N., Moriarty, D. E., and Miikkulainen, R. Evolving neural networks to play go. *Applied Intelligence*, 8(1):85–96, 1998.

Rudd, E. M., Günther, M., and Boult, T. E. MOON: A mixed objective optimization network for the recognition of facial attributes. In *Proc. of ECCV*, pp. 19–35, 2016.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., et al. Progressive neural networks. *CoRR*, abs/1606.04671, 2016.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.034864, 2017.

Saxe, A. M., McClelland, J. L., and Ganguli, S. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *Proc. of ICLR*, 2014.

Schaul, T. A video game description language for model-based or interactive learning. In *Proceedings of CIG*, pp. 1–8, 2013.

Schmidhuber, J., Wierstra, D., Gagliolo, M., and Gomez, F. J. Training recurrent networks by evolino. *Neural Computation*, 19(3):757–779, 2007.

Schrum, J. and Miikkulainen, R. Evolving multimodal networks for multitask games. *IEEE Transactions on Computational Intelligence in AI and Games*, 4(2):94–111, 2012.

Schrum, J. and Miikkulainen, R. Discovering multimodal behavior in ms. pac-man through evolution of modular neural networks. *IEEE transactions on computational intelligence and AI in games*, 8(1):67–81, 2016.

Secretan, J., Beato, N., D'Ambrosio, D. B., Rodriguez, A., Campbell, A., and Stanley, K. O. Picbreeder: evolving pictures collaboratively online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1759–1768, 2008.

Seltzer, M. L. and Droppo, J. Multi-task learning in deep neural networks for improved phoneme recognition. In *Proc. of ICASSP*, pp. 6965–6969, 2013.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR*, 2017.

Shultz, T. R. and Rivest, F. Knowledge-based cascade-correlation: Using knowledge to speed learning. *Connection Science*, 13(1):43–72, 2001.

Silver, D. L., Yang, Q., and Li, L. Lifelong machine learning systems: Beyond learning algorithms. *AAAI Spring Symposium: Lifelong Machine Learning*, 13, 2013.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *NIPS*, pp. 2951–2959. 2012.

Socher, R., Lin, C. C.-Y., Ng, A. Y., and Manning, C. D. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, pp. 129–136, 2011.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *JMLR*, 15(1):1929–1958, 2014.

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. A hypercube-based encoding for evolving large-scale neural networks. volume 15, pp. 185–212, 2009.

Such, F. P., Madhaven, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *Corr*, abs/1712.06567, 2017.

Suganuma, M., Shirakawa, S., and Nagao, T. A genetic programming approach to designing convolutional neural network architectures. In *Proc. of GECCO*, pp. 497–504. ACM, 2017.

Sutton, R. S. and Barto, A. G. *Introduction to reinforcement learning.* MIT Press, 1998.

Suutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Swarup, S. and Ray, S. R. Cross-domain knowledge transfer using structured representations. In *Proceedings of AAAI*, pp. 506–511, 2006.

Szegedy, C., Liu, W., Jia, Y., et al. Going deeper with convolutions. Proc. of CVPR, 2015.

Szegedy, C., Ioffe, S., and Vanhoucke, V. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

Talvitie, E. and Singh, S. An experts algorithm for transfer learning. In *Proceedings of IJCAI*, pp. 1065–1070, 2007.

Taylor, L. and Nitschke, G. Improving deep learning using generic data augmentation. *CoRR*, abs/1708.06020, 2017.

Taylor, M. E. and Stone, P. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, pp. 1633–1685, 2009.

Taylor, M. E., Whiteson, S., and Stone, P. Transfer via inter-task mappings in policy search reinforcement learning. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, 2007.

Taylor, M. E., Kuhlmann, G., and Stone, P. Autonomous transfer for reinforcement learning. In *Proceedings of AAMAS*, pp. 283–290, 2008.

Teh, Y., Bapst, V., Czarnecki, W. M., Quan, J., Kirkpatrick, J., Hadsell, R., Heess, N., and Pascanu, R. Distral: Robust multitask reinforcement learning. In *NIPS*, pp. 4499–4509. 2017.

Tessler, C., Givony, S., Zahavy, T., Mankowitz, D. J., and Mannor, S. A deep hierarchical approach to lifelong learning in minecraft. In *AAAI*, 2017.

Thrun, S. and Pratt, L. *Learning to Learn*. 2012.

Tieleman, T. and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 2012.

Toshniwal, S., Tang, H., Lu, L., and Livescu, K. Multitask Learning with Low-Level Auxiliary Tasks for Encoder-Decoder Based Speech Recognition. *CoRR*, abs/1704.01631, 2017.

van der Maaten, L. and Hinton, G. Visualing data using t-sne. *JMLR*, 9: 2579–2605, Nov 2008.

Vaswani, A. et al. Attention is all you need. In *Proceedings of NIPS*, pp. 5998–6008, 2017.

Veit, A., Wilber, M. J., and Belongie, S. Residual networks behave like ensembles of relatively shallow networks. In *NIPS*, pp. 550–558. 2016.

Velez, R. and Clune, J. Identifying core functional networks and functional modules within artificial neural networks via subsets regression. In *Proceedings of GECCO*, 2016.

Verbancsics, P. and Stanley, K. O. Evolving static representations for task transfer. *Journal of Machine Learning Research*, 11:1737–1769, 2010.

Vittorio, M. Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on neural networks*, 5(1):39–53, 1994.

Wei, T., Wang, C., Rui, Y., and Chen, C. W. Network morphism. In *Proc. of ICML*, pp. 564–572, 2016.

Whiteson, S. and Stone, P. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:877–917, 2006.

Whiteson, S., Kohl, N., Miikkulainen, R., and Stone, P. Evolving soccer keepaway players through task decomposition. *Machine Learning*, 59(1–2): 5–30, 2005.

Witt, C. Tight bounds on the optimization time of a randomized search heuristic on linear functions. *Combinatorics, Probability and Computing*, 22 (2):294–318, 2013.

Wu, Z., Valentini-Botinhao, C., Watts, O., and King, S. Deep neural networks employing multi-task learning and stacked bottleneck features for speech synthesis. In *Proc. of ICASSP*, pp. 4460–4464, 2015.

Yang, Y. and Hospedales, T. A unified perspective on multi-domain and multi-task learning. In *Proceedings of ICLR*, 2015.

Yang, Y. and Hospedales, T. Deep multi-task representation learning: A tensor factorisation approach. In *Proc. of ICLR*, 2017.

Yang, Z., Moczulski, M., Denil, M., De Freitas, N., et al. Deep fried convnets. In *Proc. of ICCV*, pp. 1476–1483, 2015.

Yao, X. Evolving artificial neural networks. In *Proc. of the IEEE*, volume 87, pp. 1423–1447, 1999.

Zagoruyko, S. and Komodakis, N. Wide residual networks. *CoRR*, abs/1605.07146, 2016.

Zamir, A. R., Wu, T., Sun, L., Shen, W., Malik, J., and Saverese, S. Feedback networks. *CoRR*, abs/1612.09508, 2016.

Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

Zeng, H., Edwards, M. D., Liu, G., and Gifford, D. K. Convolutional neural network architectures for predicting dna-protein binding. *Bioinformatics*, 32 (12):i121–i127, 2016.

Zhang, Y. and Weiss, D. Stack-propagation: Improved representation learning for syntax. pp. 1557–1566, 2016.

Zhang, Z., Ping, L., Chen, L. C., and Xiaoou, T. Facial landmark detection by deep multi-task learning. In *Proc. of ECCV*, pp. 94–108, 2014.

Zhang, Z., Zhao, J., and LeCun, Y. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pp. 649–657, 2015.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. In *Proc. of ICLR*, 2017.