University of Massachusetts Amherst

ScholarWorks@UMass Amherst

Doctoral Dissertations

Dissertations and Theses

March 2020

Dynamic Composition of Functions for Modular Learning

Clemens GB Rosenbaum

University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation

Rosenbaum, Clemens GB, "Dynamic Composition of Functions for Modular Learning" (2020). *Doctoral Dissertations*. 1865.

https://doi.org/10.7275/15649011 https://scholarworks.umass.edu/dissertations_2/1865

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

DYNAMIC COMPOSITION OF FUNCTIONS FOR MODULAR LEARNING

A Dissertation Presented

by

CLEMENS G. B. ROSENBAUM

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2020

College of Information and Computer Sciences

© Copyright by Clemens G. B. Rosenbaum 2020

All Rights Reserved

DYNAMIC COMPOSITION OF FUNCTIONS FOR MODULAR LEARNING

A Dissertation Presented

by

CLEMENS G. B. ROSENBAUM

Approved as to style and content by:

____

Sridhar Mahadevan, Chair

____

Eric Learned-Miller, Member

____

Philip S. Thomas, Member

____

Darby Dyar, Member

____

James Allan, Chair
College of Information and Computer Sciences

ACKNOWLEDGMENTS

First and foremost, thanks to my brilliant wife, Christiane, whose tremendous support included her following me across the Atlantic Ocean. I also want to thank my family, who was crucial in motivating me. Thank you to my old and new friends without whom the past years would have been a lot less fulfilling and certainly more boring.

I am also deeply grateful to all the mentors and collaborators I had over the last couple of years. Among them are my advisor Sridhar Mahadevan, from whom I learned much about being a researcher. Many thanks also to my committee members, Erik Learned-Miller, Philip Thomas and Darby Dyar with especially the latter offering incredibly helpful advice and support to ensure my progress.

During my internships at IBM, I also greatly benefited from the insights and expertise provided by my colleagues, whose support continued long after. These include Gerald Tesauro, Matt Riemer, and especially Ignacio Cases and Tim Klinger. I want to finally thank all of my co-student collaborators, most of them though the University of Massachusetts, for all of their helpful comments and constructive criticism.

ABSTRACT

DYNAMIC COMPOSITION OF FUNCTIONS FOR MODULAR LEARNING

FEBRUARY 2020

CLEMENS G. B. ROSENBAUM

B.Sc., TECHNISCHE UNIVERSITÄT DARMSTADT

MAGISTER ARTIUM, J.W. GOETHE UNIVERSITÄT FRANKFURT

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Sridhar Mahadevan

Compositionality is useful to reduce the complexity of machine learning models and increase their generalization capabilities, because new problems can be linked to the composition of existing solutions. Recent work has shown that compositional approaches can offer substantial benefits over a wide variety of tasks, from multi-task learning over visual question-answering to natural language inference, among others. A key variant is functional compositionality, where a meta-learner composes different (trainable) functions into complex machine learning models. In this thesis, I generalize existing approaches to functional compositionality under the umbrella of the *routing* paradigm, where trainable arbitrary functions are 'stacked' to form complex machine learning models.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . **iv**

ABSTRACT . . . . **v**

LIST OF TABLES . . . . **x**

LIST OF FIGURES . . . . **xi**

LIST OF ALGORITHMS . . . . **xv**

CHAPTER

1. INTRODUCTION **1**

Outline & Contributions 5
Definitions and Notation 7

2. RELATED WORK **9**

2.1 Modular and Compositional Learning in Other Disciplines 10
2.2 Mixtures of Experts and ‘Soft’ Routing 12

2.2.1 Conditional Activations of Neurons 13

2.3 Other forms of Conditional Computation 14
2.4 Numerical Analysis 14
2.5 Meta-learning and Architecture Search 15
2.6 Multi-task Learning 18
2.7 Existing Work on Routing 19

3. ROUTING **21**

3.1 Reinforcement Learning 22

3.1.1 Routing MDPs 23

3.1.2 Reward Design 24

3.1.2.1 Final Rewards 24
3.1.2.2 Regularization Rewards 24

3.1.3 Algorithms 25

3.1.3.1 Value-based RL Algorithms 25
3.1.3.2 Policy Gradient based RL algorithms 28
3.1.3.3 RL algorithms for routing 30

3.2 Stochastic Reparameterization 31
3.3 Training 34
3.4 Architectures 35

3.4.1 Model Architectures 35
3.4.2 Router Architectures 36

3.5 The Routing Search Space 38
3.6 A Different Perspective 39

4. CHALLENGES TO ROUTING AND OTHER FORMS OF COMPOSITIONAL COMPUTATION 41

4.1 Underlying Causes 41

4.1.1 The Transfer-Interference Trade-Off 42
4.1.2 The Exploration-Exploitation Dilemma 42

4.2 Training Stability 43
4.3 Module Collapse 45
4.4 Overfitting 47
4.5 Module Diversity 49
4.6 A uniform formal framework 50
4.7 The Challenges and the Routing Search Space 52
4.8 Modular Learning and the Bias-Variance Trade-Off 53

5. ROUTING WITH META-INFORMATION 55

5.1 Multi-Task Learning and Multi-Agent Routing 56
5.2 Experiments on Image Datasets 58

5.2.1 The Datasets 58
5.2.2 The Architecture 59
5.2.3 Results 60

5.3 Experiments on Language Datasets 65

5.3.1 The Architecture 68
5.3.1.1 Routing Classifiers 69
5.3.1.2 Routing RNN Encoders 69
5.3.1.3 Routing CBOW Encoders 70
5.3.1.4 Routing Transformers 71
5.3.2 Results 72
5.3.2.1 MULTINLI Results 73
5.3.2.2 SCI Results 74
5.3.2.3 Navigating the Transfer–Interference Trade-off 75
5.3.3 Prediction without Meta-Information at Test-Time 76
6. DISPATCHED ROUTING 78
6.1 Conceptual Advantages of Dispatching 78
6.2 Dispatching Strategies 80
6.2.1 Offline Dispatching by Clustering 80
6.2.2 Online Dispatching without an additional Objective Function 81
6.2.3 Online Dispatching with an additional Objective Function 81
6.2.4 CBOW Autoencoding 83
6.2.5 CBOW Autoencoding with Attention 84
6.2.6 One-hot (unencoded) Autoencoding 84
6.2.7 Sequence to Sequence Autoencoding 84
6.3 Experiments 84
6.3.1 Quantitative Results 85
6.3.2 Offline Dispatching 85
6.3.3 Online Dispatching 87
7. AN EMPIRICAL ANALYSIS 91
7.1 Decision Making Algorithms 91
7.1.1 Additional Algorithms 91
7.1.2 Decision Making Strategies: The Algorithms 93
7.1.3 Routing without Meta-Information and without Dispatching 94
7.1.4 Routing with Meta-Information 97
7.1.5 Routing without Meta-Information but with Dispatching 99
7.1.6 RL vs Reparameterization and the Impact of Exploration 100
7.2 Decision Making Hyperparameters 101

7.2.1 Exploration 101
7.2.2 Learning Rates 102

7.3 Reward Design 103

7.3.1 Final Rewards 103
7.3.2 Regularization Rewards 104

7.4 Other Router Design Choices 106

7.4.1 Entropy Regularization 106
7.4.2 Exploratory Actions 106
7.4.3 Splitting training data 107
7.4.4 Optimization Algorithm / SGD 108
7.4.5 Gradient Flow from Router to Activations 109

7.5 Analyzing the Challenges 109

7.5.1 Stability 109
7.5.2 Module Collapse 110
7.5.3 Overfitting 111

8. CONCLUSIONS 113

8.1 Contributions 113
8.2 Future Work 115

BIBLIOGRAPHY 117

LIST OF TABLES

Table **Page**

5.1 Examples from SCI randomly chosen from the validation set. Each row contains a triplet formed by a premise (left column), a hypothesis (right column), and a label specifying one of the three possible relations (*entails*, *contradicts*, *permits*) holding between premise and hypothesis. The last row contains an example of a probabilistic implicative (see the main text). 67

5.2 Results for MULTINLI and SCI with different baselines and their routed versions. We report average accuracy with confidence intervals over five runs with different seeds. For Transformers, we found that finetuning was highly volatile. We therefore report test results from the best-of-5 train models. 'WP' stands for Word Projection, '+D' for Dispatching, 'I2H' for Input-to-Hidden routing, and 'H2H' for Hidden-to-Hidden routing. Italics mark scores whose confidence intervals overlap with the best scores. *All results for nested SCI were computed by fine-tuning the same network previously trained on joint. † Transformers were pretrained as a language model. 73

6.1 Test results on SCI, averaged over five runs with different seeds. Each entry consists of the test accuracy with confidence intervals and the entropy of the learned dispatching policy at test time. Each result is selected from the average best-of-five dev results from a hyperparameter sweep and may thereby have different hyperparameter settings. Bold font marks the average best score, and italics mark scores whose confidence intervals overlap with the best scores. 87

LIST OF FIGURES

Figure **Page**

1.1 A *routing network*. The router consists of a parameterized decision maker that iteratively selects modules (i.e., trainable functions). Each selected function is then applied to the latest activation, resulting in a new activation, which can then again be transformed. The training of a routing network happens *online*, i.e., the output of the model is used to train the transformations using backpropagation and Stochastic Gradient Descent (SGD), and is simultaneously used to provide feedback to the decision maker. 3

2.1 A dataset with two subsets, each approximated by a different function. 15

3.1 Routing (forward) Example including a termination action $\perp$. 21

3.2 Gradient statistics. The REINFORCE Baseline is computed as $\hat{r} = (1-\alpha)\hat{r} + \alpha r$ with $\alpha = 0.1$. The temperature parameter for all approaches is 0.5, which appears to be a common choice in the literature. 33

3.3 Routing (backward) Example 34

3.4 Routing architectures 36

4.1 Depiction of the effects of transfer and interference across examples i and k during learning. 42

4.3 An example of how a 1-dimensional linear routing problem can collapse 46

4.4 Illustration of how a routed model may overfit. The learned parameter values of the three linear transformations are $a = 3, b = 0.1, c = 0.8$. 47

4.5 Module Learning Dynamics 49

4.6 The flexibility dilemma. Flexibility is the ability of a routing model to adapt to localized inputs. Low flexibility means few large clusters, and low flexibility means many small clusters. The extrema are only one cluster, i.e., collapsing and underfitting, and one cluster for each sample, i.e., overfitting. 53

5.1 Comparison of Routing Architectures on CIFAR-MTL. 61

5.2 Results on domain CIFAR-MTL 62

5.3 Results on domain MIN-MTL (mini ImageNet) 62

5.4 Results on domain MNIST-MTL 62

5.5 Comparison of per-task training cost for cross-stitch and routing networks. We add a module per task and normalize the training time per epoch by dividing by the number of tasks to isolate the effect of adding modules on computation. 64

5.6 An actual routing map for MNIST-MTL. 65

5.7 The general NLI architecture. 68

5.8 Routing the classifier. 69

5.9 Routing the input to hidden layer of an RNN. 70

5.10 Routing the hidden to hidden layer of an RNN. 70

5.11 Routing a CBOW encoder. 71

5.12 Path (module selection) overlap for MULTINLI between genres with the CBOW GloVe WP model. The diagonal represents the number of modules applied for a genre. A maximum of three means that two genres would be routed through the exact same functions. 74

5.13 The path-overlap between different signatures on SCI, using the CBOW GloVe WP model, for $b = 4$, $d = 3$. N$+|-$ and N$-|+$ are nested signatures. 75

6.1 Dispatching strategies. In *offline* dispatching, the dispatching strategy π_d is decided by information provided externally (m), and the dispatcher is not trained as part of the routing network, i.e., it is not part of the feedback loop. In *online* dispatching, the dispatcher is trained, and is thus part of the feedback loop. When trained on the *same target*, the objective function of the dispatcher is the same as for the router. When trained on a *different target*, the dispatcher has another objective function (and is part of another feedback loop), and is trained on both that target and the target provided by the routing network. 80

6.2 Online dispatching with an additional objective function. The dispatcher chooses an action which determines both the sub-policy routing the sample and an external regularization action. The dispatcher is then trained on feedback from both. The external regularization problem may be any single-step routing network, with any target z. 82

6.3 A (routed) bottleneck autoencoder. After the input S is encoded, the encoding E_S is projected to some dimensionality d_p, and then passed through one of a set of "bottleneck" modules, $\{m_1, m_2, m_3\}$, selected by the autoencoder dispatching policy π_d. These are of the form $fc_{d_p,d_{bn}} \rightarrow fc_{d_{bn},d_p}$, i.e., they first project the input from dimensionality d_p to a bottleneck dimensionality d_{bn} (such that $d_{bn} \ll d_p$). The resulting projection is then projected back from d_p to the dimensionality of the embedding. The entire model, including π_d, is trained on the reconstruction error between $\hat{E}_S$ and E_S. If d_{bn} is sufficiently small, the π_d has to cluster samples by distributing them over multiple bottleneck modules. 83

6.4 Comparison of dispatching architectures 86

6.5 Comparison of the behavior of different dispatching strategies. For each subplot, the topmost graph shows the distribution of the ground truth signatures; the center graph shows the dispatching distribution per signature, and the lowest shows the overall dispatching distribution. 89

7.1 Comparison of different decision-making algorithms on two single agent architectures. 95

7.2 Decision-making ablation studies on CIFAR. Both experiments compare different decision-making algorithms on restricted architectures. 96

7.3 Multi-task results for different decision-making algorithms. 98

7.4 Comparison of decision-making algorithms over different dispatched architectures and hyperparameter settings. 99

7.5 An ε-greedy version of REINFORCE on CIFAR 100 MTL 100

7.6 Results for different levels of exploration. High starts at 0.5 and anneals every 5 epochs by a factor of 4, until it reaches 0.01. Med starts at 0.25 and anneals every 5 epochs by a factor of 4, until it reaches 0.005. Low starts at 0.1 and anneals every 5 epochs by a factor of 4, until it reaches 0.002. 101

7.7 Results for different routing learning rates, in ratio to the module learning rate of 0.001. 102

7.8 Results for different reward functions 104

7.9 Multi-task results for different intermediate reward values 105

7.10 Squashing training for exploratory trajectories on CIFAR100 MTL, with values for α_{exp}. 107

7.11 Stochastic router-transformation training on CIFAR100 MTL 108

7.12 The effect of the optimizer choice on CIFAR100 MTL 108

7.13 Collapse on CIFAR 100 with a dispatched architecture 110

7.14 Overfitting on SCI 111

LIST OF ALGORITHMS

3.1 Routing Forward. The **router** function computes an action from the state, and can thus be any kind of hard decision-making algorithm. $\perp$ is a termination action, and **module**$_k$ is the selected module or function from the set of available modules. This algorithm may be modified to accommodate constrains on the subset of modules applicable to the current activation. 22
5.1 Weighted Policy Learner 57
7.1 Weighted Policy Learner: Approximation Version 92

CHAPTER 1

INTRODUCTION

When we think about Artificial Intelligence, we think about advances in gameplaying – from Backgammon over chess to Go – traditionally thought to be a domain of human reasoning. We think about the incredible progress in searching through huge amounts of data, progress in language translation, and video recognition. We might even think about advances in medicine and other areas of science. All of these advances share a reliance on machine learning, on automatically discovering the rules guiding a particular problem and to generalize to new examples. If we look more closely, though, we can see that machine learning played a different role in advances 20 years ago than in more recent ones. Mostly limited by the availability of affordable computation, it was oftentimes cheaper in the 1990s to hire human experts to hard code rules guiding the machine learning algorithms – as was done for Deep Blue, the chess computer famously beating the acting world champion. Twenty years later, and with trillions of computations per second more, Alpha Zero became the strongest chess player in history, without having to rely on any human supervision whatsoever. A similar story happened with machine translation, when Google translate shifted to primarily rely on machine learning in 2016.Many similar stories could be told – from self-driving cars to fraud detection.

These stories are huge successes for AI and machine learning. What they do not tell, at least not immediately, are the challenges that machine learning faces, and where we, as machine learning researchers, still have a long way to go. These challenges include (but are not limited to): learning from few examples, transferring skills and knowledge to different problems, remembering old skills and knowledge after the arrival of new information

and knowledge, abstracting reasoning skills across domains, and jointly learning to solve multiple problems. We know, however, that these skills should be achievable because human beings are able to do them with ease. And while an intelligent program should not necessarily model human cognition, it oftentimes pays off to consider insight the cognitive sciences have reached.

One of these insights is the core concept of this thesis: compositional modularity. Playing a central role in research on human cognition since the 1970s, it assumes that human cognition divides labor by being decomposed into specialized subsystems, each one performing specific subtasks. The arguments for modularity, especially in learning, are so persuasive that they have also played a role in machine learning research since the early 1990s. On a high level, they can be subsumed under the umbrella of specialization. Intuitively, it makes sense to have specialized submodules, because this 'divide-and-conquer' idea should be able to break down a problem into simpler subparts. Additionally, the solutions to each of these parts may be usable over a variety of problems, allowing for faster adaptation to new problems, by simply recomposing already learned skills.

In the context of neural networks, modular and compositional learning is a special case of conditional computation, a machine learning strategy where different tasks of arbitrary granularity – different samples, contexts, or problems and datasets – will be solved by the same model, with a possibly unique set of active parameters for each task. Conditional computation shares several advantages with modular and compositional learning: (1) Such approaches allow a system to scale better, because they allow huge (in terms of parameter count) implicit models where only a small number of parameters is actually required to be active in any single computation. (2) They should allow a system to have more specialized 'parts', each encompassing a special skill. While many modern architectures, in particular deep architectures, should in theory be able to achieve such compartmentalization, they generally do not do so in practice. (3) Depending on the degree of freedom in the composition, a system may be able to create a great many different latent models, thereby

generalizing over a wider variety of tasks. (4) The same strategy could allow the system to maintain every learned skill, as long as the adequate composition is stored. (5) Because the system may be able to avoid learning interference, and group similar samples to similar parameters, it should be able to train faster and to better performance.

In this thesis, I will argue for a special case of modular and compositional learning: functional composition. For functional composition, entire parameterized and trainable functions are 'stacked' to form a complete machine learning model. In many ways, functional compositionality reflects our human intuitions the closest, because complete functions are a good approximation of the subskills we attribute to ourselves when solving a problem. In particular, I will advocate for an approach I call *routing*, where a centralized *router* (the composition strategy) iteratively selects *modules* (trainable functions) and applies them to the input, until some stopping criterion is met. The result of the final application can then either be forwarded to other, (non-routed) functions or interpreted as the composite model's prediction. Importantly, and distinguishing this approach from the slightly different research area of architecture search, this composition happens for each sample anew, forming possibly different compositions for different samples. Additionally, the model is updated immediately after receiving feedback for the model's prediction. If the applied modules are artificial neural networks, such a model is called a *routing network*, and is depicted in Figure 1.1.

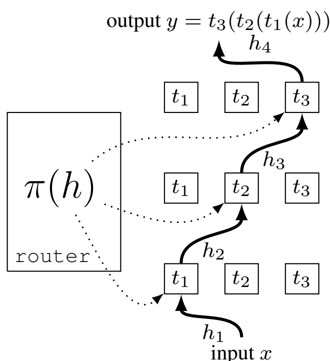


Figure 1.1: A *routing network*. The router consists of a parameterized decision maker that iteratively selects modules (i.e., trainable functions). Each selected function is then applied to the latest activation, resulting in a new activation, which can then again be transformed. The training of a routing network happens *online*, i.e., the output of the model is used to train the transformations using backpropagation and Stochastic Gradient Descent (SGD), and is simultaneously used to provide feedback to the decision maker.

These will form the focus of this investigation, because they have already proven to be very successful for several domains, from multi-task learning to language inference.

The most important aspect to routing networks (and arguably, to any kind of routed model) is the training of the modules and the training of the router. For routing networks, the modules can be trained using the highly performant and highly efficient backpropagation algorithm, which (arguably) is the main reason for the recent successes of neural architectures. Because the router, on the other hand, needs to learn to make decisions from purely evaluative, and not instructive, feedback, we need to consider different optimization strategies. While there are several other training algorithms possible for the router, I will focus on two that I believe to be the most principled. The first is reinforcement learning (RL), which is the most impactful and best investigated area of research for sequential decision making. Reinforcement learning only learns from a scalar reward provided to the agent (the router, in our case), provided as a feedback to each action (module selection, in our case). The second training algorithm for the router uses recently introduced reparameterization techniques for discrete stochastic distributions. These are, on a first glance, much more fitting solutions because they can also be trained with backpropagation, allowing for a seamless integration into routing networks.

While learning hard selections is always difficult, routing adds even more complexity to the training of both modules and router. The main problem is the mutual non-stationary nature of either training process: From the perspective of the router, the learning problem is non-stationary, because the parameters of the modules constantly change. From the perspective of the modules, the composition and thus their role in the composed model may change whenever the router changes the routing path. I will dedicate a separate chapter to this problem and some of its consequences, highlighting some of the interplay between composition and modules and analyzing the problem to derive better solutions.

Outline & Contributions

This thesis can roughly be separated into two separate parts, conceptual and empirical. In the second chapter, I will relate routing to existing approaches and paradigms in the literature. In the third chapter, I will re-introduce routing (networks) in detail. This will mainly focus on explaining the routing algorithm, as well as going over a multitude of different design decisions and their potential benefits. The fourth chapter will exclusively concentrate on the already hinted at challenges that the training of a routing network may encounter. These challenges also allow me to derive additional metrics under which routing may be evaluated.

As the most successful strategy to mitigate the problems of training a routing network is to provide meta-information (or any other form of low-dimensional projection), the empirical analysis of routing networks will be three-fold. In chapter five, I will start by presenting routing networks that rely on meta-information, and will explain why routing can become so much easier when combined with meta-information. In chapter six, I will describe different approaches of routing without meta-information, and will illustrate a procedure that can learn to emulate meta-information of the kind helpful to training a routing network. Finally, in chapter seven, I will do a thorough empirical analysis of other open questions for the successful implementation of routing networks.

Each chapter in this thesis corresponds to a contribution: In Chapter 2, when discussing related work, I will explain how routing generalizes existing approaches. In Chapter 3, I will introduce routing, a general framework for functional composition, as a major contribution. In Chapter 4, I will contribute an analysis of conditional computation approaches and why they are difficult to train. I will exemplify this on routing. In Chapter 5, I will introduce my contribution of conditioning routing on low-dimensional abstractions. In Chapter 6, I will discuss a relaxation of relying on low-dimensional abstractions, which is a contribution on its own. In Chapter 7, I will provide another contribution, a detailed empirical analysis

of numerous design and hyperparameter choices for routing. Many of the results in this chapter are negative, and I hope to give sufficient explanations and intuitions for when routing succeeds and for when it fails.

As most of this work was collaborative, I will use the scientific 'we' for the remainder of this thesis.

Definitions and Notation

We will now give an overview over the most relevant terms and the most relevant notation used throughout this thesis. First, some general terminology:

Architecture An *architecture* is a description of the parameterized function that takes an input and produces a prediction, without specifying the parameter values.

Model A model is an instantiated architecture, i.e., one specific architecture with specific parameter values. For this thesis, a model is generally composed of multiple modules.

Modules *Modules* are parameterized functions that take an activation and produce a new activation. Interchangeable terms are *transformations, layers,* and *function block.*

Activations *Activations* are any kind of intermediary result computed by a machine learning model. We may use them to also include the input to and the output of the model in certain contexts.

Downstream task As routing has nested optimization problems, the *downstream task* is the highest level task, which generally is the one we are actually interested in solving.

Tasks A *task* is one particular problem that we try to solve. For supervised learning, this is generally a dataset with tuples containing the input and the desired output.

Now, some notation:

$\langle x, y \rangle$ A dataset for supervised learning consists of input and output pairs, $\langle x, y \rangle$, where x generally refers to the input and y to the ground truth target output.

$\hat{y}$ — $\hat{y}$ describes the output of a model, an estimation of the target.

$\mathcal{L}$ — $\mathcal{L}$ will always be used for loss or objective functions. To specify, losses will be subscripted. For supervised learning, they will be a function between the model output and the target: $\mathcal{L}(\hat{y}, y)$. However, other losses can be defined (e.g., the REINFORCE loss for reinforcement learning).

π — As common in reinforcement learning, π describes a *policy*, i.e., a conditional distribution over actions. It is generally a function of the state s and it may be parameterized by parameters θ, written as $\pi_\theta(s)$.

h — h will always be an activation (for *h*idden). This may include the 0th activation, i.e., the input, and the final activation, i.e., the output.

a — As common in reinforcement learning, all actions (i.e., module selections) will be specified by a.

$E(\cdot)$ — The function E always described the expected value of its argument, which needs to be a random variable.

θ — When either the router or the modules are parameterized, their parameters are referred to as θ.

$P(\cdot)$ — $P(\cdot)$ is the probability function. $P(a|s, \pi)$ describes the probability of taking action a given state s and policy π.

$\mathbb{1}_d(k)$ — The $\mathbb{1}_d(k)$ function is the indicator (or one-hot) function. It will produce a vector of dimensionality d that equals 0 everywhere, except for index k, where it equals 1.

CHAPTER 2

RELATED WORK

> "Suitably designed modular architectures should learn faster than single networks because similar functions can be learned by the same network of the modular architecture, resulting in the benefits of positive transfer of training, whereas dissimilar functions can be learned by different networks, thereby avoiding the detrimental effects of negative transfer of training."
>
> Jacobs, Jordan and Barto 1991 [Jacobs et al., 1991a]

This quote from 1991 describes the motivation for "task-decomposition modular networks" [Jacobs et al., 1991a], arguably the first direct ancestor to routing networks. However, task-decomposition modular networks were not the only modular architecture introduced in the early 90s, and not even the most influential. The most influential were, and are, mixtures of experts architectures (MoE) [Jacobs et al., 1991b, Jordan and Jacobs, 1994], which are "soft" versions of routing networks. As such, they do not make exclusive hard selections, but instead learn weights (so-called 'gates') over the output of different modules. This simplifies their training immensely, because these gates retain differentiability and can thus be trained with backpropagation. However, they are not truly modular, because they do not use separate components for different problems. We will describe these and modern variations in more detail in Section 2.2.

Another, more recent line of research closely related to routing is "meta-learning". The "meta" in meta-learning describes a second learning process on top of the original model. This optimizes for the same target, but modifies some meta-level property of the

underlying process. These can include hyperparameters, gradient update procedures, and model structure and architecture. In this sense, routing is a generalization of meta-learning based architecture search, because it generalizes the assumption that all samples should be treated identically. This research will be described and related to routing in detail in Section 2.5.

As will become clear throughout this thesis, routing has a strong relationship to multi-task learning, partially because it was originally designed to navigate transfer between multiple tasks. In this context, the earliest work related to routing networks are, "task-decomposition modular networks" [Jacobs et al., 1991a], and arguably Richard Caruana's PhD thesis [Caruana, 1997], where he describes the benefits of sharing (and not sharing) parameters for transfer and interference of learning. While multi-task learning is not per se related to routing, it is certainly an application that has produced many different approaches with similar intent. We will therefore explore some of the more related ideas in Section 2.6.

This chapter starts with an overview of modular and compositional architectures *outside* of computer science, as the idea of modularity is a dominant paradigm in research into human cognition. After describing related work, some of which described above, it will finish with a section (Section 2.7) reviewing work on routing by other researchers.

2.1 Modular and Compositional Learning in Other Disciplines

The earliest debate with a direct impact on modular computation was probably in the cognitive sciences, because human learning relies on similar notions of modularity and compositionality. Of particular relevance is the work of Jerry Fodor [Fodor, 1975, 1983, Fodor and Pylyshyn, 1988], who postulated that human reasoning and learning are roughly structured along the structure-content divide of natural language [Fodor, 1975]. For him, specific areas of the brain encapsulate specific skills, while other areas have the more general task of assigning stimuli and intentions to their respectively specialized areas. Partially as

a consequence of the impact of his work, this remains a major paradigm in the cognitive sciences until today (Bechtel and Richardson [2010] provide a thorough overview over these developments until the late 2000s).

Additionally, studies in neurophysiology appear to confirm these beliefs, because deficits in high-level cognitive functions are regularly associated with impairments of particular regions of the brain that are regarded as modules, typically under the assumptions that these modules operate independently and, to varying degrees, that their effects are local [Farah, 1994, Shallice, 1988, Bechtel and Abrahamsen, 2002]. This relates to two more fundamental aspects of the organization of the brain: functional segregation of neural populations and anatomic brain regions as specialized and independent cortical areas, and functional integration, the complementary aspect that accounts for coordinated interaction [Tononi et al., 1994]. Recently, Kell et al. [2018] have been able to replicate human cortical organization in the auditory cortex using a neural model with two distinct, task-dependent pathways corresponding to music and speech processing. When evaluated in real world tasks, their model performs as well as humans and makes human-like errors. This model is also able to predict with good accuracy fMRI voxel responses throughout the auditory cortex, which suggest certain correlation with brain-like representational transformations. Kell et al. [2018] use these promising results to suggest that the tripartite hierarchical organization of the auditory cortex commonly found in other species may also become evident in humans once more data is available and more realistic training mechanisms are employed. In particular, their results suggest that the architectural separation of the processing of the streams is compatible with functional segregation observed in the non-primary auditory cortex [Kell et al., 2018, and references therein].

2.2 Mixtures of Experts and 'Soft' Routing

The aforementioned Mixtures of Expert (MoE) architectures [Jacobs et al., 1991b, Jordan and Jacobs, 1994] do not make hard selections, but instead compute a weighted sum over the outputs of all the modules. While original work only focused on a single layer to be computed, modern approaches allow for a sequence of models to be selected, making them yet more similar to routing networks. (Which is why, they have been referred to as 'soft' routing models). That is, given a fixed set of modules $\mathcal{M}$, they compute the following output for a given activation h:

$$f_{MoE}(\mathcal{M}, h) = \sum_{m \in \mathcal{M}} w(h)m(h) \tag{2.1}$$

Where $w(h)$ is a learned weight (or gate) to the modules' output. For top-k MoE architectures, we have:

$$f_{MoE_k}(\mathcal{M}, h) = w_1(h)m_1(h) + w_2(h)m_2(h) + ... + w_k(h)m_k(h) \tag{2.2}$$

with the constraint that $m_1 \neq m_2 \neq ... \neq m_k$, and with $w_1(h)...w_k(h)$ generally as the k highest valued weights. This differs from routing, which computes

$$f_{routing}(\mathcal{M}, h) = m(h) \tag{2.3}$$

where m is oftentimes chosen as the highest-valued model for input h. Even though routing could be considered an MoE architecture where all except one weight equals zero, this greatly distorts routing conceptually. If the focus remains on making a single hard decision, it allows the principled investigation of problems common to decision making, such as credit assignment and exploration.

This critique extends to several recent approaches that do not simultaneously compute the output over all modules, but instead only over a subset. As a result, sparse or 'top-k'

mixtures of experts architectures [Shazeer et al., 2017, Ramachandran and Le, 2019] must include special procedures for adding noise to the system and load balancing to achieve strong performance [Shazeer et al., 2017]. Consequently, it makes sense even with a sparse system to model the selection of a subset of experts as an explicit reinforcement learning problem. For example, in [Bengio et al., 2015] each "expert" is gated by a Boolean routing decision that is modeled as a reinforcement learning problem.

2.2.1 Conditional Activations of Neurons

Other related work for neural architectures includes what we might term "conditional computation on a per-neuron basis", of which there are two types. In 'uninformed' conditional computation, the inclusion of a neuron in the computation does not depend on the input sample. This includes regularization approaches, such as L0 regularization [Louizos et al., 2017], where the network is regularized to only use the same subset of neurons for all samples. It also includes random regularization approaches such as Dropout [Srivastava et al., 2014], where the network only uses a random subset of samples.

The second group can be called 'informed' conditional computation, as the function triggering a single neuron depends on the current sample. One example is [Bengio et al., 2015], where each neuron can be masked by having a per-neuron input-dependent trigger function, to merge networks for different tasks. Another interesting example of conditional computation that falls into this group are activation functions, in particular ReLUs [Nair and Hinton, 2010], because these condition local composition based on the input to the model. However, these are only a special limited case of modular and compositional computation, as they do not allow for a separation of the interpretation of each module, and the module composition. But this can be one of the greatest advantages of compositional approaches such as routing networks. As an example, consider routed visual question answering approaches such as [Andreas et al., 2015] or [Wu et al., 2019]. There, the composition of the modules is determined by the question, but the interpretation of the images is determined by

the modules. This is also true for multi-task routing approaches [Rosenbaum et al., 2017, Cases et al., 2019] and multi-task per-neuron compositional approaches [Bengio et al., 2015], because the composition is decided by the task information, but the module interpretation is decided by the visiual aspects of the samples.

2.3 Other forms of Conditional Computation

Conditional computation describes a property of a deep learning model in which not all neurons are required for all activations. Instead, each neuron will only be activated for some inputs. This was theorized first in [Bengio, 2013], and introduced in [Bengio et al., 2013]. Functional composition, i.e., routing, is both more general and more specific. Because it delimiters the exact set of active parameters by setting them module-wise, it is more specific. But routing is more general in that it allows for a flexible ordering of the neurons. However, routing can be seen as a model of recursive conditional computation. A similar idea was applied in a reinforcement learning context in [Bengio et al., 2015].

Another semi-modular approach that combines intuitions behind conditional computation on a neuron level and on a module level is 'Pathnet' [Fernando et al., 2017]. Similar to routing, pathnet also relies on complete modules in the form of neural network layers. Similar to a top-k MoE architecture and models of conditional computation on a neuron level, several of these models are then combined to form the new output. Dissimilar to routing, the ordering of these models is fixed. These paths are trained using genetic algorithms, while the modules are trained with SGD.

2.4 Numerical Analysis

Machine learning has a very close relationship to numerical analysis, and many concepts of one have corresponding concepts in the other. This includes a specific application to routing and related approaches in modular computation, because they are strongly re-

lated to the concept of *local approximations* in numerical analysis. Local approximation techniques are those that find a different approximation for a different subset of samples, as depicted in Figure 2.1. This makes the entire approximation, composed by the sum of its local parts, highly non-linear, which is an oftentimes desireable trait. In this sense, routing is a local approximation technique, as it also finds different approximations for different subsets of samples. More precisely, it is a local approximation technique where the clusters are found dynamically, and the number of clusters is limited by the number of combinations determined by the width and depth of the router design. Additionally, each approximation may share a different amount of parameters with other approximations.

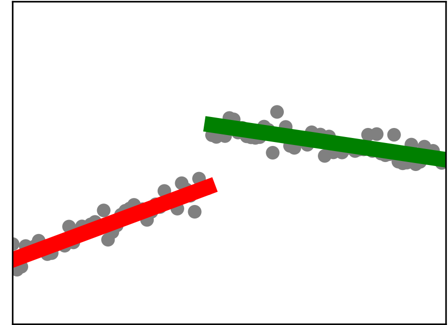


Figure 2.1: A dataset with two subsets, each approximated by a different function.

2.5 Meta-learning and Architecture Search

Routing networks are an extension of a popular line of recent research focused on automated architecture search, or more generally, architecture-based meta-learning. General meta-learning augments the learning process with additional learning processes that optimize for the same target by modifying the underlying learning process. This modification may occur on several levels.

For gradient-based meta-learning (obviously only applicable to differential machine learning models), the gradient direction or size is in itself another optimization problem. There are several kinds of motivation for this line of work. One is that regular stochastic gradient descent may not sufficiently accommodate the structure of the given problem, so that task-specific gradient steps will yield superior learning performance [Andrychowicz et al., 2016]. Another motivation may be taking gradient steps that yield parameter states

that can be easily adapted for few-shot learning [Finn et al., 2017, Nichol and Schulman, 2018].

Another very important line of work is the automated learning of optimal hyperparameter values. While naïve solutions, i.e., hyperparameter-sweeps, are possible (and probably the de facto standard for most published work), the process of searching for optimal hyperparameters can itself form an optimization problem [Feurer et al., 2015, Snoek et al., 2012].

This brings us to architecture search. Here, the search space of the optimization problem is the exact configuration of the model. The goal is to reduce the burden on the algorithm designers by automatically learning black box algorithms that search for optimal architectures. The search space may be simple and only contain different values for the hidden dimension of an architecture, or range to complex properties such as the choice and order of parameterized transformations (layers, in the context of neural architectures). This line of meta-learning in particular is very relevant to routing.

Architecture search automatically finds the optimal architecture of a model for an entire dataset, while routing finds the optimal architecture for privileged subsets, possibly even for each individual sample. The first research along this line was a very straightforward search utilizing reinforcement learning: Construct an architecture, train it until convergence, evaluate its performance and use that (accuracy) score as a reward, then repeat [Zoph and Le, 2017, Baker et al., 2017]. The problem with this approach is its computational cost, because even training one model to convergence can take a powerful computer several days.[1] Other approaches rely on evolutionary algorithms [Miikkulainen et al., 2017, Fernando et al., 2017], approximate random simulations [Brock et al., 2017], and adaptive growth [Cortes et al., 2016]. Liang et al. [2018] even introduced an evolutionary algorithm approach

[1] Zoph and Le [2017] in particular were famous for a while for relying on an unprecedented amount of computation.

targeting multi-task learning that comes very close to the original formulation in [Rosenbaum et al., 2017].

Formally, both routing and architecture search define so-called *bilevel optimization* problems, as one optimization problem is nested in another one. The architecture search bilevel optimization problem can be defined as (with M as the set of possible models/architectures μ^* as the optimal architecture, Θ the possible values of the parameterization, θ^* the optimal set of parameter values, $\mathcal{L}$ the downstream loss function, and DS as a given dataset):

$$\mu^* = \underset{m \in M}{\text{argmin}} \underset{\langle x,y \rangle \in DS}{E} (\mathcal{L}(m_\theta(x), y)) \tag{2.4}$$

$$\text{subject to:} \quad \theta = \underset{t \in \Theta}{\text{argmin}} \underset{\langle x,y \rangle \in DS}{E} (\mathcal{L}(m_t(x), y)) \tag{2.5}$$

While routing can be defined as:

$$\theta^* = \underset{\theta \in \Theta}{\text{argmin}} \underset{\langle x,y \rangle \in DS}{E} (\mathcal{L}(\mu_\theta(x), y)) \tag{2.6}$$

$$\text{subject to:} \quad \mu_\theta = \underset{m \in M}{\text{argmin}} \, \mathcal{L}(m_\theta(x), y) \tag{2.7}$$

The main differences are: (1) The order or the nesting. For architecture search, optimizing the parameters is nested in searching over architectures. For routing, optimizing over models (and not architectures, as they are conditioned on specific θ) is nested in optimizing the parameters of the modules. (2) While architecture search optimizes the parameters over the entire dataset, routing optimizes the choice of model for each sample individually.

One-shot neural architecture search [Brock et al., 2017, Pham et al., 2018, Bender et al., 2018] tries to simplify the costly search process of full architecture search by jointly evaluating large groups of models that share parameters. In this, it becomes even closer to routing, as all routing models also share parameters, as the model parameters are optimized in the outer optimization process (equation 2.6). However, one-shot architecture search still constraints the search space by having more restrictive choices of which modules are

available at a given step. Additionally, it still works under the assumption that all samples in a dataset should work on the same model. Routing, not having these restrictions, can consequently be seen as a generalization of one-shot architecture search [Ramachandran and Le, 2019].

2.6 Multi-task Learning

As will become clear throughout this thesis, routing has the ability to pick up on local modes of multi-modal distributions, and learning different routing paths for each. This can be seen as an implicit multi-task approach, because multi-task learning is a special case of multi-modal learning when the different modes are known, and labeled, a priori. The general motivation behind this is already articulated in "task-decomposition modular networks" in 1991 [Jacobs et al., 1991a], as illustrated by the introductory quote.

Consequently, there are parallels between routing and several lines of research in multi-task deep learning. In fact, routing was first proposed to learn paths for different tasks, balancing positive and negative transfer [Rosenbaum et al., 2017].

Because modular architectures generally offer solutions to balancing positive and negative transfer, many approaches related to routing have been applied to it. These include mixtures of experts models, or more generally soft parameter sharing models. While these have been considered in the multi-task and lifelong learning setting [Stollenga et al., 2014, Aljundi et al., 2016, Misra et al., 2016, Ruder et al., 2017, Rajendran et al., 2017], they do not allow for nearly the level of specialization of those based on routing. This is because they do not eliminate weight sharing across modules and instead only gate the sharing. In practice, this still leads to significant interference across tasks as a result of a limited ability to navigate the transfer-interference trade-off [Riemer et al., 2019] in comparison to models that make hard routing decisions.

2.7 Existing Work on Routing

Routing networks as a general paradigm of composing trainable transformations were first introduced in [Rosenbaum et al., 2017]. Since then, there have been several approaches extending them.

Chang et al. [2019] show that a vanilla routing network can learn to generalize over patterns and thus to classes of unseen samples if it is trained with curriculum learning. Their experiments include cases of simple formula parsing and image rotation. They show that, if trained correctly, routing can generalize to length $k + 2$ formulas after only being trained on length k formulas. Similarly, they show that routing can generalize to rotating images by more than some angle α, after only being trained to rotate images up to a rotation of α.

Kirsch et al. [2018] identify one of the challenges of compositional computation, collapse, and develop a new policy gradient-based routing algorithm that utilizes EM techniques to alternately group samples to transformations, and then applies them. This allows the model to learn non-collapsing distributions over the different model selections.

Ramachandran and Le [2019] investigate another challenge to routing, architectural diversity over transformations. Although they are using a top-k routing approach, and thus strictly speaking a “soft” version of routing, they make several compelling arguments as to how diversity can help a routing network learn more expressive configurations. They investigate their technique on Omniglot, reaching impressive state of the art performance.

Cases et al. [2019], which is our work and is presented in detail in Section 5.3, shows how routing can be paired with high quality linguistic annotations to learn compositional structures that maximize utilization of task-specific information. Departing from the original multi-task learning approach, this work also introduces the first successful application of a ‘dispatcher’.

Alet et al. [2018] combine a routing-like approach with other meta-learning approaches, to allow for quick adaptation to new tasks. However, this approach relies on *pre-trained* composable transformations.

Because the most successful applications of routing depend on the availability of meta information to cluster the data, we investigated alternative clustering approaches in [Rosenbaum et al., 2019a]. This ‘dispatching’ was originally introduced in [Rosenbaum et al., 2017], but complicated the training too much to allow for stable architectures. In [Rosenbaum et al., 2019a], we develop techniques to stabilize the training.

CHAPTER 3

ROUTING

Routing describes a general framework of repeatedly selecting trainable function modules with a trainable *router*. As such, arbitrary components of a machine learning model can be routed, as long as the model is composed of a sequential application of functions in a set of (compatible) functions. We will focus here on routing networks, where the learnable function modules are neural networks and parts thereof. The router receives the state of the computation, the current activation – the input x at step 0 – and evaluates it to select the best function. This produces a new activation, which then triggers a new recursion through the router and possibly the modules. If the router decides that the result has been processed sufficiently, the last activation will be handed to other computations or interpreted as the output of the model. This result will then be used to jointly train the function modules and the router.

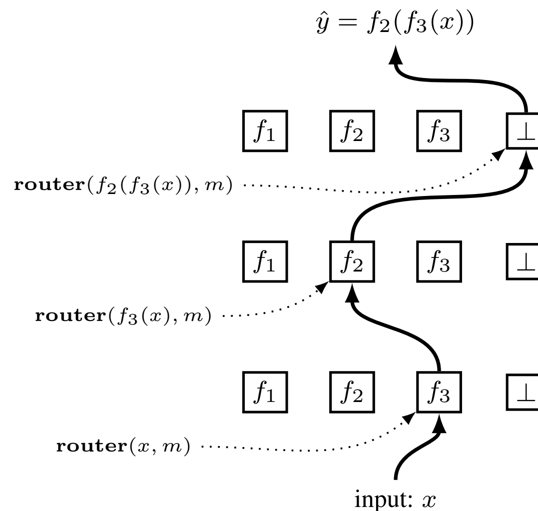


Figure 3.1: Routing (forward) Example including a termination action $\perp$.

Algorithm 3.1: Routing Forward. The **router** function computes an action from the state, and can thus be any kind of hard decision-making algorithm. $\perp$ is a termination action, and **module**$_k$ is the selected module or function from the set of available modules. This algorithm may be modified to accommodate constrains on the subset of modules applicable to the current activation.

```
   input  : x ∈ R^d, d the representation dim; m possible metainformation;
   output : h - the tensor f_{a_1} ∘ f_{a_2} ∘ ... ∘ f_{a_t}(x)
1  h ← x
2  while True do
3      a ← router(h, m)
4      store tuple ⟨h, m, a⟩
5      if a = ⊥ then
6          return h
7      else
8          h ← module_a(h)
```

Because routing relies on *hard* decisions to select the modules (as opposed to 'soft' decisions, where several modules are activated and combined in different ways), training algorithms for the router are limited. While other approaches are conceivable (in particular genetic algorithms and other stochastic training techniques), we focus on reinforcement learning (RL) and Stochastic Reparameterization in this work. The main part of this chapter was introduced in Rosenbaum et al. [2017], with some additional work introduced in Cases et al. [2019], Rosenbaum et al. [2019b].

3.1 Reinforcement Learning

Reinforcement learning describes a general machine learning paradigm in which an agent that makes a series of hard decisions is trained by providing simple scalar feedback, the reward. Applied to routing, the general outline is that the output of the model $\hat{y}$ has to be translated to a reward r which is then used to improve the router's policy π.

3.1.1 Routing MDPs

Markov decision processes (MDPs) are a common model for formally describing sequential decision-making problems. While we will argue in Section 4.6 that existing attempts to model compositional computation are problematic, we will still adopt a simple formalism to permit discussion of some relevant concepts.

Given a routing MDP $M = \langle S, A, R, P, \gamma \rangle$, a set of applicable modules F, a termination module $\perp(x) = x$ that terminates the routing trajectory but that otherwise acts as the identity function, the space of samples X, the space of activations H (i.e., the space of applications of members of M to X), and the space of possibly available meta-information I, we can define:

$$\text{the states } S = (X \cup H) \times I$$

$$\text{the actions } A = F \cup \{\perp\}$$

$$\text{the transition probability } P(s, a, s') = \begin{cases} 1, & \text{if } s = h, a = f, s' = m(h), h \in S, m \in A \\ 0 & \text{otherwise} \end{cases}$$

$$\text{the discount factor } \gamma = 1.$$

The reward function R can freely be designed by the model designer and is not defined any further for now. Solving an MDP consists of finding an optimal policy $\pi^*(\theta)$ (parameterized by parameters θ) that maximizes the objective, or score function, J corresponding to expected cumulative return of all actions:

$$J(\pi(\theta)) := E\left(\sum_a \gamma r_a | \pi(\theta)\right) = E\left(\sum_a r_a | \pi(\theta)\right).$$

It can be useful to express the score function in terms of the cumulative return $g_t^\pi := \sum_t \gamma r_a | \pi(\theta)$:

$$J(\pi(\theta)) := E\left(g_t^\pi\right).$$

3.1.2 Reward Design

One of the most important questions when modeling a problem as an MDP is the question of how to design the reward function R. For a routing network, there are two types of rewards to be considered. The first is the final reward that reflects the model's performance in solving the main problem, i.e., of predicting a sample-label. The second are different forms of regularization rewards that can either be computed for entire trajectories or as immediate responses to individual actions.

3.1.2.1 Final Rewards

The final reward r_f models the overall performance of the model for a sequence of routing decisions. If the underlying machine learning problem is a classification problem, then the outcome is binary – correct or false – and an obvious choice is a simple binary reward of ± 1 corresponding to the prediction agreement with the target label.

Because the objective for the training of the modules is a loss function $\mathcal{L}_m(\hat{y}, y)$, a different reward design for the same objective can be derived as the negative of the target loss: $r_f = -\mathcal{L}_m(\hat{y}, y)$. The reward has to be the negative of the target loss because, by convention, we minimize model losses but maximize RL rewards. In addition, this allows for a natural application of routing networks to regression problems, because these cannot be translated into a simple binary ± 1 reward. While the second reward design seems like the better choice, because it both richer in information and a more principled meta-learning objective that coincides with the main model's objective, it does not necessarily perform better in practice depending on the problem of focus.

3.1.2.2 Regularization Rewards

Because the reward signal is sparse for complex compositions, it can also be quite helpful to incorporate additional rewards to act as intrinsic rewards or regularizers of the choices made by the router. One option is to model problem-specific information in this reward – e.g., to reflect the cost of computation of choosing a specific function, or to incorporate domain-

knowledge, such as intuitions as to which model should be preferable. The second option, as we introduced in [Rosenbaum et al., 2017], is to use this regularizer to incentivize transfer between different kinds of samples. There, the reward can be defined to correlate with how often a particular function a is chosen within some window w, $C(a) = \frac{\#a}{\sum_w samples}$. This motivates the router to choose, and thereby share, that function for a wider set of samples. This reward is defined as $R(a) = \frac{\rho}{t}C(a)$, with a ratio ρ, normalized by the trajectory length t, such that even for long trajectories this reward may be limited to be smaller than the final reward r_f. In [Rosenbaum et al., 2017] we only investigate values of $\rho \in [0, 1]$. However, this can lead to a lack of variety in decision-making, as the selection collapses, as discussed later in Section 4.5 in detail. To compensate, we introduced a different set of values for ρ in [Cases et al., 2019], $\rho \in [-1, 0]$, with the goal of incentivizing diversity of decision-making.

3.1.3 Algorithms

Generally speaking, reinforcement learning algorithms fall into one of two groups. The first are value based approaches, in particular q-value based approaches. The second are policy-gradient based approaches.

3.1.3.1 Value-based RL Algorithms

The first class of reinforcement learning algorithms are so-called value-based approaches that generally learn a state's q-value to define a policy. A q-value is an estimate of the value, or expected discounted return, of taking a particular action in a particular state:

$$Q^{\pi}(s, a) = E\left(\sum R_t | s_t = s, a_t = a, \pi\right) \tag{3.1}$$

These state-action values can be straightforwardly converted into policies, i.e., action-instructions by simply always selecting the highest value q-value for a given state. Discussing the arguably most famous value-based RL algorithm, q-learning, will be useful for later

discussions of the properties of routing. For q-learning, the q-value update rule is defined as (with the already introduced notation, and with α as the learning rate):

$$Q^{\pi}_{t+1}(s,a) = Q^{\pi}_{t}(s,a) + \alpha \left(r(s,a) + \gamma \max_{a} Q^{\pi}_{t}(s',a) - Q^{\pi}_{t}(s,a) \right) \tag{3.2}$$

However, this rule only defines how to update the value estimates, not how to retrieve the state, action, reward, next state tuples required to perform the update. To retrieve these, a common approach is called ε-greedy action selection, which is defined for q-learning as:[1]

$$\pi(s,a) = \begin{cases} 1-\varepsilon, & \text{if } a = \operatorname*{argmax}_{a'} Q^{\pi}(s,a') \\ & \text{(For ties, choose one action randomly)} \\ \frac{\varepsilon}{|\mathcal{A}|-1} & \text{otherwise.} \end{cases} \tag{3.3}$$

In combination, these two formulas fully describe the ε-greedy q-learning algorithm. As such, this algorithm has several properties worth discussing. First, it illustrates a common and important concept in reinforcement learning, bootstrapping. Simply put, bootstrapping means learning to recursively improve guesses from other guesses over time. In this case, bootstrapping means that we somehow initialize q-values for all states first (possibly very badly), and then improve their accuracy by computing their difference to the guess at the next state.

Another important property of q-learning is that it is “off-policy”. This means that q-learning learns Q^* directly, even when doing ϵ-greedy exploraiton. This property comes from the max-operation, because we improve a q-value guess for $Q^*_t(s,a)$ even if we do not select the highest-value action in $Q^{\pi}_t(s',a)$.

Lastly, q-learning allows us to illustrate the dilemma between exploitation and exploration in reinforcement learning. This problem intuitively captures the dilemma of relying on known good solutions (exploitation) at the cost of possibly better, but unknown – and thus also possibly worse, solutions (exploration). While it might be appealing to err on

the side of caution and to only exploit, it is possible that the initial guess will determine all future policies, and that it is far inferior to many other solutions. So learning in any meaningful way requires exploration.

There are several alternatives to learning q-values than the above algorithm. One, called SARSA [Rummery and Niranjan, 1994], is an on-policy version of q-learning (on-policy in that it estimates q-values it actually sampled, i.e., Q^π). Another, called Advantage Learning [Baird III, 1993], relies on the fact that we can decompose the q-values for the actions of a given state into an on-policy average expectation for that state, i.e., its *value* $V^\pi(s) = E\left(\sum_t r_t | \pi\right)$, and the per-action deviations, the per-action *advantage* $A^\pi(s,a)$, from that average:

$$Q_t^\pi(s,a) = V_t^\pi(s) + \underbrace{(Q_t^\pi(s,a) - V_t^\pi(s))}_{A_t^\pi(s,a)} \quad (3.4)$$

$$= V_t^\pi(s) + A_t^\pi(s,a). \quad (3.5)$$

That is, $A_t^\pi(s)$ measures how much a particular q-value for a state differs from the on-policy expected return of that state. Consequently, a greedy policy can be determined given only $A_t^\pi(s)$. The general advantage of this separation is that the explicit value component can compensate for any bias occurring during the learning of the q-values. As this benefit is important for routing, we will now try to explain it further. Imagine that only one action a_1 was taken all the time in some state s_k, as its estimate might be slightly higher than the estimate for all other actions, even though the true expected return may be higher for an action different from a_1. Now additionally consider that the estimates for later states get better and better, and thus also the q-value for a_1. For regular q-learning, this can ‘lock’ the selection to a_1 for a considerable amount of time during training, as all of the improvement in decision-making is attributed to a_1 when in state s. However, as we know, another action a_2 might indeed be better. For advantage learning, $V_t^\pi(s)$ will offset the

inductive bias over training, thereby making the gap in the estimates of $Q_t^\pi(s, a_1)$ and $Q_t^\pi(s, a_2)$ smaller, allowing the algorithm to more quickly update it's estimates to correctly reflect a_2's superiority. We will relate this inductive bias phenomenon to routing in a later section.

3.1.3.2 Policy Gradient based RL algorithms

The second class of reinforcement learning algorithms works by computing the policy directly, and not by way of a value function (although most recent algorithms combine both). Because a policy is a mapping from states and actions to probabilities, this implies that policy gradient based learning optimizes this distribution directly to maximize J. One immediate advantage is that this allows stochastic policies to be learned, as can be required by e.g., learning policies for specific forms of games.

The most important policy gradient algorithm is REINFORCE [Williams, 1992], because it was the first policy gradient algorithm that defines the gradient for parameterized policies, i.e., policies that are not stored as a table but by a function approximator. This algorithm has become particularly relevant because it allows losses for differentiable models to be defined, such as deep networks. The standard formulation of this algorithm is, for a score function J, depending on a policy π parameterized by parameters θ, and a trajectory $\mathcal{T}$ of state, action and reward tuples:

$$\nabla J(\pi(\theta)) = \nabla E\left(\sum_a \gamma r_a | \pi(\theta)\right) \tag{3.6}$$

$$= \nabla E\left(\sum_a Q^\pi(s,a)\pi(a|s,\theta)\right) \tag{3.7}$$

$$= E\left(\sum_a Q^\pi(s,a)\nabla_\theta \pi(a|s,\theta)\right) \tag{3.8}$$

$$= E\left(\sum_a Q^\pi(s,a)\pi(a|s,\theta)\frac{\nabla_\theta \pi(a|s,\theta)}{\pi(a|s,\theta)}\right) \tag{3.9}$$

$$= E\left(\underbrace{\sum_a Q^\pi(s,a)\pi(a|s,\theta)}_{=g_t^\pi}\frac{\nabla_\theta \pi(a|s,\theta)}{\pi(a|s,\theta)}\right) \tag{3.10}$$

$$= E\left(g_t \frac{\nabla_\theta \pi(a|s,\theta)}{\pi(a|s,\theta)}\right). \tag{3.11}$$

This equivalence defines the gradient for the parameters that follows the gradient of the score function. As this can be used to immediately optimize the policy to maximize the score function, we can derive the REINFORCE update (with some learning rate α):

$$\theta' = \theta + \alpha g_t^\pi \frac{\nabla_\theta \pi(a|s,\theta)}{\pi(a|s,\theta)} \tag{3.12}$$

In particular when combined with a differentiable model as a function approximator, this update can be used to backpropagate along the parameters of the network. Note that REINFORCE does not require a specific exploration strategy because it is stochastic by nature.

However, REINFORCE suffers from extremely high variance, making it often unusable. Consequently, many extensions have been introduced to correct for the variance by introducing different forms of bias. This bias may correct for the range of the reward by introducing a 'baseline' that is subtracted from the reward. This baseline may be a simple average, or a more sophisticated estimation of the current state's expected reward. In practice, these extensions improve convergence properties dramatically over vanilla REINFORCE.

3.1.3.3 RL algorithms for routing

With these two classes of algorithms in mind, we will now make some general comments about their applicability to routing. We will offer a thorough empirical investigation of these two different approaches to decision-making in chapter 7.

In existing work on routing, several RL algorithms have been successfully used to train routing policies. Rosenbaum et al. [2017] compare a variety of algorithms, but settle for a multi-agent algorithm, fitting their multi-agent approach to multi-task learning. Cases et al. [2019] find that plain Q-Learning performs the best for their domain. Kirsch et al. [2018] extend a REINFORCE based algorithm with an alternating, EM-like training for modules and router that also optimizes for module diversity and does lookaheads. Meanwhile, Chang et al. [2019] use Proximal Policy Optimization.

While this might seem to suggest that there is not necessarily a "best" reinforcement learning algorithm, there are some important considerations to be made when designing a router.

Replay Buffers Routing is generally incompatible with 'replay buffers'. A replay buffer is a cache of previously encountered state, action, reward tuples. They are motivated by the fact that the training of a neural network requires its training samples to be independently drawn from their distribution, as training on related samples makes the training prone to get stuck in local optima. However, for RL trajectories, the last k samples are generally correlated. This motivates replay buffers, because they 'break up' this correlation by collecting a large number of trajectories first, to then draw individual tuples independently. This works as long as the states buffered actually contain useful training information. If the policy approximator, or the interpretation of the states, changes too much, the replay buffer is invalidated and new samples need to be drawn. For routing, with its constantly changing modules, this invalidation happens after every backpropagation step through the modules, making replay buffers generally useless for training routing networks.

Non-Stationarity Another important property of the routing decision-making problem is the intrinsic non-stationarity of the problem, because the interpretation of the modules changes every time they are updated. This, in turn, means that not only are the activations passed on to the router change, but also that the expected value of selecting a particular module for a particular activation changes over time. One solution, in theory, would be to allow the router access to all parameters of all of the modules from which it can choose from. While this would get around the non-stationarity, the state space would become unmanageable large.

The main way this poses a problem to traditional RL algorithms might be best explained by the following example: Given a router and a set of modules, routing causes the module parameters to change, and thereby affects the expected value for each of the modules given a specific state sample or activation; however, it *also changes the value of the modules for other activations*. This means that the estimates of the modules' values will now be erroneous for all other samples and activations. More specifically, as the quality of the modules generally increases over time, their values will most likely be underestimated. Because routing works best for low levels of exploration, they will probably not selected enough to correct their value estimates. Consequently, this may "lock" the router in its selection (see also Section 4.3 for a discussion of this problem) early on.

Intuitively, this could be compensated for by a bias-correcting term that offsets the value estimates for the models by their general increase in value. One such approach is the previously discussed advantage learning, as it specifically models such a bias term. We will discuss this in more detail in Section 7.1.

3.2 Stochastic Reparameterization

When it comes to hard decision-making in differentiable architectures, stochastic reparameterization was recently introduced as an important alternative to Reinforcement Learning. While earlier work on stochastic reparameterization focused on low-dimensional

stochastic distributions, [Maddison et al., 2016] and [Jang et al., 2016] discovered that the concrete – or Gumbel Softmax – distribution allows to reparameterize k dimensional distributions (We will also refer to the corresponding decision-making algorithm as "Gumbel").

In difference to RL based approached that rely on a final scalar feedback that is translated into the policy's gradient, Gumbel can be added as a neural network layer and trained via back-propagation *immediately* [Jang et al., 2016]. This is a highly appealing property for routing, because it simplifies design and implementation tremendously.

Maddison et al. [2016] already note that the concrete distribution and its reparameterization do not yield an unbiased estimate of the gradient. The REINFORCE estimator [Williams, 1992] for the gradients of the score function, on the other hand, is known to be unbiased (but has very high variance). Therefore two more techniques were introduced to estimate the gradient of discrete random variables. The first, REBAR [Tucker et al., 2017] compensates for REINFORCE's variance by introducing a control variate based on the concrete distribution. The second, RELAX [Grathwohl et al., 2018], generalizes this for non-discrete and non-differentiable stochastic functions, and allows a neural network to model the control variate.

To better understand the relative performance of reparameterized approaches when compared to a reinforcement learning algorithm, we performed the following analysis: Given a known distribution as a policy π parameterized by θ over a known reward function r, we can analytically compute the value of the score-function $J(\theta) = E(r|\pi(\theta)) = r \cdot \pi(\theta)$, where $\cdot$ denotes the inner product. We also analytically compute the gradient for all parameters θ. We know that these gradients are the ground truth that policy gradient and reparameterization approaches only approximate. We can then approximate the same gradients $\frac{\delta J}{\delta \theta}$ using REINFORCE, Gumbel and RELAX, and finally compute the average difference between the ground truth values and the approximations.

For Figure 3.2, we used this approach to compute both the average gradient differences and the average variances over all parameters. More specifically, we initialized a random reward function (where each reward is uniformly sampled from $[0, 1]$), of dimensionality k. We also randomly initialized a policy of dimensionality k, parameterized by θ. After computing the ground truth gradient for a given pair of reward function and policy, we used the same reward function and policy to sample the gradients for θ $22 * k$ times (so that on average each action may be sampled equally often, even with increasing dimensionality). For each k, we computed and approximated the gradients for 22 reward and 22 policy values, totalling over $10000 * k$ datapoints for each point in Figure 3.2.

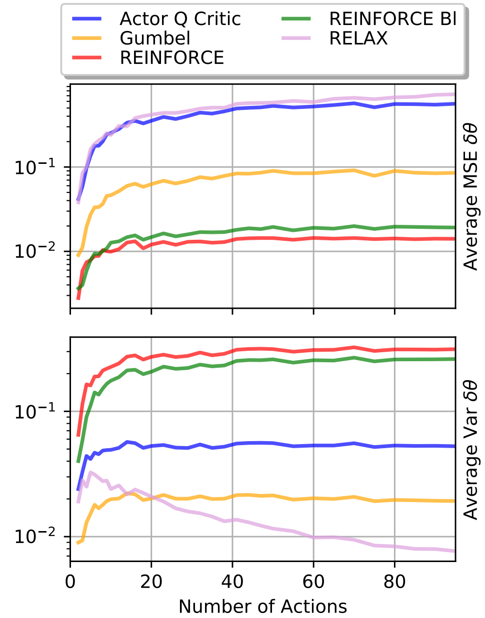


Figure 3.2: Gradient statistics. The REINFORCE Baseline is computed as $\hat{r} = (1 - \alpha)\hat{r} + \alpha r$ with $\alpha = 0.1$. The temperature parameter for all approaches is 0.5, which appears to be a common choice in the literature.

Figure 3.2 shows the mean squared error between the ground-truth gradient values and the sampled values using reparameterization and different RL policy gradient algorithms on top, and the corresponding variances on the bottom. While the gradients estimated by the Gumbel softmax trick are of *much* lower variance, they are also biased, with an average MSE larger than the MSE of the Policy Gradient algorithms. This is consistent with the analysis in [Maddison et al., 2016]. As for RELAX, we found very low variance for high dimensionalities, but – interestingly enough – a much higher MSE than REINFORCE and even than Gumbel, suggesting a higher bias. However, this may stem from the problem that RELAX relies on a trained surrogate network, which is difficult to train for these single-sample experiments. While we tried to accommodate this requirement by training this network over multiple samples first, this might not suffice to fully initialize RELAX.

However, we evaluate numerous different decision-making algorithms in the context of routing in Section 7.1. There, RELAX and its surrogate network are trained over millions of iterations, without yielding results that allow clear conclusions about how reparameterization techniques relate to REINFORCE based approaches in practice – at least in the context of routing.

3.3 Training

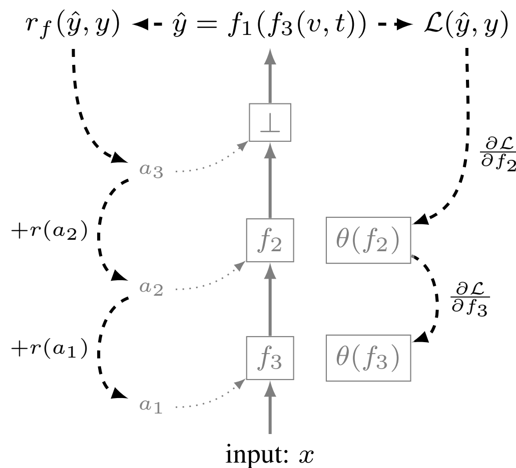


Figure 3.3: Routing (backward) Example

---

input : The network's output $\hat{y}$, the ground-truth target y, and the decision making trajectory $\mathcal{T}$

1 compute the model loss $\mathcal{L}(\hat{y}, y)$
2 compute the final reward $r_f(\hat{y}, y)$
3 **for** *each tuple* $\langle s_k, a_k, r(a_k), s_{k+1}, a_{k+1}\rangle$ *in* $\mathcal{T}$ **do**
4 compute the Bellman error (or other corresponding loss) for the current tuple, and add it to $\mathcal{L}_{RL}$
5 backprop on $\mathcal{L} + \mathcal{L}_{RL}$ and update using SGD

Algorithm 3.1: Backward step: Training of a routing network

The training of a routing network is illustrated in Figure 3.3 and the corresponding algorithm is Algorithm 3.1. The core idea is that the training of the router and of the

modules happens simultaneously, after completing an episode, i.e., the forward pass for one example (or a batch thereof). After the network has been assembled in the forward pass, the output of the network is translated into a loss for the module parameters and a final reward for a reinforcement learner. That reward, in combination with any accumulated per-action rewards, can be used to define a training loss for the router, either a Bellman Error, a negative log probability loss or some other loss function that is used to train a decision maker. The resulting losses can be added and then backpropagated along the decision-making parameters to define a gradient for each parameter in the model. Backpropagating along the sum of the losses allows for higher mini-batch parallelization in the update process of the network.

Finally, a gradient descent style algorithm can use these gradients to derive a new set of parameter values. However, it is worth noticing that routing networks can be highly sensitive to the choice of optimization algorithm. We will discuss this further in Section 7.4.4.

3.4 Architectures

3.4.1 Model Architectures

In general, any machine learning model can be routed. However, for non-layered architectures where only one decision needs to be made, routing collapses to the better investigated model selection problem. For layered architectures, any single layer can be routed by creating parallel copies of the respective layer, each with different (hyper)parameters. Existing routing architectures have included routing several fully connected layers with identical hyperparameters but different parameters [Rosenbaum et al., 2017, Kirsch et al., 2018, Chang et al., 2019, Cases et al., 2019], routing entire convolutional networks [Kirsch et al., 2018], routing the hidden to hidden transformation of recurrent neural architectures [Kirsch et al., 2018, Cases et al., 2019], routing the input to hidden transformation [Cases et al., 2019], and routing word representations [Cases et al., 2019]. However, only Ra-

machandran and Le [2019] have studied the effect of routing among modules that have different architectures.

This puts two non-competing perspectives on routing. In the first, the focus is on controlling parameter sharing and training specialized modules for different samples. In the second, routing tries to optimize the architecture on a per-sample basis. Ultimately, we want both: routing networks that can choose between different architectures, but that can also choose not to share specific parameters.

3.4.2 Router Architectures

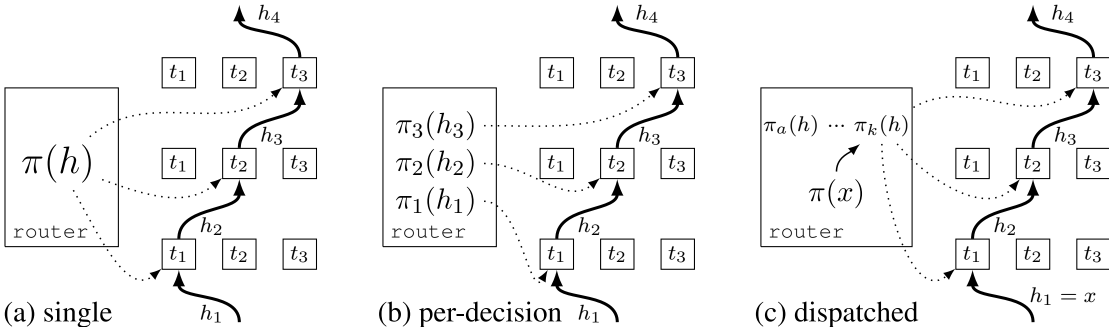


Figure 3.4: Routing architectures

Because the only general requirement for the architecture of the router is that a router takes an activation or a sample as input and outputs a single integer value interpreted as a module selection, many different architectures are possible. However, it is oftentimes impossible for the router to treat all samples and activations uniformly. Consider routing a neural network where each layer has a different output dimensionality. In this case, the router cannot itself be a single neural network, because it cannot handle the changing dimensionality. Then a router *needs* to consist of multiple different networks. We could also conceive of hierarchical routers, where a hierarchy of decisions leads to the final module selection, again implemented as multiple neural networks. We can describe these architectures by the respective input-output pairs of each of the (sub) routers. We will now discuss three out of many possible high-level design ideas, and their potential benefits.

In the conceptually simplest version used by Chang et al. [2019], Kirsch et al. [2018], Ramachandran and Le [2019], and unsuccessfully tried by Rosenbaum et al. [2017], there is only one router that learns to make all required decisions (compare Figure 3.4(a)). This router's statespace, i.e., the input to the one function approximator, contains the possible space of all activations, including the input activation. This in turn means that input and all activations need to have the same dimensionality. Additionally, it requires additional work to stabilize as it is the hardest architecture to train – curriculum learning for [Chang et al., 2019], separate grouping for [Kirsch et al., 2018] and soft-routing for [Ramachandran and Le, 2019].

We introduced two more architectures in [Rosenbaum et al., 2017] that divide the routing problem over multiple subrouters. The first is assigning one subrouter to each step in the routing path (see Figure 3.4(b)). Consequently, each subrouter has a statespace constrained to the activations possible at the respective depth. In general, this approach has the disadvantage that it only allows a recursion depth as deep as the number of subrouters defined. However, it is an effective way to implement routing architectures where constraints require the modules available at different depths to be different[2].

The second option is what we call a "dispatched" routing network (compare Figure 3.4(c)). We introduced it in [Rosenbaum et al., 2017], but only thoroughly investigated it in [Rosenbaum et al., 2019a]. This hierarchical configuration has an extra preceding subrouter with the interesting task to assign – or cluster – samples in input space first, before assigning each sample to be routed to one of a set of parallel subrouters, each of which exclusively works in activation space.

In practice, dividing the decision problem over multiple subrouters, each with only a subset of actions and states to learn, can make training the router considerably easier. In particular in early stages of training, the problem of training a policy on modules that are

[2]We assume that several other routing papers also use separate approximations to deal with constraints, though they do not mention this explicitly.

similarly untrained can make fully recursive, single subrouter routing models hard to train. For further empirical comparison, see Section 7.1.

Another routing architecture design choice is the available action space. One already-discussed option that is particularly relevant for fully recursive routing models is to include a "termination" action to stop routing and forward the last activation. This termination action is not required for per-decision router designs (Figure 3.4(b)), but could be implemented if model constraints permit. Another possible action, introduced by [Rosenbaum et al., 2017] for per-decision routing networks, is to include a "skip" action that does not terminate routing, but instead simply skips one sub-router. These two actions can result in identical behavior for limited-depth routing networks.

3.5 The Routing Search Space

For the routing optimization problem, the model needs to jointly optimize over models and module parameters. Compared to a routed model's unrouted equivalent that only optimizes the module parameters, the search space is combinatorially larger. For a maximum depth d and a number of modules at each step m, we have approximately m^d implicitly defined models,[3] and thus a search space m^d times larger.

This complicates the optimization problem tremendously, as will be elaborated in the following section. To compensate, parts of the challenges to make routing work in practice consists of finding constraints on this search space that still allow finding good solutions to a particular problem. The routing architectures presented in the previous section are some approaches to solve this problem. We will revisit this perspective after talking about the challenges routing faces in the following chapter.

[3] The correct number of models is higher, as the model may stop the recursion early. If we also consider a 'null' action to model this, we can easily derive $(m+1)^d$ as an upper boundary.

3.6 A Different Perspective

This far, we have introduced routing as jointly training modules transforming the input, and training the router that composes these modules. That is, we have introduced it as the following bi-level optimization problem (also compare Section 2.5):

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \underset{\langle x,y \rangle \in DS}{E} (\mathcal{L}(\mu_\theta(x), y)) \tag{3.13}$$

$$\text{subject to:} \quad \mu_\theta = \underset{m \in M}{\operatorname{argmin}} \mathcal{L}(m_\theta(x), y) \tag{3.14}$$

While accurate, focusing on the two nested optimization problems does not fully capture the intricacies of the interplay of these two optimization problems conceptually. We will now analyze what is required to solve these two problems jointly, deriving a perspective that focuses on several underlying learning problems. This perspective will prove very useful in explaining problems and algorithmic choices later on.

At the core of this perspective is the observation that modules are generally shared across models, and that the second optimization problem, equation 3.14, has to navigate how modules are shared. That is, beyond purely finding a good composition of modules for a given sample, it needs to make sure that compatible samples are routed through similar paths while incompatible samles are routed through dissimilar paths. Put differently, to decide on an activation-by-activation basis if different samples should share specific modules along a routing path requires the router to learn a task-specific latent distance metric. This metric will then be used to partition the samples and to determine path-overlap between the samples of different partitions.

Combining this perspective with the interpretation suggested by equations 3.13 and 3.14, we can thus identify three different conceptual learning problems that need to be solved jointly when training a routing network:

Learning-to-Transform A routing network needs to train its modules to contain useful compartmentalized functions that can be meaningfully separated and combined with other

transformations. This is complicated by 'learning-to-compose' as only stable routing paths will yield optimal solutions for the modules.

Learning-to-Compose The router of a routing network needs to meaningfully compose modules. This is complicated by learning to modularize, as the modules are not fully trained yet, introducing non-stationary dynamics into the routing problem. It is additionally complicated by 'learning-to-partition' as the router needs to find compositions that respect the relationships between different samples.

Learning-to-Partition The router needs to learn a latent distance metric that allows it to jointly

– partition the sample distribution into clusters of samples that can share entire routing paths.

– learn distances between different clusters of samples so that it can decide on which modules are shared between the paths of these clusters.

This is complicated by 'learning-to-modularize', as different module parameterizations may yield different distances between different clusters, or even form different clusters.

CHAPTER 4

CHALLENGES TO ROUTING AND OTHER FORMS OF COMPOSITIONAL COMPUTATION

There are several challenges particular to training routing networks, which should generalize to many forms of compositional learning. These include training stability, module collapse, and overfitting as well as difficulties with performance extrapolation and formalizing the setting. Instability in training may occur because of a complex dynamic interplay between the router and module training. Module collapse may occur if module selection collapses to a policy which makes the same decision for all inputs. Overfitting may be severely exacerbated in modular architectures because of their added flexibility to learn specialized modules for a very narrow subset of samples. Successfully extrapolating the performance of specific modules out over the course of training would potentially allow a more successful selection strategy, but achieving this is a difficult problem itself. Finally, there is no good formalization of popular training methodologies that unifies reinforcement learning for the module selection training with supervised training for the modules. We introduced these five challenges in [Rosenbaum et al., 2019b] first.

4.1 Underlying Causes

Training a routing network is challening as it is difficult to balance the three learning problems introduced in section 3.6: Learning-to-transform, learning-to-compose and learning-to-partition. However, one may argue that core problems of a more general nature underlie these challenges to modular architectures. In fact, we can see that their interplay is difficult because we must simultaneously balance two challenges of learning: the transfer-

interference trade-off [Riemer et al., 2019], and the exploration-exploitation dilemma [Sutton and Barto, 1998].

4.1.1 The Transfer-Interference Trade-Off

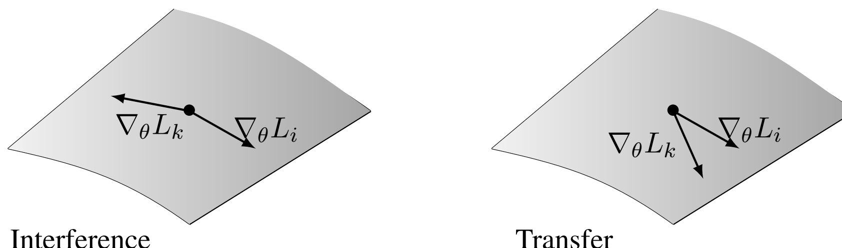


Figure 4.1: Depiction of the effects of transfer and interference across examples i and k during learning.

The "transfer-interference trade-off" refers to the problem of choosing which parameters in the model are shared across different input samples or distributions. When there is a large amount of sharing of model parameters during training, we may see better performance as a consequence of *transfer*, because each parameter is trained on more data. But it may also lead to worse performance if the training on different samples produces updates that 'cancel' one another out, or cause *interference*. This plays an important role for compositional architectures, as one of the core ideas behind compositionality is to have specialized modules that fit one but maybe not another sample.

4.1.2 The Exploration-Exploitation Dilemma

The "exploration-exploitation dilemma", while mostly associated with reinforcement learning, generally occurs in other cases of goal-driven stochastic sampling as well. It arises when a decision maker has insufficient information to determine an optimal decision, and needs to learn more about its environment first. At a given time in the learning process, the decision maker will oftentimes believe that one choice works better than the others. However, because it does not yet know the true distribution of all choices, another choice may in fact be much better. But to find it, the decision maker has to take the risk of also performing

worse than if it had stuck with the known solutions. Sticking with known solutions is known as *exploitation*, while taking a risk trying to discover new solutions is called *exploration*. Balancing these two is one of the core challenges to reinforcement learning [Sutton and Barto, 1998].

This does not mean that it may not occur in other contexts as well. In fact, it is applicable for every goal-driven sampling process. This obviously includes the previously discussed reparameterization strategies, but it also extends to top-k approaches where only k out of a much larger number of selections are chosen.

Applied to any form of modular learning, the composer (the router for a routing network) has to strike a balance between these two. Consider that after the initialization of a routing network, each sample will be associated with a given path, simply by virtue of the initial parameter values of the router. If, on the one hand, the network was to only exploit, it would take these paths as ground truth. Because these paths are not very expressive yet, many even incompatible samples will be routed the same. This will produce interference, preventing the routing network from learning good module parameters. If, on the other hand, the network was to only explore, each sample with be routed through a new random path. This means that the modules cannot benefit from any kind of transfer, as even similar samples will not reliably share parameters.

Striking the right balance between exploration and exploitation can therefore help with interference but is not sufficient to mitigate it entirely. This entanglement complicates the learning and means we cannot treat these tradeoffs in isolation.

4.2 Training Stability

Training a routing network requires learning to transform, to compose, and to partition. Considering in particular learning to partition in light of the transfer-interference trade-off means that we need to learn a latent measure of distances between different samples, forming clusters that will optimize routing paths to assign largely different paths to interfering

samples, and overlapping paths for compatible samples. This is particularly relevant during training, because failure to do so will cause catastrophic interference of training the modules.

This is complicated by the fact that, in the early learning phase when both modules and the router have just been initialized, neither routing paths nor modules have taken on any real meaning. This means that the router needs to find clusters and assign paths, while the modules need to learn good transformations. But without having meaningful modules, clustering the data will be difficult, as the latent distance metric it needs to learn depends on the parameters of the modules. This creates a “chicken-and-egg” problem: How can a routing network learn to solve one of the three underlying problems, if each depends on having learned learned to solve another?

In our experience, this problem can result in the routing dynamics never stabilizing, leaving the network at effectively random performance. This problem roughly correlates with the complexity of the decision-making problem. If the router only needs to make decisions based on a good low-dimensional projection of the samples or activations, then stability is less of a problem. If, however, the router has to consider very complex distributions in high dimensional spaces, it consistently struggles.

A mitigating strategy could be to slow down or speed up the learning behavior of either the modules or the router, by either fixing their parameters for a short initial period, or by reducing their learning rate. While all of these approaches introduce *bias* into the learning phase, their stabilizing effect can justify them. As the module learning rates tend to depend on the overall architecture, such a strategy should focus on finding the learning rate for the router. We will present an empirical analysis in chapter 7.2.2.

Another strategy that can be applied is curriculum learning [Bengio et al., 2009], as is done by [Chang et al., 2019]. This can be very effective because the router is guided towards good solutions by the ordering of the samples. However, it can only be applied if the training data can be naturally ordered by complexity, which is not the case in many

important domains. Neither of these strategies offers a general solution to the problem. They may work in some settings and fail in others.

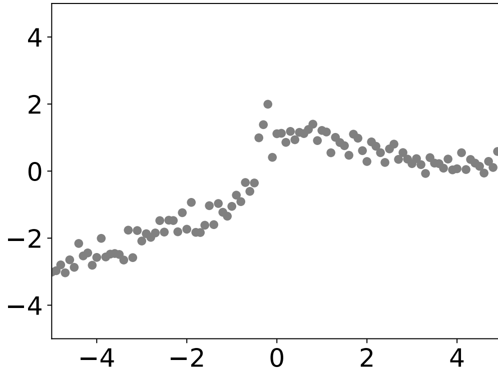
Yet another strategy is to try and solve one of the three learning problems separately, so that its dynamics are taken out of the learning process. We will introduce one such strategy in chapter 5.

4.3 Module Collapse

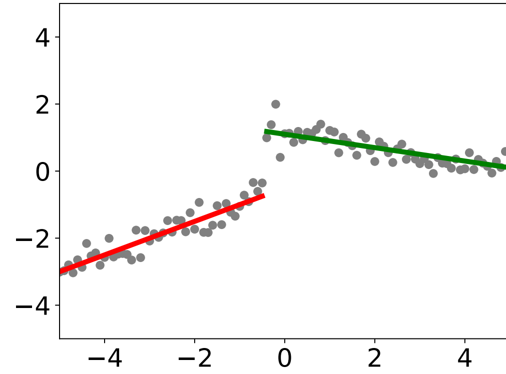
Another common training difficulty is *module collapse* [Kirsch et al., 2018]. This occurs when the router falls into a local optimum of learning-to-compose, choosing one or two modules exclusively. Module collapse occurs when the router overestimates the expected reward of a selection. This may happen due to random initialization of the modules or due to random initialization of the policy, such that one module has higher initial value. Either way, the module will be chosen more often by the router, and therefore receive more training. This in turn improves the module so that it will be selected yet more often and receive yet more training until the module is dominant and no others seem promising.

As an example, consider Figure 4.3, which depicts a routing problem of one dimensional linear regression (because it is easily illustrated). In plot (a), we depict a distribution with two noisy linear modes. Plot (b) shows the perfect, and desired, routing solution, where the routing model correctly assigns a routing path to each mode. Plot (c) depicts the regression curves produced by the available modules at initialization. Because the red and yellow approximations have been initialized producing too great an initial loss, the router will only update these during exploration phases, and will instead choose the green approximation. This may result in a suboptimal approximation (depicted in plot (d)), which can form a local optimum that will trap the router without prolonged additional, poorly rewarded exploration.

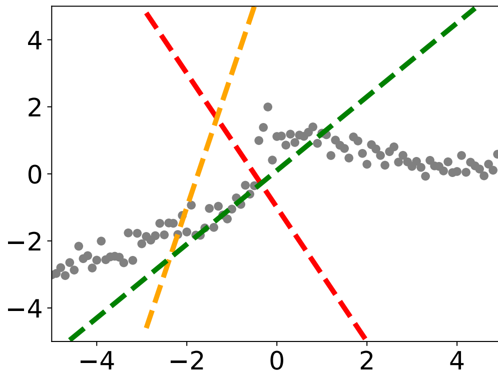
This illustrates the view of collapse as a local optimum achieved by maximizing for early transfer. When every selection is "bad", training a transformation even on generally less compatible samples, such as is shown in Figure 4.3, will improve its representative



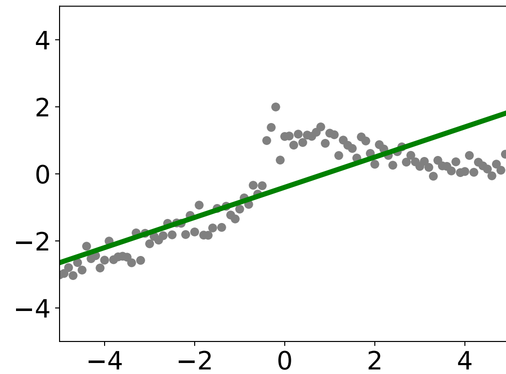
(a) A distribution with two linear modes



(b) The desired routing solution; the model captures both modes



(c) The routing modules at initialization



(d) Module Collapse: the router reaches a local optimum using the green module only

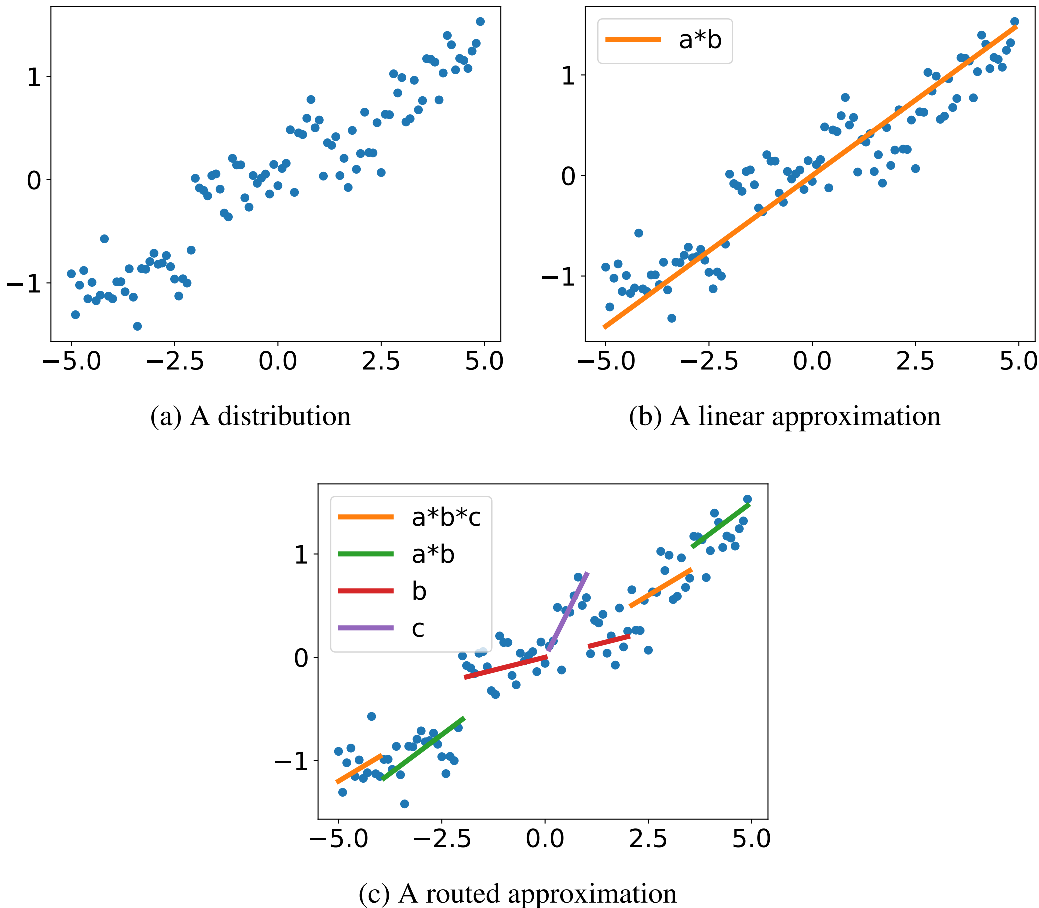
Figure 4.3: An example of how a 1-dimensional linear routing problem can collapse

power. Similarly, it can be seen as a local optimum in the exploration-exploitation dilemma, since the router correctly believes *at a given time* that one module is better than others, and may thus not explore the other options sufficiently.

Existing solutions to this problem include adding regularization rewards that incentivize diversity [Cases et al., 2019], separating the training of the router from the training of the modules with an EM-like approach that learns over mini-batches, explicitly grouping samples first [Kirsch et al., 2018], and conditioning the router entirely on discrete meta-data associated with the sample, e.g., task labels [Rosenbaum et al., 2017, Cases et al., 2019].

This has the additional benefit of reducing the dimensionality of the information on which the decisions are made, which, in turn, makes the routing problem easier.

4.4 Overfitting



(a) A distribution

(b) A linear approximation

(c) A routed approximation

Figure 4.4: Illustration of how a routed model may overfit. The learned parameter values of the three linear transformations are $a = 3, b = 0.1, c = 0.8$.

Previous work [Kirsch et al., 2018, Shazeer et al., 2017] observes that models of conditional computation can overfit more than their “base”-models, but do not examine this issue in depth. We have also encountered this problem multiple times, and believe that it stems from the flexibility of routing models to compose highly local approximations. Consider the example in Figure 4.4, where we again depict a routed linear scalar approximator.

Suppose we have a training dataset consisting of observations of a noisy linear process, such as the one in Plot (a). It can be clearly seen that a linear model gives good (and desired) approximations, as in Plot (b), and that it does not overfit to the noise present in the data. However, imagine that we wanted to learn such an approximation with a routing model that consists of three parameterized functions, $f_1 = a \cdot x, f_2 = b \cdot x, f_3 = c \cdot x$, which the router can combine to a maximum depth of three. If the router now routes purely based on input information, i.e., if the router can route each sample through a different route, then it may choose a different path for different subsets of the training data, resulting in different (local) approximations as illustrated in Plot(c). For neural architectures in particular, this implies that routing breaks an implicit smoothness assumption that arguably allows good generalization in spite of high dimensionality (see [Zhang et al., 2016] for a further discussion). Because the space of routes or paths grows exponentially with routing depth, deep routing networks can potentially learn highly local approximations with only few samples covered per approximation, or routing path.

Within the transfer-interference trade-off, overfitting is a natural consequence of avoiding training interference. In models of uniform, i.e., non-compositional, computation, the training algorithm always enforces some amount of interference over the model's parameters. While often harmful, it can also steer training away from local suboptimal solutions that cause the model to overfit.

Because this problem is not well investigated for compositional machine learning models, we can only speculate on possible solutions. The most obvious is to apply regularization of some kind that prevents the router from fitting samples to highly expressive local approximators. A very successful solution with a different set of problems is the 'routing-by-meta-information' architecture of [Rosenbaum et al., 2017, Cases et al., 2019] that we will discuss in detail in chapter 5. Another possible solution might be module diversity: if only different kinds of approximators can be fit, then the "locality" of routing does not have the same effect. This would explain why [Ramachandran and Le, 2019] were able to

achieve very high performance with a modular architecture even when learning with few examples, although we need to take their particular top-k architecture into account. Another uninvestigated area that seems promising for exploration is the tradeoff in capacity between the router and the modules. Is it better to have a 'smart' router and 'dumb' modules or vice versa? Either will probably have an effect on overfitting.

4.5 Module Diversity

While much of the prior work has focused on the case where each module is of the same kind but with different parameters, routing modules that have different architectures have the potential to be even more powerful and sample efficient. This allows for a larger coverage of functional capabilities, thereby increasing the expressive power of a routing network. Interestingly, this is a specific instance of the "Algorithm Selection Problem" [Rice, 1976]. For learning problems, however, the problem is complicated by the different learning dynamics that different architectures might exhibit.

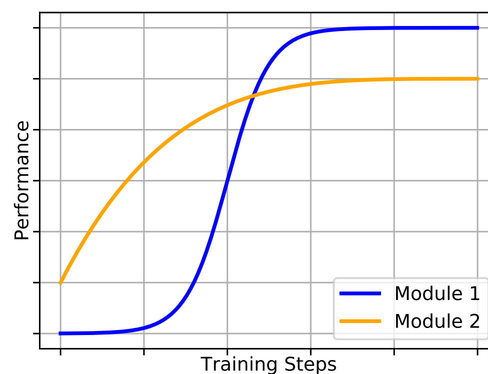


Figure 4.5: Module Learning Dynamics

Consider Figure 4.5. For any learning algorithm selection problem, deciding that Module 2 will yield superior performance requires the composing algorithm to sufficiently explore – and thus train – both selections. As for routing networks, exploration correlates with interference, this process may unfortunately degrade the overall performance.

This means that even if the router were to explore Module 1 sufficiently, the sampling noise of exploration could potentially interfere with the training procedure of selections taken before or after. While some progress on this problem has been made [Ramachandran and Le, 2019], it focuses on selections with similar learning characteristics. The full problem, for selections with arbitrary learning characteristics, was only explored in an offline setting, where selections are trained until completion and not varied while training [Zoph and Le, 2017, Liu et al., 2018]. Finding optimal selections online remains an open problem due to the difficulty of anticipating future performance of modules in their larger context.

4.6 A uniform formal framework

All of the previous challenges are complicated by the lack of a principled mathematical formalization for compositional learning. In particular, it is unclear how the training of the composition strategy relates to and interacts with the training of the modules. Although existing approaches have found very successful solutions, even without such a formalization, a principled framework may provide additional insight, convergence guarantees and directions for the development of better algorithms.

The main problem is that the mutual interference between the training of the modules and the training of the composition is badly understood. It is not clear how exactly composing functions in different ways will impact the composed modules' training. In particular when the modules are trained with gradient-based approaches, recomposing the modules should have high impact on their gradients.[1]. For the other direction, consider the case of a reinforcement learning strategy for training the routers (arguably the most relevant update strategy). Reinforcement learning relies fundamentally on the assumption that the environment is governed by an underlying Markov Decision Process (MDP) [Sutton and Barto, 1998]. A naive though intuitive characterization of routing in this context might

[1]For an empirical analysis of how compositionality may interfere with the training of the modules, compare section 7.4.4

interpret individual modules as actions. But an MDP requires a static, non-changing set of actions and here the modules are themselves trained and change over time. Although in practice this strategy has been shown to work well [Rosenbaum et al., 2017, Chang et al., 2019, Cases et al., 2019], it lacks theoretical justification, and more principled approaches may yield superior performance.

This critique extends to the two special cases of MDPs introduced in the literature: Meta-MDPs and stochastic games. Meta-MDPs were introduced by [Hay et al., 2014] and applied to routing by [Chang et al., 2019]. They specifically model the computational steps required to make a decision in another underlying MDP. That is, their states are partial results, and their actions are computations. Once they terminate, the resulting computation produces an action in the underlying decision problem. This approach emphasizes that the routing problem consists of computation steps, and not steps taken in an environment.

A stochastic game, related to routing by myself in [Rosenbaum et al., 2017], is the multi-agent extension to MDPs and allows for an arbitrary number of agents to exist, which collaborate or compete in the same environment. As other agents interact with the environment, the environment becomes non-stationary from the perspective of any single agent. While this approach consequently models the non-stationarity of a routing problem better than a Meta-MDP, it still does not solve the more fundamental problem of possible interference between update strategies. We will discuss this approach in greater detail in section 5.

There are two scenarios that have more principled solutions to this criticism, although they, too, may benefit from a more targeted solution as they suffer from many of the same problems in practice. In the first, for neural networks, we use some form of reparameterized sampling function to update the composition (compare section 3.2). This addresses the core of the problem because the reparameterized decision-making algorithms explicitly use the same update strategy as the (neural) models. In the second scenario, the problem solved with a routing network is optimizing a policy for a given MDP. In this case, the routing network

is the policy, and can consequently be modeled as a coagent network [Thomas and Barto, 2011]. A coagent network is a theoretical framework that, like Meta-MDPs, models the decision-making in an underlying MDP but explicitly allows for non-stationary interactions between parts of the policy. These parts, called coagents, can update locally on stochastic information. For routing, we can model the policy solving the underlying MDP as a routing network with routing coagents and transformation coagents (i.e., the module parameters are part of the policy). Then the policy gradient theorems in [Thomas, 2011] for the acyclic and [Kostas et al., 2019] for the cyclic case hold, because the modules will also be updated with the same policy gradient updates. Extending this work on coagent networks to also cover non-reinforcement learning problems such as supervised learning as well might offer a principled way of modeling routing and similar RL-integrated approaches for conditional computation.

4.7 The Challenges and the Routing Search Space

As already mentioned at several times, routing networks form complex optimization problems. Partially, this is because they combinatorially expand the search space beyond the basic problem of finding optimal module parameters. Partially, it is because the space they form is highly non-linear. In the following, we try to describe the challenges outlined in this chapter as properties of this search space.

Instability is not directly related to the size of the search space, as even in large state spaces training can be stable. However, instability is caused by the search space's non-linearity, because even small changes to the router's policy can greatly 'confuse' the training algorithm of the modules.

Collapse is a local optimum in the routing solution space, in particular for the parameters describing the policy.

Overfitting is an immediate consequence of the size of the search space. As discussed before, the theoretically worst case is if the router were to assign a single sample to each routing path. As such, it is exacerbated by the possible number of routing paths.

Module Diversity cast as a property of the search space has to do with how specific regions of the function space have different higher level properties, such as curvature.

4.8 The Challenges and Local Approximations: The Exacerbated Bias-Variance Trade-Off for Modular Learning

From the perspective of the routing search space, much of the work in this thesis deals with regularization and constraint techniques that limit the search space or that penalize specific regions of it.

Especially when comparing the two challenges of collapse and overfitting, it becomes clear that they are not unrelated, but are an effect of too little – or too much – 'flexibility' on the router's part, where flexibility is a routing network's ability to localize its routing decisions. If the router is not flexible enough to realize how truly different modes in a distribution should be treated differently, it will cause underfitting, oftentimes through collapse. If, on the other hand, the router is too flexible in mapping different inputs to different routing paths, it creates hyperlocal approximations that can overfit badly.

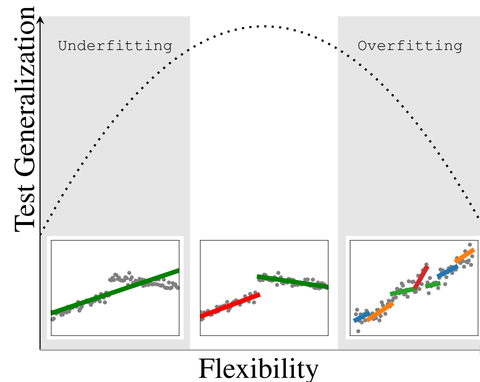


Figure 4.6: The flexibility dilemma. Flexibility is the ability of a routing model to adapt to localized inputs. Low flexibility means few large clusters, and low flexibility means many small clusters. The extrema are only one cluster, i.e., collapsing and underfitting, and one cluster for each sample, i.e., overfitting.

This problem of 'flexibility' is known as the 'Bias-Variance Trade-Off' in machine learning, which is exacerbated by compositional architectures. At its core, the effects of this trade-off become much worse for routing because routing has an enormous degree of expressivity. Instead of the (possibly already high) original expressivity of an unrouted model, routing defines a combinatorial number of implicit local models, each of which with the same expressivity as the original model. If the model finds a good 'locality' of approximations, routing can allow a model to achieve truly impressive levels of generalization (compare [Chang et al., 2019], and in particular the section on nested implicatives in [Cases et al., 2019]), as it can adapt to unseen samples by mapping these to known solutions in a highly variable way. However, on both sides of this 'locality' of approximation lie the above mentioned pitfalls of collapse and overfitting.

While we illustrated and explained this problem on routing (neural models), it is by no means limited to it. Any modular approach that can treat different samples differently will be exposed to this dilemma between under- and overfitting, as it comes with any form of local approximation approach.

CHAPTER 5

ROUTING WITH META-INFORMATION

As discussed earlier in section 3.6, successfully training a routing network requires solving three different problems: Learning-to-transform, learning-to-compose, and learning-to-partition. Only if all three problems are balanced during training can a routing network learn successfully, because each problem interacts with the other two.

This illustrates again how routing generalizes several existing approaches in the literature. Learning-to-compose approaches (for a recent example, consider [Alet et al., 2018]) fix the modules, and learn to compose the network for specific samples. Apart from the obvious learning-to-compose, this also requires the model to learn to partition. This is equally true for fixed path approaches that solve the problem of learning-to-transform ([Andreas et al., 2016] could be considered such an approach). Another, previously uninvestigated alternative is to solve the last of the three problems, learning-to-partition, separately. But of the two components behind learning-to-partition, clustering the sample set and learning the path-distances between these clusters, learning the path-distances depends on the routing paths and the modules and can therefore not be separated. That leaves us with the clustering part of learning-to-compose, and the following question: Can we find externally defined clusters that we can use to simplify and stabilize the routing problem? This chapter tries to answer this question by stipulating that appropriate clusters can be formed by meta-information.

Meta-information generally refers to additional low-dimensional information that is provided along with the samples of a dataset. For many applications, it is collected by default. Scientific measurements generally collect time, place, and something like an instrument

identifier. Another more common example is photography, because most cameras store time, place, camera model, aperture size and similar as meta-information.

While many forms of meta-information do not form any kind of useful distance metric, those that form meaningful semantic abstractions do. These can come in many forms. Some are semantically valuable labels like partial descriptions of content. Others, particularly relevant to this chapter, are task-labels. Task labels occur in the context of multi-task learning, when a model is trained to jointly solve several related, but distinct problems [Caruana, 1997]. Such approaches are motivated by transfer, boosting overall performance by having one task benefit from training on another. Consequently, solving a multi-task learning problem means navigating the transfer-interference trade-off.

5.1 Multi-Task Learning and Multi-Agent Routing

Given a multi-task learning problem, we can define a routing network where we stipulate that all samples from a particular task should be routed the same way, i.e., where a task defines a cluster for the learning-to-partition problem. This allows us to greatly simplify the routing problem, because we can define a router that does not consider the actual content of the sample, or of a later activation, but that routes purely based on task-label. Such a router architecture splits the router into several independent sub-routers. From the decision making perspective, this means that the router is constituted by multiple distinct agents, one responsible for each task.

A router decision policy using multiple agents can be seen as a multi-agent approach to routing. Even though we criticize this perspective in 4.6, it is a useful perspective to explore. It turns the routing problem into a stochastic game, which is a multi-agent extension of an MDP. In stochastic games, multiple agents interact in the environment and the expected return for any given policy may change without any action on that agent's part. In this view, different agents with compatible tasks can gain from transfer by training the same modules, while agents solving incompatible tasks can avoid interference by training different modules.

Similar to the relationship between MDPs and reinforcement learning, solving a stochastic game from the perspective of an individual agent is addressed by multi-agent reinforcement learning (MARL).

Algorithm 5.1: Weighted Policy Learner

---

input : A trajectory $T = (S, A, R, r_{final})$
n the maximum depth;
$\hat{\mathcal{R}}$, the historical average returns (initialized to 0 at the start of training);
γ the discount factor ; and
λ_π the policy learning rate

output : An updated router policy π

1 **for** *each action* $a_i \in A$ **do**
2   Compute the return:
3     $\mathcal{R}_i \leftarrow r_{final} + \sum_{j=i}^{n} \gamma^{j-i} r_j$
4   Update the average return:
5     $\hat{\mathcal{R}}_i \leftarrow (1 - \lambda_\pi)\hat{\mathcal{R}}_i + \lambda_\pi \mathcal{R}_i$
6   Compute the gradient:
7     $\Delta(a_i) \leftarrow \mathcal{R}_i - \hat{\mathcal{R}}_i$
8   Update the policy:
9   **if** $\Delta(a_i) < 0$ **then**
10         $\Delta(a_i) \leftarrow \Delta(a_i)(1 - P(a_i|\pi))$
11   **else**
12         $\Delta(a_i) \leftarrow \Delta(a_i)P(a_i|\pi)$
13     $\pi \leftarrow$ simplex-projection$(\pi + \lambda_\pi \, \mathbb{1}_{|A|}(a_i) \, \Delta(a_i))$

One MARL algorithm specifically designed to address this problem is the weighted policy learner (WPL) algorithm [Abdallah and Lesser, 2006], shown in Algorithm 5.1. WPL is a non-standard[1] actor-critic policy gradient algorithm designed to dampen oscillation and push the agents to converge more quickly to a possibly temporary optimum. This is done by scaling the gradient of the expected return for an action a according the probability of taking that action $\pi(a)$ (if the gradient is positive) or $1 - \pi(a)$ (if the gradient is negative). Intuitively, this has the effect of slowing down the policy updates when the policy is moving away from a temporary equilibrium strategy and increasing it when it approaches one.

[1]WPL is non-standard in that it does not utilize the log-probability trick.

The full WPL algorithm is shown in Algorithm 5.1. It is assumed that the critic in form of a historical average return $\hat{\mathcal{R}}_i$ for each action a_i is initialized to 0 before the start of training. The function simplex-projection projects the updated policy values to make it a valid probability distribution. The projection is defined as: $clip(\pi)/\sum(clip(\pi))$, where $clip(x) = \max(0, min(1, x))$. In its original form, WPL does not rely on state information, and has only a tabular representation of its policy. Consequently, the algorithm can only be used for fixed-depth routing, because each decision requires one separately stored action distribution. Details, including convergence proofs and more examples giving the intuition behind the algorithm, can be found in [Abdallah and Lesser, 2006]. We extend this algorithm to work with function approximators in section 7.1.1.

5.2 Experiments on Image Datasets

In this section, we will discuss routing results on several image domains. Additionally, we will explicitly consider the multi-task learning scenario, in which every sample at train and at test time has meta-information determining its task. This section was published as [Rosenbaum et al., 2017].

5.2.1 The Datasets

The multi-task datasets experimented on are: Multi-task versions of MNIST (MNIST-MTL) [Lecun et al., 1998], Mini-Imagenet (MIN-MTL) [Vinyals et al., 2016] as introduced by Ravi and Larochelle [2017], and CIFAR-100 (CIFAR-MTL) [Krizhevsky, 2009] where we treat the 20 superclasses as tasks. In the binary MNIST-MTL dataset, the task is to differentiate instances of a given class c from non-instances. We create ten tasks and for each we use 1k instances of the positive class c and 1k each of the remaining nine negative classes for a total of 10k instances per task during training, which we then test on 200 samples per task (2k samples in total). MIN-MTL is a smaller version of ImageNet [Deng et al., 2009] that is easier to train in reasonable time periods. For mini-ImageNet, we randomly choose

50 labels and create tasks from 10 disjoint random subsets of five labels each chosen from these. Each label has 800 training instances and 50 testing instances – so 4k training and 250 testing instances per task. For all ten tasks, we have a total of 40k training instances. Finally, CIFAR-100 has coarse and fine labels for its instances. We follow existing work [Krizhevsky, 2009] creating one task for each of the 20 coarse labels and include 500 instances for each of the corresponding fine labels. There are 20 tasks with a total of 2.5k instances per task; 2.5k for training and 500 for testing. All results are reported on the test set and are averaged over three runs.

Each of these datasets has interesting characteristics that challenge the learning in different ways. CIFAR-MTL is a “natural" dataset whose tasks correspond to human categories. MIN-MTL is randomly generated, so it will have less task coherence. This makes positive transfer more difficult to achieve and negative transfer more of a problem. MNIST-MTL, while simple, has the difficult property that the same instance can appear with different labels in different tasks, causing interference. For example, in the “0 vs other digits" task, “0" appears with a positive label but in the “1 vs other digits" task it appears with a negative label.

5.2.2 The Architecture

The experiments are conducted on a simple convnet architecture that appeared recently in [Ravi and Larochelle, 2017]. This model has four convolutional layers, each consisting of a 3x3 convolution and 32 filters, followed by batch normalization and a ReLU. The convolutional layers are followed by three fully connected layers, with 128 hidden units each. We will call this architecture ‘Base’ throughout this section. The routed version of the network routes the three fully connected layers and for each routed layer there is one randomly initialized module per task in the dataset (for e.g., CIFAR-MTL this makes three layers with 20 modules each). The preceding convolutional layers are shared in between all tasks unless otherwise noted.

For multi-task routing, we can define routers that do not consider the sample or the current activation, because the clusters the network needs to learn in single task learning are externally defined by the task-labels. In these cases, we implement the policies as tables that can be indexed by the task labels and the routing depth. This means that, unless a special 'pass' action is available, the router will make exactly three decisions, one for each routed fully connected layer.

In cases where we want the router to decide its paths based on the sample or later activations, we implement the agent's policy as one or many neural network function approximators. In these cases, the approximators are always two layer MLPs with a hidden dimension of 64. While tabular routers have a separate table per decision depth, approximation routers do not. To also give the router access to the depth in these cases, the latest activation is augmented by concatenating a 1-hot representation of the depth. For single-agent routers, the task is equally concatenated as a 1-hot representation. That is, the state for the router would be encoded as concat(h, one_hot(m), one_hot(d)), with h being the current activation, m the task index, and d the depth. For one experiment, we also implement a recurrent router that has access to a special 'pass' action. For this architecture, the modules are accessed recurrently, i.e., the same set of modules is accessed at each decision step of the routing.

5.2.3 Results

We began the experiments with a parameter sweep over the collaboration reward ρ, the learning rate, and the optimization algorithm. For a more general comparison and analysis of the impact of these hyperparameters, see section 7.2. The parameter sweep performed here yielded a collaboration reward $\rho = 0.0$ (no collaboration reward) for CIFAR-MTL and MIN-MTL and $\rho = 0.3$ for MNIST-MTL, a learning rate of 10^{-2} which is annealed by 10 every 20 epochs, and SGD as the optimization algorithm. We additionally compared several reinforcement learning algorithms, REINFORCE, Q-Learning and WPL,

but no reparameterization algorithms, as we only became aware of these later. The best-performing decision-making algorithm out of the ones tested here was consistently WPL. The experiments shown here are thus using WPL. However, because WPL was originally defined in tabular form only, approximation policies are trained with REINFORCE. Only for later work (see section 7.1.1), did we extend WPL to work with function approximators.

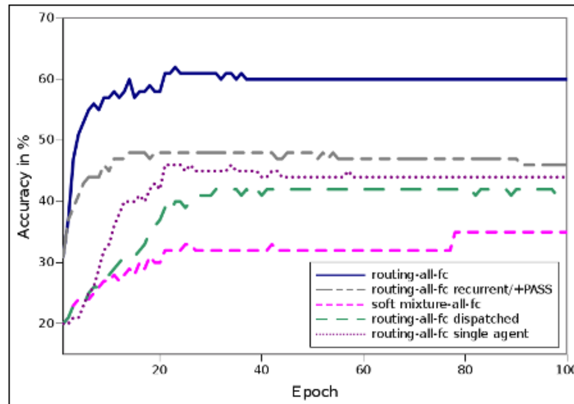


Figure 5.1: Comparison of Routing Architectures on CIFAR-MTL.

The first experiment compares different routing architectures. All of these are variations of the convolutional architecture introduced above (four convolutional layers, three fully connected layer). The architectures compared are: *routing-all-fc*, which routes all of the fully connected layer (including the final output layer), and which has a separate tabular policy for each task. All of the following architectures also route the fully connected layers only. The *routing-all-fc recurrent+Pass* architecture also has a separate policy for each task, but recurs through the same modules. It additionally allows the possibility that layers can be skipped. The *routing-all-fc single agent* architecture implements the router as a single function approximated policy that routes by current activation, but that still needs to make three decisions for the three routed layers. These layers are not recurred through. It is trained with REINFORCE. *routing-all-fc dispatched* is a dispatched architecture as introduced in section 3.4.2. Here, the router is implemented by having a hierarchy between a neural network policy that assigns samples to subrouters, which in turn are tabular. All decisions

are trained with REINFORCE. To compare with a 'soft' routing version, *soft mixture-all-fc* is a sofmax-gated version of the same architecture that computes a softmax-weighted sum over the output of all modules at each layer.

Among these different architectures, a clear hierarchy can be seen in the results shown in figure 5.1. The worst architecture is the soft mixture model that is outperformed by all routing approaches. For the four routing approaches compared, the pure multi-task, tabular multi-agent approach outperforms all others.

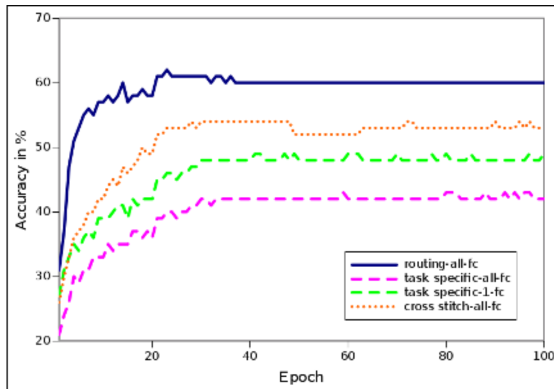


Figure 5.2: Results on domain CIFAR-MTL

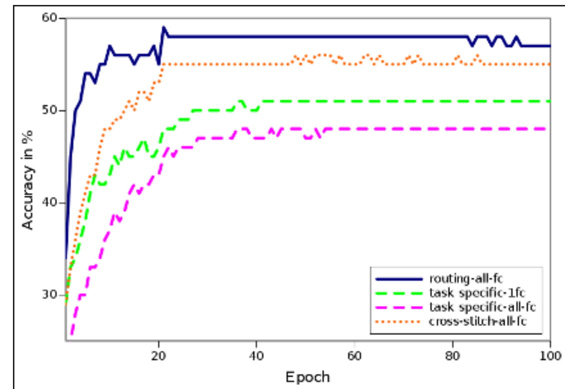


Figure 5.3: Results on domain MIN-MTL (mini ImageNet)

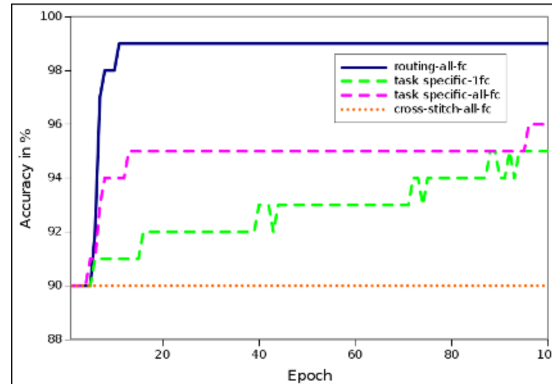


Figure 5.4: Results on domain MNIST-MTL

The next experiment compares *routing-all-fc* as the best performing routing architecture on different domains against the cross-stitch networks by Misra et al. [2016] and two challenging baselines: *task specific-1-fc* and *task specific-all-fc*, described below.

Cross-stitch networks are a kind of linear-combination model for multi-task learning. They maintain one model per task with a shared input layer, and “cross stitch" connection layers, which allow sharing between tasks. Instead of selecting a single module in the next layer to route to, a cross-stitch network routes to all the modules simultaneously, with the input for a module i in layer l given by a linear combination of the activations computed by all the modules of layer $l-1$. That is: $\text{input}_{li} = \sum_{j=1}^{k} w_{ij}^{l} v_{l-1,j}$, for learned weights w_{ij}^{l} and layer $l-1$ activations $v_{l-1,j}$. For our experiments, we add a cross-stitch layer to each of the routed layers of our base model.

The *task-specific-1-fc* baseline has a separate last fully connected layer for each task and shares the rest of the layers for all tasks. The *task specific-all-fc* baseline has a separate set of all the fully connected layers for each task. These baseline architectures allow considerable sharing of parameters but also grant the network private parameters for each task to avoid interference. However, unlike routing networks, the choice of which parameters are shared for which tasks, and which parameters are task-private is made statically in the architecture, independent of task. Architectures of this kind were made popular in Caruana [1997].

The results are shown in Figures 5.2, 5.3, and 5.4. In each case, the routing net *routing-all-fc* performs consistently better than the cross-stitch networks and the baselines. On CIFAR-MTL, the routing net beats cross-stitch networks by 7% and the next closest baseline *task-specific-1-fc* by 11%. On MIN-MTL, the routing network beats cross-stitch networks by about 2% and the nearest baseline *task-specific-1-fc* by about 6%. The results on CIFAR-MTL may be better because the task instances have more in common whereas the MIN-MTL tasks are randomly constructed, making sharing less profitable. On MNIST-MTL, the random baseline is 90%. We experimented with several learning rates but were unable to get cross-stitch networks to train well, because we did not want to provide it with the offline pretraining suggested in [Misra et al., 2016]. While such additional pretraining would help cross stitch networks, it would precondition the modules and would not test their ability to navigate the transfer-interference trade-off from scratch. Without such pretraining, routing

networks beat cross-stitch networks by 9% and the nearest baseline (*task-specific-all-fc*) by 3%. The soft version we evaluated before also had trouble learning on this dataset, furthering the point that ‘soft’ versions are not truly able to navigate the transfer-interference trade-off.

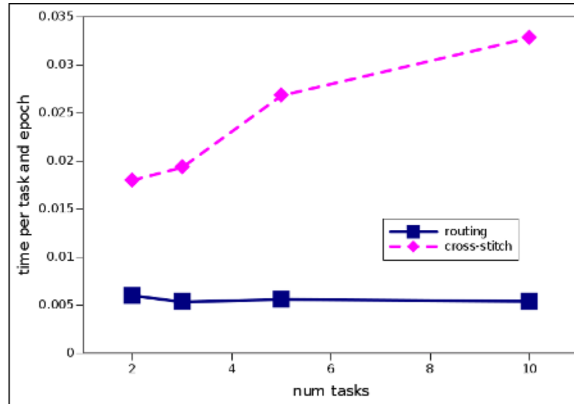


Figure 5.5: Comparison of per-task training cost for cross-stitch and routing networks. We add a module per task and normalize the training time per epoch by dividing by the number of tasks to isolate the effect of adding modules on computation.

Apart from achieving higher performance, training a routing network is much faster than training a soft model. On CIFAR-MTL, for example, training time on a stable compute cluster was reduced from roughly 38 hours to 5.6, an 85% improvement. A set of scaling experiments to compare the training computation of routing networks and cross-stitch networks trained with 2, 3, 5, and 10 modules was meant to verify this. The results are shown in Figure 5.5. Routing networks consistently perform better than cross-stitch networks and the baselines across all these problems. Adding modules has no apparent effect on the computation involved in training routing networks on a dataset of a given size. On the other hand, cross-stitch networks have a soft routing policy that scales computation linearly with the number of modules. Because the soft policy backpropagates through all modules and the hard routing policy only backpropagates through the selected block, the hard policy scales much more easily to many task-learning scenarios that require diverse types of modules.

To explore why the multi-agent approach seems to do better than the single-agent, another experiment compares their policy dynamics for several CIFAR-MTL examples.

For these experiments, $\rho = 0.0$, so there is no collaboration reward that might encourage less diversity in the agent choices. In the cases we examined, we found that the single agent collapsed and often chose just one or two modules at each depth, and then routed all tasks to those. We discussed this behavior as a challenge to compositional computation in section 4.3.

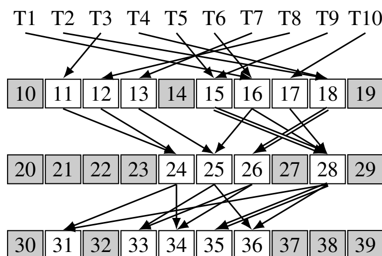


Figure 5.6: An actual routing map for MNIST-MTL.

Routing networks, when conditioned on different tasks, learn a policy which, unlike the baseline static models, partitions the network quite differently for each task, and also achieves considerable diversity in its choices as can be seen in Figure 5.6. This figure shows the routing decisions made over the whole MNIST MTL dataset. Each task is labeled at the top and the decisions for each of the three routed layers are shown below. We believe that because routing networks have separate policies for each task, they are less sensitive to a bias for one or two modules and each agent learns independently what works for its assigned task.

5.3 Experiments on Language Datasets

In this section, we will describe several different routing architectures for natural language inference (NLI). For NLI, a classification task, a machine learning model needs to decide if a given premise implies the truth of a given hypothesis. An example is "Boston is the capital of Massachusetts." implies "Boston is in Massachusetts.". The other two classification outcomes are contradiction ("Boston is the capital of Massachusetts." contradicts "Boston is in Rhode Island.") and permits ("Boston is the capital of Massachusetts."

permits "Amherst is a town in Massachusetts.") We will focus on two different NLI datasets, MULTINLI and SCI, because both of them have additional meta-information associated with each sample. We will treat the meta-information, as before, as task labels. Because neither of these is considered to be explicitly multi-task, this allows us an analysis of the usefulness of different kinds of meta-information as hard clusters. This section was published as part of Cases et al. [2019].

MultiNLI The MULTINLI corpus Williams et al. [2018] contains 392k training examples, 10K dev examples, and 10K test examples. The examples come from five genres: fiction, government reports, the *Slate* website, the Switchboard corpus [Godfrey and Holliman, 1997], and Berlitz travel guides. We treated these genre labels as meta-information for the model.

Stanford Corpus of Implicatives The premise of routing with meta-information is that routing benefits from clusters that capture some essential property of the sample distribution, i.e., that form meaningful semantic abstractions. To verify, we did experiments with the Stanford corpus of implicatives (SCI)[2], because each sample is annotated with high-quality semantic information in the form of *signatures*. Signatures, discovered by Karttunen [1971], are implicative constructions that describe logical properties of specific verbs. Examples are *manage to* or *waste chance to* (e.g., *They wasted their chance to win*). Signatures characterize the inferences these verbs support in positive and negative contexts. This makes them particularly informative for NLI predictions, thereby providing a semantically meaningful partition for the router.

For instance, the positive sentence *Joan managed to solve the problem* entails *Joan solved the problem*, and the negative sentence *Joan didn't manage to solve the problem* contradicts *Joan solved the problem*. In contrast, *waste chance* has the opposite signature,

[2] `https://nlp.stanford.edu/projects/sci/`

the order compelled him to appear as a witness	**entails**	he appeared as a witness
we have missed an opportunity to examine the art market today	**contradicts**	we have examined the art market today
Mr Odinga had not been forced to change his plans	**permits**	Mr Odinga had changed his plans

Table 5.1: Examples from SCI randomly chosen from the validation set. Each row contains a triplet formed by a premise (left column), a hypothesis (right column), and a label specifying one of the three possible relations (*entails*, *contradicts*, *permits*) holding between premise and hypothesis. The last row contains an example of a probabilistic implicative (see the main text).

since *they wasted the chance to befriend him* contradicts *they befriended him*, and *they didn't waste the chance to befriend him* entails *they befriended him*. See table 5.1 for examples and signatures for all three possible outcomes.

For evaluating routing, another very important property of signatures is that they are compositional: when two or more implicative constructions are composed in a sentence, they create a *nested implicative construction* [Nairn et al., 2006]. For example, *John managed to remember to get the keys* entails that *John got the keys*. Therefore, signatures make implicatives ideal for evaluating different degrees of compositional generalization, because they provide semantically rich meta-information. For routing, this means that a routing network should be able to pick up structures that allow it to generalize properties from several simple signatures to nested signatures, by composing existing modules in new ways.

The SCI dataset contains ≈10K premise–hypothesis pairs. Seven signature types are represented, in addition to eight stochastic signatures, i.e., verbs with a non-constant signature.

Additionally, SCI is provided in four versions. In the *joint* version, the underlying distribution of implicatives is shared across train, validation, and test splits. In the *disjoint* version, a different subset of implicatives is used in train from those used in validation and test. This allows generalization to unseen constructions to be tested. The lexical items that make up implicative constructions overlap between the splits. For example, *take vow* appears only in training and validation, while *make vow* only appears in test. Another version is *mismatch*, where different subsets of the signatures are present in training/validation and test. The last version is *nested*, where implicatives are composed of several simpler implicatives. Apart from evaluating how routing can generalize, this also makes the four

variants differently difficult to solve. From easiest to hardest (measured by standard RNN architecture performance), we have joint, disjoint, mismatch, and nested.

5.3.1 The Architecture

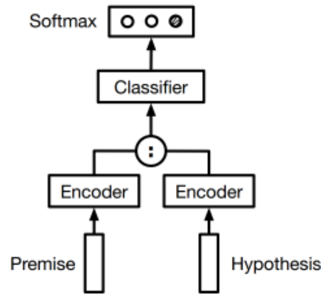


Figure 5.7: The general NLI architecture.

The high-level architecture is a simple comparison model as depicted in Figure 5.7. Both the premise and the hypothesis are encoded, the encoding is concatenated, and a classifier consisting of several fully connected layers outputs a three-way softmax that is interpreted as *entails, permits, contradicts*.

If not otherwise mentioned, we assume that the routed modules are all fully connected layers with the same input and output dimensions. We furthermore assume that the router is always selecting from the same set of functions when defining paths through the routing layers, because this straightforwardly allows recursive routing without any constraints. That is, the router is implemented with a separate stop action $\perp$ when it decides that the current activation is sufficient for prediction, but it can otherwise recur through the same set of modules. However, we limited the number of allowed recursion steps, because we found that this stabilizes training immensely. Unless otherwise noted, the implementation of the router is a tabular per-step per-task multi-agent subrouter as introduced in section 3.4.2.

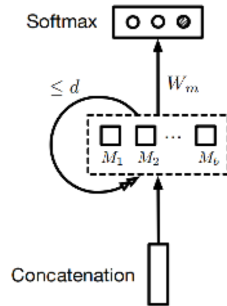


Figure 5.8: Routing the classifier.

5.3.1.1 Routing Classifiers

The simplest method we explored involves routing the layers of the classifier component of the architecture illustrated in Figure 5.7. To do this, we defined a fixed set of b fully connected layers (FC), each of the same dimensionality, and we allow the router to choose any path through this set up to a maximum length d (Figure 5.8). The final activation produced by the chosen path is then densely connected to a non-routed output layer. The main differences to the architecture presented in the previous section for image classification are that the routing is fully recursive and that the final layer is shared.

5.3.1.2 Routing RNN Encoders

In routing RNNs, we focused on their two core transformations: from the hidden state at time step t_k to time step t_{k+1}, and from the input to the hidden state. We have designed routing architectures for both of them. While we only considered LSTMs Hochreiter and Schmidhuber [1997], the techniques applied will be straightforward to adapt to other cell types.

Figure 5.9 shows an architecture where we route the input-to-hidden (I2H) transformation. Similarly, Figure 5.10 shows an architecture where we route the hidden-to-hidden (H2H) transformation. While these transformations are often designed as single fully connected layers, we allowed a recursive application of a maximum of d steps from a selection of b modules to be applied. The selections for the transformations f, i, o, c are tied to be the

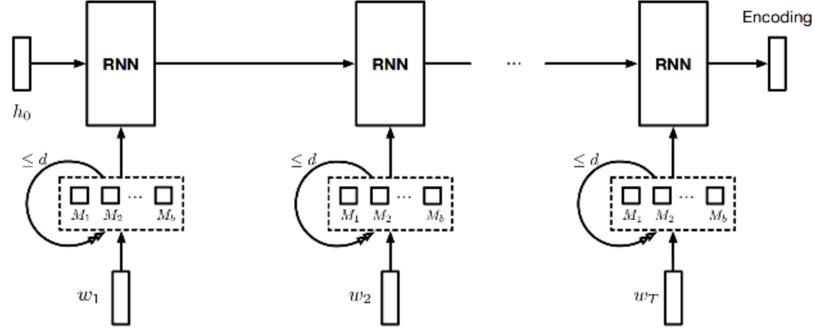


Figure 5.9: Routing the input to hidden layer of an RNN.

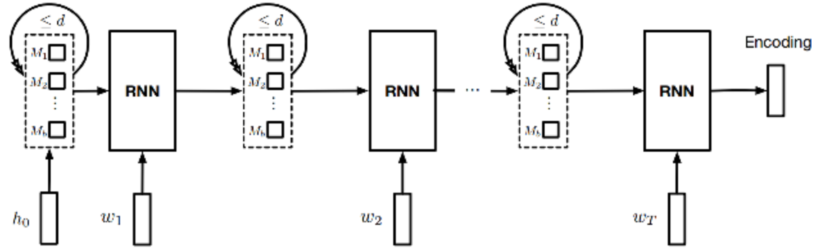


Figure 5.10: Routing the hidden to hidden layer of an RNN.

same. That is, the router makes only one decision per step, but all transformations f, i, o, c (for forget, input, output, and cell state, the core components and transformations of an LSTM) are chosen by that one decision. The corresponding transformations (in form of their weight matrices $W_{f/i/o/c}$ for I2H and $U_{f/i/o/c}$ for H2H) become:

$$K = \text{FC}_k \circ \cdots \circ \text{FC}_m(x_t), \forall K \in \{W_f, W_i, W_o, W_c, U_f, U_i, U_o, U_c\} \tag{5.1}$$

5.3.1.3 Routing CBOW Encoders

We also experiment with routing a continuous bag of words (CBOW) encoding to contextualize word interpretations. To do so, we add a word-level transformation before the main addition (Figure 5.11). This transformation can again be routed recursively (with up to d steps through b modules). As the I2H RNN model, this approach allows the task-specific learning of word encodings. The entire CBOW model can be defined as (with $w_1, \ldots, w_t$ as the sequence of words in the premise or hypothesis):

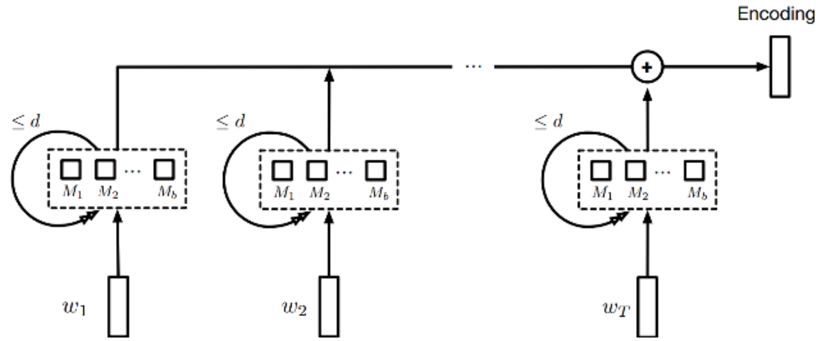


Figure 5.11: Routing a CBOW encoder.

$$\text{CBOW}_R(w_1, \ldots, w_t) = \sum_{i=1}^{t} \text{FC}_{k_1} \circ \cdots \circ \text{FC}_{k_d}(w_i) \tag{5.2}$$

5.3.1.4 Routing Transformers

The Transformers model [Vaswani et al., 2017] was recently pre-trained as a language model [Radford et al., 2018], achieving impressive results on several NLI datasets. Because the corpus for pretraining is not available, we instead used the parameter-files distributed by the authors.[3] The encoding uses twelve Transformer-blocks. Each block consists of a convolutional attention layer followed by two convolutional layers (along with several dropout and layer-norm layers). This allows routing at different levels of granularity, of which we investigated two: routing entire blocks and routing the attention step within each block. Because we use pre-defined parameters, we cannot apply the blocks or the attention layers recursively. Furthermore, we have to add routing in the fine-tuning phase of using Transformers by creating b copies of each routed module (the depth is necessarily $d = 12$, as determined by the pretrained models, making the model non-recursive). When fine-tuning, the router then diversifies the initially identical modules. To add small initial diversification of the otherwise identical modules, we experimented with adding noise (with $\sigma^2 < 10^{-7}$) to the parameters of each module.

[3] `https://github.com/openai/finetune-transformer-lm`

5.3.2 Results

We evaluated the routing architectures introduced in the previous section, along with different non routed baselines. For all models (except Transformers), we adopt the architecture in Figure 5.7, in which the premise and hypothesis are processed separately, and the final representations of each are concatenated and fed into the classifier layers. We utilized pretrained word representations (GloVe; [Pennington et al., 2014]). For all non-routed models, we provided explicit access to the genre label via special keywords put in front of each sentence before encoding (i.e., the first word of each sentence becomes 'k', where k is the task index). The routed models used this label as meta-information, creating a separate sub-router for each signature-class.

Unless otherwise specified, we used embedding and hidden dimensions of 300. The classifier consists of three fully connected layers with input and output dimensions of 600 (also when routed). The final layer projects from 600 to the output dimension, 3. All nonlinearities are ReLUs. After a parameter sweep in a range ten times larger and smaller, we found that a learning rate of $1e{-3}$ yields the best results. For Transformers, all hyperparameters are determined by the published parameter files. The routing width $b = 15$, such that there is possibly one module per signature. We also experimented with different maximal depths, but found that the router rarely composed more than three modules for a specific signature. As higher signatures additionally increased training complexity, we set the depth $d = 3$.

In experiments with different decision-making algorithms, we found that q-learning was consistently as good as other more complex decision-making algorithms algorithms, so we report only results for q-learning. As before, a complete comparison of different decision making algorithms can be found in section 7.1. Because Routing Networks have more parameters than their non-routed counterparts, we ran experiments with larger non-routed networks. We found that this did not affect performance, only resulting in more overfitting.

Encoding	Routing	MULTINLI match	SCI joint	SCI disjoint	SCI mismatch	SCI nested*
CBOW	None	61.94±0.13	57.26±0.18	55.68±0.47	53.41±0.86	50.98±0.92
	Classifier	51.99±0.17	62.33±0.80	61.17±0.28	51.55±1.61	
	WP	64.38±0.24	*74.95*±0.64	*73.91*±0.54	68.69±0.93	
	WP+D	**65.84**±0.12	**75.56**±0.77	**74.87**±0.49	**71.08**±0.52	**75.43**±0.29
RNN	None	**66.53**±0.21	67.04±2.36	63.52±2.00	*59.32*±2.87	57.69±2.75
	Classifier	65.60±0.33	**71.02**±0.85	**67.82**±0.81	**60.58**±1.35	
	I2H	47.79±0.47	53.25±0.28	56.57±1.83	53.99±0.55	
	H2H	62.34±0.25	54.89±0.93	53.08±0.57		
Transformers†	None	76.12	88.12	88.07	85.64	
	Classifier	76.50	86.05	86.68	73.45	
	Attention	**76.63**	87.91	**88.45**	**85.95**	
	Block	75.87	**88.22**	87.88	85.33	

Table 5.2: Results for MULTINLI and SCI with different baselines and their routed versions. We report average accuracy with confidence intervals over five runs with different seeds. For Transformers, we found that finetuning was highly volatile. We therefore report test results from the best-of-5 train models. 'WP' stands for Word Projection, '+D' for Dispatching, 'I2H' for Input-to-Hidden routing, and 'H2H' for Hidden-to-Hidden routing. Italics mark scores whose confidence intervals overlap with the best scores. *All results for nested SCI were computed by fine-tuning the same network previously trained on joint. † Transformers were pretrained as a language model.

5.3.2.1 MULTINLI Results

Our MULTINLI results are given in Table 5.2. The best model combines the Transformer base model with routing the attention layers. Our methods for routing the RNN seem to be less successful, but word-representation routing offers clear benefits with the CBOW base model. Interestingly, the baseline (non-routed) models perform at the same level as the very similar models without genre labels evaluated by [Williams et al., 2018], even though they are also exposed to the genre. It seems that these models are not able to take advantage of the meta-information. In contrast, routing networks seem to provide the space needed to condition linguistic senses on these labels, at least for CBOW models. On further analysis, we found that the problem with RNNs is the challenge discussed in section 4.4: overfitting. These models are too powerful, resulting in near perfect training performance, but unfortunately they do not generalize.

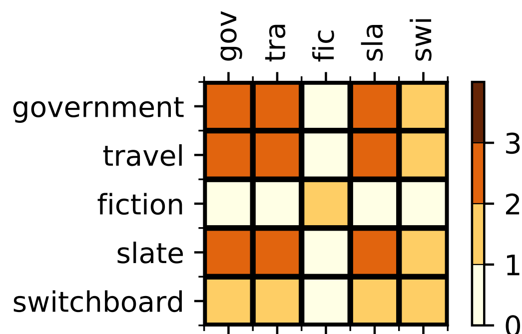


Figure 5.12: Path (module selection) overlap for MULTINLI between genres with the CBOW GloVe WP model. The diagonal represents the number of modules applied for a genre. A maximum of three means that two genres would be routed through the exact same functions.

Our hypothesis is that routing will not only lead to better performance on diverse tasks like MULTINLI, but also that the paths – i.e., the sequence of functions selected by the router – followed by the network will reflect high-level task structure. Figure 5.12 suggests that this is the case for MULTINLI. Here we show the degree of path-overlap for all pairs of genres. As we might expect, government (the 9/11 report), Slate (current affairs), and travel cluster together, distinct from the two more isolated genres (Switchboard; spoken language) and fiction (mostly from the 20th century).

5.3.2.2 SCI Results

For each example in SCI, we use its associated implicative signature as the meta-information label, determining task membership. This serves as a subtler kind of semantic information than genre. As Table 5.2 shows, routing lead to considerable gains in performance over most benchmarks. Routing of the classifier helps consistently with all models. Even simple word-level routing yields major improvements for CBOW. As with MULTINLI, we believe that the modules allow the conditioning of linguistic senses. Routing at the word-level for sequential models seems suboptimal here. The training accuracies ($> 99.5\%$) suggest that the problem is, again, overfitting.

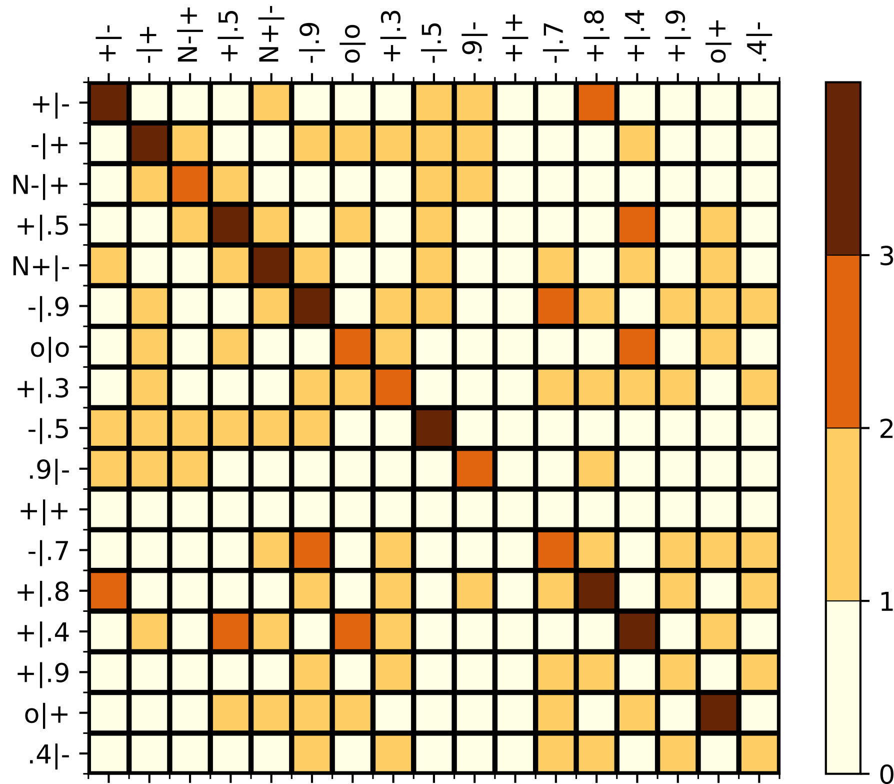


Figure 5.13: The path-overlap between different signatures on SCI, using the CBOW GloVe WP model, for $b = 4$, $d = 3$. N$+|-$ and N$-|+$ are nested signatures.

We expect examples with similar signatures to be routed along similar paths. Figure 5.13 summarizes the path-overlap between different signatures. Many of these similarities make intuitive sense. For example, +|.4 and +|.5 are highly similar, as are −|.7 and −|.9. In addition, the isolation of the unusual signature +|+ (limited to just two constructions) seems expected, as does the affinity of the nested signatures N+|− and N−|+ to their unnested counterparts. However, some signatures only contain very few samples [Cases et al., 2019]. This will probably result in highly noisy routing paths for classes with fewer samples.

5.3.2.3 Navigating the Transfer–Interference Trade-off

Another advantage of language domains over image domains is their relative interpretability. For routing, they allow us to relate specific linguistic phenomena to routing paths. These, in turn, reflect how a routing network navigates the transfer-interference trade-off by

controlling the amount of weight sharing between signatures. While baseline networks can learn to navigate this trade-off if they are optimized for the appropriate objective [Riemer et al., 2019], in general these models do not have sufficient supervision on how to do this.

To illustrate, we ran experiments on baseline models to analyze their behavior for possibly interfering problems. There, we noticed that baseline sequence models learned to classify *take vow* early in the training process, as the word *vow* is lexicalized through training on *make vow* examples. However, performance decreases over time as *take* is lexicalized through training on examples of other implicatives with different signatures, such as *take chance*. This behavior is the characteristic outcome of catastrophic interference: learning *take chance* results in decreased performance on *take vow* as the two examples interfere. This is a particularly revealing instance because these two phrasal verbs are similar on the surface, but have quite different semantic properties, as reflected by their signatures. However, given that routing networks are able to route them differently (Figure 5.13), interference is less likely.

5.3.3 Prediction without Meta-Information at Test-Time

This chapter has illustrated how task information or high-quality meta-information in general can be extremely useful. Unfortunately, such information is oftentime not available at test-time. To compensate, we evaluated an extension to routing networks in which an additional neural network module is trained to assign examples to meta-information classes (genres for MULTINLI; signatures for SCI). This is a form of dispatched routing (compare section 3.4.2), because the task of that model is to learn clusters exclusively. It then acts as a hierarchy on the other sub-routers.

It is important that such a model is not trained to guess the signatures, i.e., trained using the supervisory sample-metainformation tuples, but is instead trained on the final output of the network. When we trained the dispatcher jointly with the models and the subrouters, we found that it was unstable and did not perform well. Instead we trained the dispatcher *after*

the rest of the routed model was already fully trained. The training target was, as before, a translation of the model's output, translated into a reward. The results for this approach are extremely promising. Table 5.2 includes an evaluation of this variant with a CBOW base model and WP routing (denoted as +D). As we can see, accuracy actually *increases* by a small amount over WP alone.

Having +D also allows the network to better generalize to unseen examples and unseen patterns. Consider the relative performance drop for CBOW WP+D from the full joint SCI dataset (75.56%) to disjoint (−0.69%) and from disjoint (74.87%) to mismatch (−3.79%), and compare this to the plain WP version: full (74.95%) to disjoint (−1.04%), and disjoint (73.91%) to mismatch (−5.22%).

Additionally, dispatching allows us to tackle the nested version of the dataset because the router can compose new paths for the compositional signatures contained there. Consequently, we test different architectures on nested by finetuning a model trained on joint first. (i.e., we train a model on joint to convergence, and then take that model and train it on nested.) Table 5.2 shows the truly impressive capability of a routing architecture to generalize in these cases, achieving an 18% performance increase over a standard, non-routed recurrent architecture. This suggests that routing is indeed able to fulfill one of the promises of compositional modularity: recomposing solutions for known problems to solve yet unencountered ones.

CHAPTER 6

DISPATCHED ROUTING

Picking up the previously introduced idea of a dispatcher, a separate network that is trained to guess task (or any meta-information), an obvious question is if we can train such a network in an end-to-end fashion. Thinking about the fundamental three problems of routing, having a separate dispatcher would directly address the clustering aspect to learning-to-partition. In this case, each of the three problems would have a dedicated component. While we introduced the general idea in [Rosenbaum et al., 2017], it did not work in combination with the domains and architectures experimented with. In this chapter, we will introduce some more recent insights and architecture designs. This is very recent work, and the main results were released in [Rosenbaum et al., 2019a].

The previous chapter showed that language is a better test-bed for routing, for two reasons. The first is performance, because routing offers larger benefits there. We assume that this can be explained by a language domain's higher potential for interference. The second is interpretability, because word-based routing models allow to intuitively grasp specific routing decisions.

Consequently, this chapter works on the tacit assumption that the presented ideas will be applied to language models. However, the core insight into how dispatching can help design stable routing networks should translate to any domain.

6.1 Conceptual Advantages of Dispatching

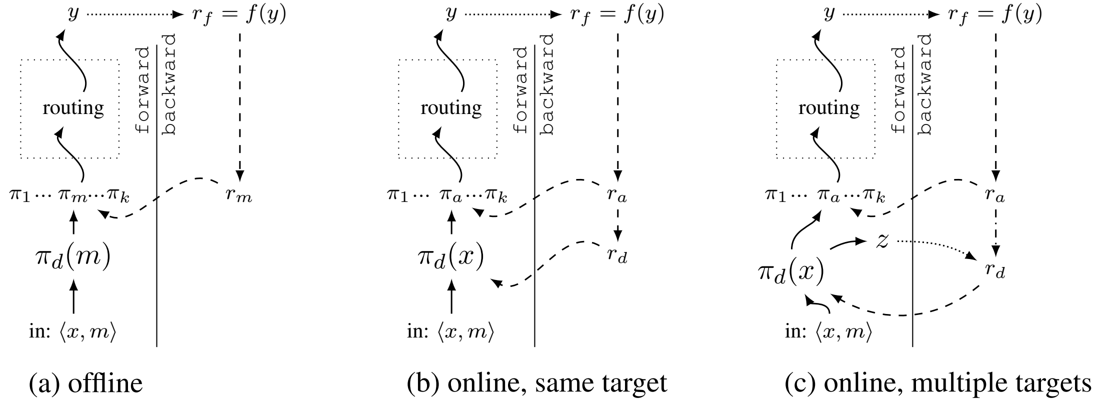
In chapter 4, we detailed several challenges to all compositional architectures, and to routing in particular. The most important for dispatching is how it can address the problems

of finding the 'alignment' of samples, as expressed by the flexibility dilemma of modular learning. Dispatching can have special properties that can help to find the right local approximations in a unique way. The first is that the dispatcher's objective function does not have to be the same as the one of the routing network. Given that the dispatcher is trained on the loss of the routing network or some derivative thereof, it can be regularized with an additional loss (or, for RL dispatchers, reward). This idea will be discussed in more detail below, and is depicted in Figure 3.4(c). If appropriately chosen, this regularization loss can prevent collapse. It can also prevent overfitting, because it may prevent the formation of very small clusters.

The second advantage of dispatching can be related to the problem of learning-to-partition. More specifically, dispatching introduces a specific component that only learns to cluster a distribution, thereby finding distinct groups of samples functionally similar in light of the downstream task. Having such a specific component reduces the burden on the router, but it also affects the routing search space. Given that there is only one decision to take, namely assigning a sample to a cluster, the number of paths is non-combinatorial, which adds an additional safeguard against overfitting. Additionally, a dispatcher can also help with the second problem of learning-to-partition, i.e., learning distances between clusters. It does so because it allows parts of the router to be implemented in a way that ignores the current activations (as done, e.g., by a tabular router). The router can then learn distances purely based on the dispatched cluster, dramatically simplifying the learning problem.

The third advantage is how dispatching can be combined with well-investigated strategies for clustering. As argued before, dispatching effectively clusters samples first which are then uniformly routed. This allows us to interpret these clusters and to investigate if there is a relationship between these 'functional' clusters and clusters naturally ascribed by humans [Potts, 2019].

Additionally, while the natural question is which clusters are learned when dispatching, we may also ask how useful existing clustering strategies are to solving the problem at hand.



(a) offline (b) online, same target (c) online, multiple targets

Figure 6.1: Dispatching strategies. In *offline* dispatching, the dispatching strategy π_d is decided by information provided externally (m), and the dispatcher is not trained as part of the routing network, i.e., it is not part of the feedback loop. In *online* dispatching, the dispatcher is trained, and is thus part of the feedback loop. When trained on the *same target*, the objective function of the dispatcher is the same as for the router. When trained on a *different target*, the dispatcher has another objective function (and is part of another feedback loop), and is trained on both that target and the target provided by the routing network.

This means that we could cluster the data, either in an online or offline fashion, and use these clusters as 'tasks', looking at the resulting performance of the routing network.

6.2 Dispatching Strategies

We introduced several kinds of dispatching. The first is offline dispatching, the second is online dispatching with the same objective function (as in the original formulation by [Rosenbaum et al., 2017]), and the third is online dispatching with multiple objectives. We will investigate all three, with a focus on using existing clustering algorithms for offline dispatching, because we want to move beyond relying on externally provided meta-information.

6.2.1 Offline Dispatching by Clustering

Offline dispatching, depicted in Figure 6.1(a) relies on cluster-information available *before* the routing network is trained. In the previous chapter, we investigated human annotations that can be interpreted as task-information [Rosenbaum et al., 2017, Cases et al.,

2019]. This can be naturally extended to clusters that are not intertwined with the training of the routing network, but that are still learned, if separately. This may find clusters that are relevant for the objective of the routing problem.

The easiest offline clusters can be retrieved by computing an encoding of all the samples in a dataset, which will then be clustered. For language domains such as natural language inference (the domain that will be the focus in this chapter), the simplest of these encodings are CBOW encodings that simply add the respective word (GloVe, [Pennington et al., 2014]) embeddings to form a single vector representation. This vector representation, in turn, can be clustered by any clustering algorithm. We consider two: K-Means clustering and Agglomerative Clustering, because these are established clustering baselines. We also experimented with DBSCAN, but found that the space of encodings is such that DBSCAN does not converge.

6.2.2 Online Dispatching without an additional Objective Function

Both kinds of online dispatching (this and the following one) introduce a clear hierarchy in the online decision making process. The dispatcher acts as an additional step in sample-space that assigns a sample to a secondary subagent which will do the actual routing. For online dispatching without another regularizer, depicted in Figure 6.1(b), the dispatcher is trained on the same objective as the other routing decisions. This objective might be final performance, or any other function of the final output of the routed model. This architecture has been unsuccessfully tried before in [Rosenbaum et al., 2017, Cases et al., 2019]. However, we will try several variations in the hope of finding a working configuration.

6.2.3 Online Dispatching with an additional Objective Function

This approach to dispatching tries to mitigate both the problems of offline dispatching and single objective online dispatching by merging the core ideas of both. It achieves this by training the dispatcher on a combination of an external objective, such as general purpose clustering objective, and the final routing objective, depicted in Figure 6.1(c). This can be

achieved by optimizing the dispatcher, a reinforcement learning agent, on a weighted sum of rewards corresponding to these objectives, also illustrated in Figure 6.2:

$$\mathcal{R}_{total} = \alpha\mathcal{R}_{dispatcher} + (1-\alpha)\mathcal{R}_{external} \tag{6.1}$$

One very simple external objective that should not be interfering catastrophically with the main objective is self-reconstruction. That is, the dispatching action is not only trained on the performance on the downstream task, but also on its performance on a *bottleneck autoencoder*. A (routed) bottleneck autoencoder (RBAE), as depicted in Figure 6.3, differs from a 'regular' autoencoder in that it can choose the projections to form its smallest, inner representations. The intuition is that, given that the innermost dimensionality is small enough, the RBAE cannot compress all samples with the same projections, but instead needs to diversify. Because this process also involves hard decisions, the bottleneck can be implemented as a single-step routing network. The routing component of the autoencoder can thus be defined as solving the following optimization problem (with M as the set of available modules, F_{pre} the encoding preprocessing, and F_{post} as the postprocessing, Enc the encoding, S the phrase and MSE as the mean-squared difference):

$$\underset{m\in M}{\text{argmax}} -MSE(E(S), F_{post}(m(F_{pre}(Enc(S))))) \tag{6.2}$$

A natural translation of this problem into a routing reward is the negative reconstruction loss, similar to the previously discussed negative loss at the downstream task:

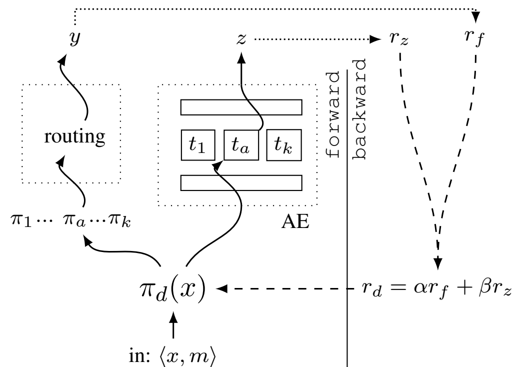


Figure 6.2: Online dispatching with an additional objective function. The dispatcher chooses an action which determines both the sub-policy routing the sample and an external regularization action. The dispatcher is then trained on feedback from both. The external regularization problem may be any single-step routing network, with any target z.

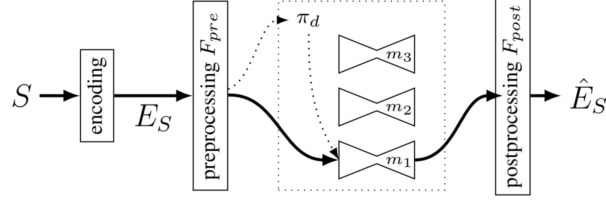


Figure 6.3: A (routed) bottleneck autoencoder. After the input S is encoded, the encoding E_S is projected to some dimensionality d_p, and then passed through one of a set of “bottleneck” modules, $\{m_1, m_2, m_3\}$, selected by the autoencoder dispatching policy π_d. These are of the form $fc_{d_p,d_{bn}} \rightarrow fc_{d_{bn},d_p}$, i.e., they first project the input from dimensionality d_p to a bottleneck dimensionality d_{bn} (such that $d_{bn} \ll d_p$). The resulting projection is then projected back from d_p to the dimensionality of the embedding. The entire model, including π_d, is trained on the reconstruction error between $\hat{E}_S$ and E_S. If d_{bn} is sufficiently small, the π_d has to cluster samples by distributing them over multiple bottleneck modules.

$$\mathcal{R}_{RBAE} = -\mathcal{L}_{reconstruction} \tag{6.3}$$

Because the dispatching decisions also decide on the clusters used for the downstream task, we can easily design an architecture with one subrouter deciding both the bottleneck and the dispatching strategy. This subrouter effectively acts as a dispatcher, but we can then train it on the combined reward on the downstream task and the self-reconstruction task of the RBAE, as outlined in equation (6.1). A simple way to modify this architecture is to use different encodings for the RBAE. With the focus of this work was language, we designed autoencoders relying on the following encodings.

6.2.4 CBOW Autoencoding

In the simplest case, the autoencoder works immediately on a plain CBOW encoding on some word embedding, GloVe in our case. The autoencoder encoding is then defined as (with S as the premise, and w as the words in S):

$$Enc(S) = \sum_{w \in S} GloVe(w) \tag{6.4}$$

6.2.5 CBOW Autoencoding with Attention

In this modification, the autoencoder works on a weighted CBOW encoding, where a learned weight is associated to each word. This approach is commonly known as an attention model. The corresponding autoencoder encoding is defined as (with S as the premise, w as the words in S, and A as the attention function):

$$Enc(S) = \sum_{w \in S} A(w) GloVe(w) \tag{6.5}$$

6.2.6 One-hot (unencoded) Autoencoding

This simpler version of an autoencoder does not rely on an pre-trained embedding, but instead uses a sum of one-hot word representations. This forces the autoencoder to develop a simple language model. The encoding is (with S as the premise, and w as the words in S):

$$Enc(S) = \sum_{w \in S} OneHot(w) \tag{6.6}$$

6.2.7 Sequence to Sequence Autoencoding

Another autoencoder we experiment with is a full sequence-to-sequence autoencoder. Here, the input is encoded using a recurrent neural network (an LSTM, to be precise). As before, this encoding is then pushed through a routed bottleneck. Finally, the encoding is used as the input to a sequential *decoder* that reconstructs the full input sequence. The encoding is (with S as the premise, and w as the words in S):

$$Enc(S) = SEQ(S) \tag{6.7}$$

6.3 Experiments

In this section, we will evaluate the proposed architectures on the Stanford Corpus of Implicatives (SCI) [Cases et al., 2019], which was introduced in section 5.3.

6.3.1 Quantitative Results

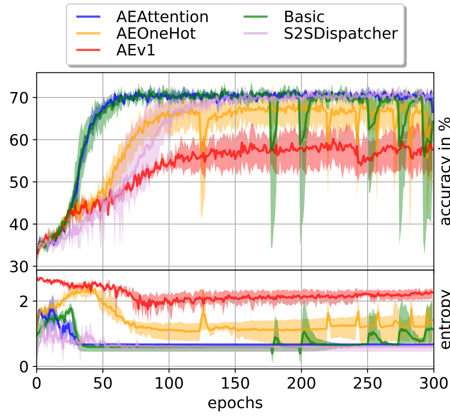
The first experiment evaluates the performance on SCI over the different approaches to dispatching. To get a better sense of how the dispatching strategies are able to navigate the flexibility dilemma between collapse and overfitting (see sections 4.3 and 4.4), all results also include the dispatching selection entropy. The results are shown in Table 6.1.

The results presented in the previous chapter form the baselines for these experiments. These are: the (unrouted) sequential model, the (unrouted) CBOW model and word-level CBOW routing, conditioned on signatures (the first three rows of Tables 6.1, 5.2). The architecture is the same architecture, i.e., routing the individual word projections of a CBOW encoding, because it is clearly the strongest routed architecture from section 5.3. The hyperparameters are the same as in the previous chapter, with a routing width of $b = 15$, a maximal depth of $d = 3$, and a learning rate of $1e - 3$. Further analysis of the impact of the hyperparameters can be found in the following chapter, Chapter 7. In contrast to the previously discussed results, where we claimed that online dispatching without an additional target does not generalize well, the dispatching architecture here encodes the input with a sequential network, and not with a CBOW representation.

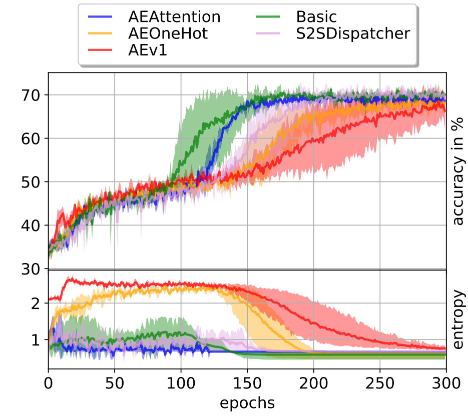
The questions asked here are: (1) Is it possible to achieve the same performance (in particular the impressive boost of simple bag of words architectures) without relying on meta-information, but relying on dispatching only? (2) How does an end-to-end trained dispatcher generalize to the more difficult variants of SCI – disjoint, mismatch, and nested? (3) Are there substantial structural overlaps with the externally provided 'tasks', i.e., the signatures?

6.3.2 Offline Dispatching

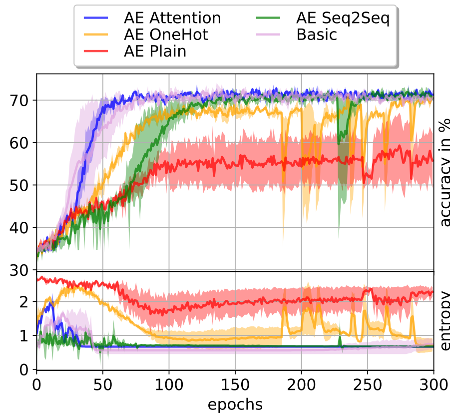
We show results for offline dispatching, i.e., for clusters computed offline before training of the routing network in Table 6.1. We ran experiments with 5, 10, and 20 clusters, and found that 5 clusters works best on average, and is always within the standard deviation of



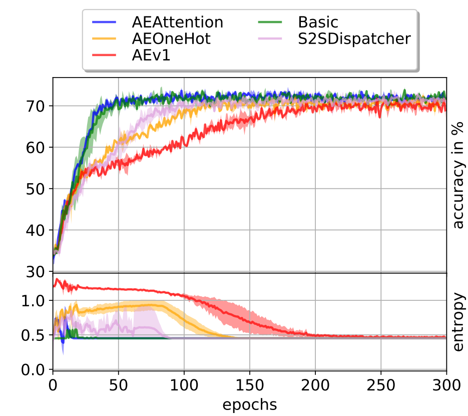
(a) Advantage Learning; Correct Classified Reward; Collab Reward 0.0



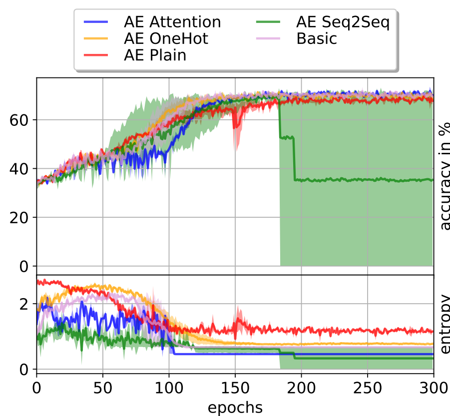
(b) Q-Learning; Correct Classified Reward; Collab Reward 0.0



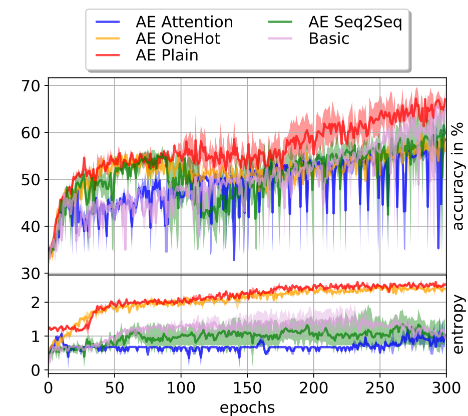
(c) Advantage Learning; Correct Classified Reward; Collab Reward -0.1



(d) Q-Learning; Correct Classified Reward; Collab Reward -0.1



(e) Advantage Learning; Negative Loss Reward; Collab Reward -0.1



(f) Q-Learning; Negative Loss Reward; Collab Reward -0.1

Figure 6.4: Comparison of dispatching architectures

				Joint & Match		Disjoint		Mismatch		Nested	
Enc	Dispatching		#class	Accuracy	Entr	Accuracy	Entr	Accuracy	Entr	Accuracy	Entr
Seq	not routed		1	67.04±2.36	0	63.52±2.00	0	59.32±2.87	0	57.69±2.75	0
CBOW	not routed		1	57.26±0.18	0	55.68±0.47	0	53.41±0.86	0	50.98±0.92	0
	Offline	Signatures	15	**74.95**±0.64	1.82	**73.91**±0.54	1.8	**71.08**±0.52	1.82	**75.43**±0.29	0.46
		K-Means	5	62.26±2.79	0.45	61.68±1.4	0.35	59.29±0.59	0.42	71.67± 4.03	0.39
		Aggl	5	64.63±0.62	0.44	60.86±0.75	0.46	61.56±0.55	0.43	72.45± 1.79	0.59
	Online	Basic	≤ 20	*71.4*±1.39	0.56	64.13±0.3	0.45	**67.77**±0.61	0.45	**83.24**±0.58	0.62
		AE	≤ 20	70.42±0.83	2.58	61.31±0.97	0.84	63.08±0.94	0.59	79.38± 0.99	0.31
		AE Attn	≤ 20	**72.7**±0.21	0.46	**65.05**±0.78	0.45	*67.74*±0.23	0.44	80.35±1.42	0.41
		AE OH	≤ 20	70.89±0.98	0.71	61.94±1.12	0.49	64.01±0.53	0.68	79.77±2.87	0.4
		AE Seq	≤ 20	70.49±0.7	0.45	63.3±1.12	0.55	65.43±0.53	0.45	*82.37*±2.6	0.41

Table 6.1: Test results on SCI, averaged over five runs with different seeds. Each entry consists of the test accuracy with confidence intervals and the entropy of the learned dispatching policy at test time. Each result is selected from the average best-of-five dev results from a hyperparameter sweep and may thereby have different hyperparameter settings. Bold font marks the average best score, and italics mark scores whose confidence intervals overlap with the best scores.

the best performing setting. These clusters are generally inferior to the provided signatures. However, they *do* improve performance over their unrouted baseline. For the mismatched and nested datasets, they allow the routed model to beat even the generally much stronger sequential baseline. In particular for the nested dataset, they beat the baselines by at least 14 % on average. For the two clustering techniques compared, agglomerative clustering generally works better than k-means clustering.

6.3.3 Online Dispatching

Table 6.1 shows the test scores of the different online dispatching algorithms after 300 epochs. Figure 6.4 show dev results over 300 epochs to illustrate the learning behavior over time. Combined, these results allow us to draw some conclusions:

(1) The best-performing dispatching strategies are the autoencoding with attention and, surprisingly, basic dispatching without an external dispatching signal. While these results are (mostly) not competitive with the results relying on human designed signatures, they clearly outperform all baselines.

(2) The 'plain' autoencoder only learning to reconstruct the CBOW encoding has the highest selection entropy. Even when its selection entropy reduces over time, it does so very slowly. Furthermore, all architectures except the 'plain' autoencoder and the onehot autoencoder learn distributions with very similar entropies.

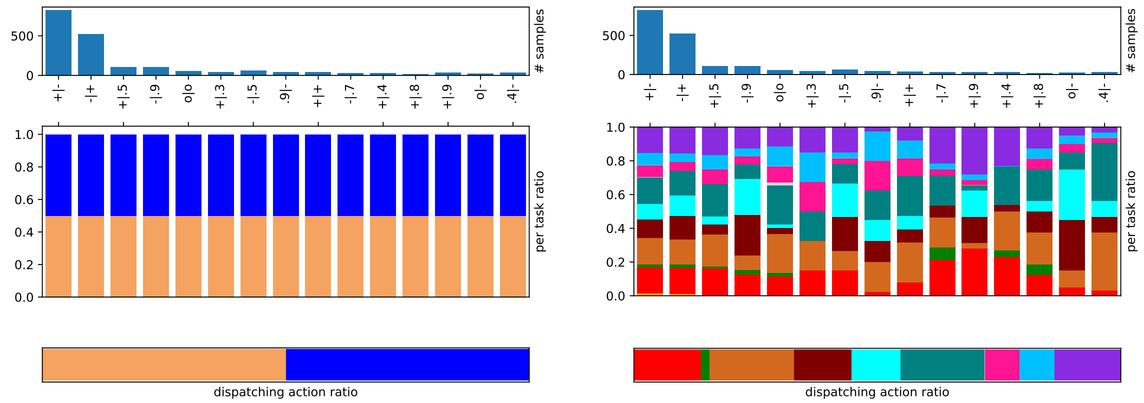
(3) The accuracy results are generally very stable, with most confidence intervals within less than $\pm 1\%$ accuracy for most architectures.

(4) The most impressive result is the ability to generalize to longer (nested) sequences, even outperforming the results relying on signatures.

The interpretation of these results is less obvious. (1) That attention helps the autoencoder is not surprising, because it allows the encoding to learn to ignore stop words, names, and other words irrelevant to the implication question. However, it is surprising that dispatching without any external loss signal is able to learn as well as it does, in particular as [Rosenbaum et al., 2017, Cases et al., 2019] explicitly claim that this does not work well. The main architectural difference, arguably explaining this gap, is that the basic dispatcher used here does not use a CBOW encoding, but a sequential encoding instead. Apparently, the different in expressivity between these two is sufficient to learn meaningful clusters.

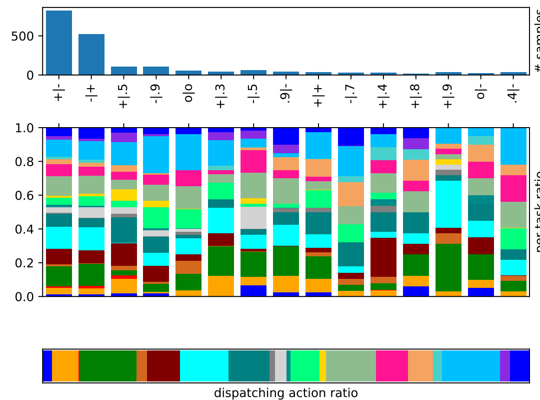
(2) There are several possible explanations for why a particular architecture produces higher or lower entropy. The attention CBOW autoencoder, for instance, can learn to ignore stopwords when compressing, and only focus on relevant keywords. This makes the compression problem easier, probably requiring fewer choices – and thereby allowing lower selection entropy. Similarly, the 'plain' CBOW autoencoder or the onehot autoencoder cannot find such reduction strategies, probably requiring more bottlenecks to route through.

This interpretation is somewhat challenged by the entropy values in table 6.1, because there is no clear picture emerging. While plain autoencoding tends to have the highest entropy, it has the lowest for nested. Generally, most of the entropy results are in the $0.4-0.5$



(a) Entropy: 0.46. Best performing AE+attention dispatcher; Collab Reward -0.1

(b) Entropy: 0.56. Best performing basic dispatcher; Collab Reward 0.0



(c) Entropy: 2.58. Best performing AE dispatcher; Collab Reward 0.0

Figure 6.5: Comparison of the behavior of different dispatching strategies. For each subplot, the topmost graph shows the distribution of the ground truth signatures; the center graph shows the dispatching distribution per signature, and the lowest shows the overall dispatching distribution.

range. As shown in Figure 6.5, this generally corresponds to two dominant dispatching clusters with minor deviations.

(3) The stability of the learning process is not only reflected by the confidence interval of the accuracies, but also extends to the entropy results, because all entropies are in a ± 0.1 range. A particularly interesting result is the plain autoencoder result on joint & match, because it achieves a good score while maintaining extremely high entropy of 2.58.

Additionally, the entropy has a confidence interval of ± 0.01, effectively implying that this model learns highly similar cluster distributions over all runs. This strongly suggests that a dispatcher learns meaningful and global clusters, and not only locally optimal splits.

(4) This result suggests that a dispatched routing network's strongest point is its ability to generalize. What happens intuitively is that a dispatcher can analyze the composition of an SCI premise and then correctly select function blocks that will mirror this composition.

The results presented here confirm that the intuition behind dispatching – having a dedicated architectural component learning latent distances between samples – does indeed simplify the learning process for a routing network. While the proposed architectures do not (yet) allow clustering of the same quality as high-quality meta-information does, their performance strongly suggests that dispatching may offer an end-to-end trainable alternative.

CHAPTER 7

AN EMPIRICAL ANALYSIS

In this section, we will evaluate the impact of numerous design decisions for, and extensions to, routing networks – both for routing networks relying on meta-information and those that do not. We will evaluate the impact of different router architectures, of different decision-making algorithms, and of different hyperparameter settings. We will include several ideas we tried that did not work. While the results may be negative, they allow a further analysis of different properties of routing. This analysis was partially presented in all of [Rosenbaum et al., 2017, Cases et al., 2019, Rosenbaum et al., 2019b,a]. However, most the results were first presented in [Rosenbaum et al., 2019b], with some additions in [Rosenbaum et al., 2019a].

7.1 Decision Making Algorithms

One of the most important questions when designing a router is the choice of decision-making algorithm. As discussed in section 3.1.3, two general classes are applicable: reinforcement learning algorithms and reparameterizations of stochastic distributions. We will show their relative performance, and then offer an analysis to the differences in between algorithms.

7.1.1 Additional Algorithms

Because WPL offered the best performance on image domains, but is in its original form not applicable to routers with function approximators, we designed a version that can be combined with function approximators. We will introduce this version before advancing to further analysis.

Algorithm 7.1: Weighted Policy Learner: Approximation Version

input : A trajectory $T = (S, A, R, r_{final})$, a critic approximator $\hat{R}$ that maps states to approximated returns, and a policy approximator π that maps states to action distributions.

output : Losses for the critic approximator and the policy approximator

```
1  # Initialize the losses:
2  L_critic ← 0
3  L_actor ← 0
4  for each action a_i ∈ A do
5      # Compute the return:
6      R_i = r_final + Σ_{j=i}^{n} γ^{j−i} r_j
7      # Compute the critic loss (the baseline):
8      L_critic ← L_critic + MSE(R̂(s), R_i)
9      # Compute the gradient:
10     Δ(s, a_i) ← R_i − R̂_i(s)
11     if Δ(s, a_i) < 0 then
12         Δ(s, a_i) ← Δ(s, a_i)(1 − P(a_i|s, π))
13     else
14         Δ(s, a_i) ← Δ(s, a_i)P(a_i|s, π)
15     π_target(s) ← simplex-projection(π(s) + 𝟙_{|A|}(a_i) Δ(s, a_i))
16     # Cumulate the actor loss:
17     L_actor ← L_actor + MSE(π(s), π_target(s))
18 minimize L_critic + L_actor
```

The algorithm is shown in 7.1. Here, the reward baseline is learned by a critic, while the actor uses an approximator, i.e., another neural network, to estimate the policy as a function of the state. The reward baseline is learned by training on the MSE between actual and expected reward. The policy gradient is computed using the normal WPL update rule, but is not immediately applied as would be the case for REINFORCE. Instead, the target policy is computed, and the approximator can be trained to minimize the MSE between the target policy and the current policy. While this approximation version works in practice, it is not covered by the guarantees of the original algorithm.

7.1.2 Decision Making Strategies: The Algorithms

For the remainder of this chapter, we will refer to the following decision making algorithms:

- *Q-Learning* We use a vanilla q-learning implementation directly following [Watkins, 1989]. For function approximation versions, we train the approximation network to minimize the ℓ^2-norm of the difference between the current estimate and the target value.

- *Advantage Learning* We additionally experiment with a variant of advantage updating [Baird III, 1993]. That is, we have a learner for the on-policy value function, and a separate learner for the advantage. We include this algorithm to evaluate the argument from Section 3.1.3, that separate learners could stabilize the learning process. When implementing this algorithm, we tried two variations, one Monte Carlo version where we learn the value function by estimating the entire return, and one bootstrapping version that learns the value function by temporal differences. We found that the Monte Carlo version learns faster, and will use that version throughout this section. That is, with $\hat{V}$ as the value estimator, $\hat{A}$ the advantage estimator, t as the current step, d the routing depth, r_f and $r_{i,j}$ as the final and intermediary rewards, and α as the learning rate, we have:

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha \left(r_f + \sum_{j=t}^{d} r_j - \hat{V}(s_t) \right) \tag{7.1}$$

$$\hat{A}(s_t, a_t) \leftarrow \hat{A}(s_t) + \alpha \left(r_f + \sum_{j=t}^{d} r_j - \hat{V}(s_t) - \hat{A}(s_t, a_t) \right) \tag{7.2}$$

- *REINFORCE* We include a basic version of REINFORCE [Williams, 1992] without a baseline. That is, we train the router to minimize $-\log P(a_k|\pi)(r_f + \sum_{j=k}^{t} r_{i,j})$, where k is the current time index, t is the trajectory length, r_f the final reward and r_i any intermediary reward.

– *AAC* We also include advantage actor-critic [Mnih et al., 2016], i.e., we offset REINFORCE with a value-learned critic and train the actor on the advantage estimate.

– *WPL* We use the weighted policy learner as defined in the previous section.

– *Gumbel* The first non-reinforcement learning algorithm we experiment with is reparameterization via the Gumbel-softmax distribution [Maddison et al., 2016, Jang et al., 2016]. The temperature parameter is 1.

– *RELAX* The second reparameterization algorithm we experiment with is RELAX [Grathwohl et al., 2018]. We use a temperature parameter value of 0.5, a value coefficient parameter of 0.5, and an entropy coefficient of 0.01.

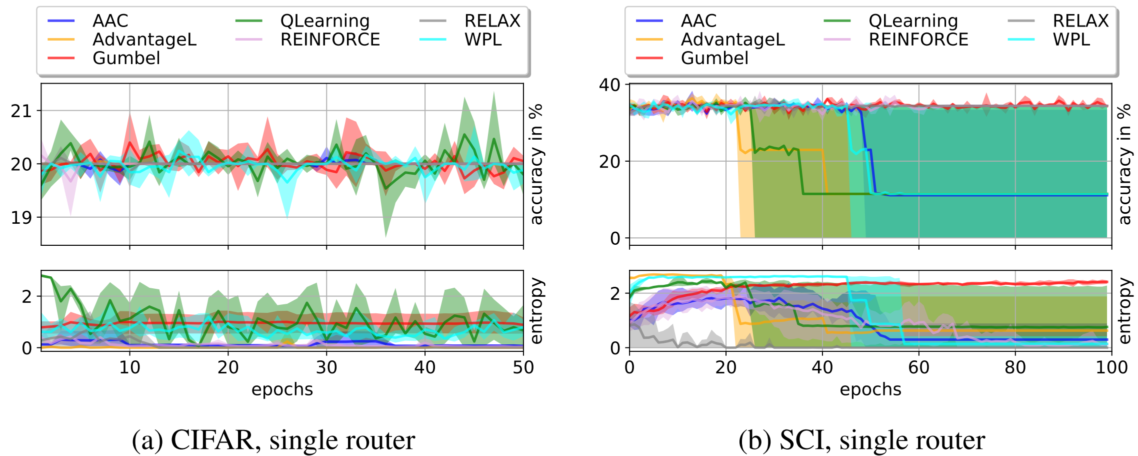
7.1.3 Routing without Meta-Information and without Dispatching

Routing decisions that do not rely on meta-information rely on the activation at the input to the routed layer instead. That is, they only consider the activation produced by the previous layer, which may or may not be routed, and which may even be the input. This allows the most fine-grained control over the routing path, because there are potentially an infinite number of paths, so that each sample could be routed through a different path. The most straightforward implementation is a 'single router' architecture depicted in Figure 3.4(a), where the router consists of only one subrouter, with one parameterization that gets passed in activations from any layer. This also allows arbitrary depth routing networks, because the router may decide to apply an arbitrary number of transformations before stopping. This is intuitively the most difficult scenario to learn, because the router will have to jointly solve the problems of learning-to-compose and learning-to-partition, without any further signal. Considering this from the perspective of the decision-making, the interplay between the three problems creates at least two challenges for the router:

The first is that the subrouter does not only need to learn where to route based on an activation alone, but even needs to do so for activations that may have been produced at

completely different steps in the routing network. This makes the distribution of activations the router will have to decide on dramatically more difficult for the router to interpret, because it is likely that each 'depth' of activations adds another mode to the distribution.

The second, related, reason is that the activations are highly volatile, with any change to the subrouter's policy dramatically transforming the distribution of activations, making the modes non-stationary and even more difficult to disentangle. For illustration, consider a simple single router model where the router can select from $\{t_1, t_2, t_3\}$. At a given training time, the router only selects t_1 or t_2. When an update changes the router to use t_3 instead of t_2, the space of activations will now have been created by $\{t_1, t_3\}^k$, instead of by $\{t_1, t_2\}^k$. It is obvious how this can 'confuse' a router, in particular when this non-stationarity is combined with the already existing non-stationarity produced by updating the transformations.



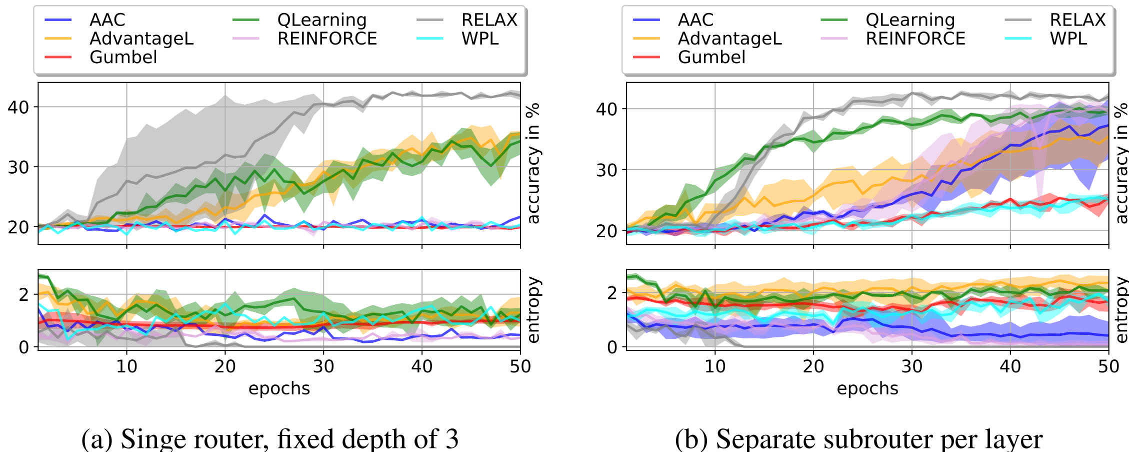
(a) CIFAR, single router

(b) SCI, single router

Figure 7.1: Comparison of different decision-making algorithms on two single agent architectures.

Consequently, performance for single agent architectures on both CIFAR and SCI, as depicted in Figure 7.1 is basically the same as random. For the results on language inference in Figure 7.1(b), the routing problem becomes so complex that the router even picks a fourth class for language inference (which generally only has three) that only exists for implementation reasons, thereby consistently achieving 0% accuracy instead of the 33% of the random baseline.

To further investigate this, we additionally experimented with two more architectures not relying on meta-information. In the first, we fixed the depth for the single router architecture, and in the second, we designed a per-layer subrouter architecture depicted in Figure 3.4(b). Other than the single router architecture used for the experiments in Figures



(a) Singe router, fixed depth of 3

(b) Separate subrouter per layer

Figure 7.2: Decision-making ablation studies on CIFAR. Both experiments compare different decision-making algorithms on restricted architectures.

7.1, this architecture does not have one approximation network over all routing layers, of which there may be an arbitrary number, but instead only makes (at most) k decisions, with a separate subrouter for each layer. As shown in Figure 7.2(a), fixing the routing depth does help to stabilize the training for some algorithms. As with other experiments, the (Q-)value-based approaches start learning, but, more surprisingly, RELAX is able to stabilize, although at the cost of complete collapse. We assume that in this highly volatile environment, the rich gradient information provided by RELAX can help the most. In Figure 7.2(b), we show results for a separate subrouter for each layer, thereby limiting the maximum number of applied transformations to 3 (each subrouter is able to terminate earlier). This makes the decision-making problem easier so that some algorithms are able to stabilize. However, most policy gradient approaches, including RELAX, collapse to achieve this stability, while the remaining policy gradient algorithms, WPL and Gumbel, do not achieve noteworthy performance. Only Q-Learning and Advantage learning learn

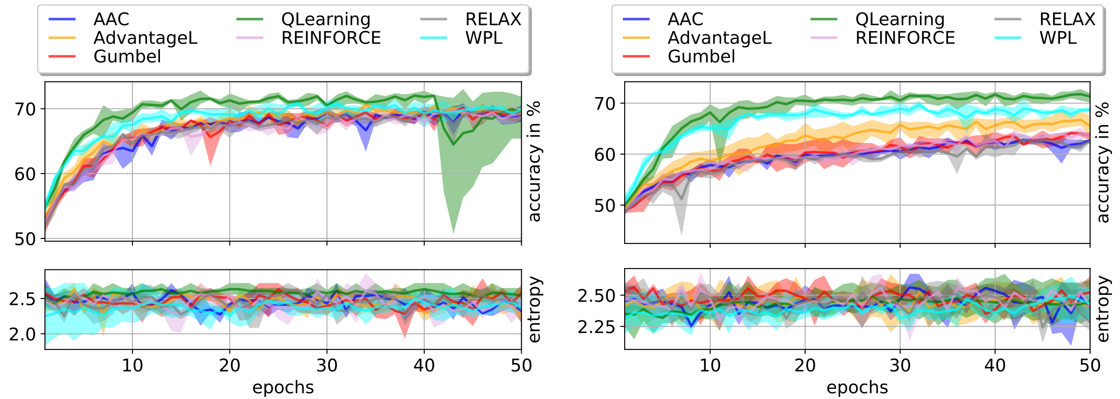
while maintaining selection entropy. Interestingly, the router apparently does not learn to completely minimize interference, as it does not achieve non-routed performance, not even when it collapses. It should be noted, though, that even for algorithms that successfully stabilize, the solutions found are local optima of very bad performance. For both the results in Figure 7.2, the final performance stays about 30% below the performance achieved by models relying on meta-information.

These results strongly suggest that single agent routing networks – and quite probably any other kind of approach to compositional computation without any further stabilizing additions – have difficulty overcoming the initial instability of the training process. There are few solutions apart from the already discussed multi-task/meta-information and dispatching approaches. One, as introduced by [Chang et al., 2019], is to carefully curate the order in which samples are provided to the network. This also drastically limits the complexity of the decision problem, because only 'similar' samples are shown at any given time during training, but relies on such an order to exist for a given distribution.

7.1.4 Routing with Meta-Information

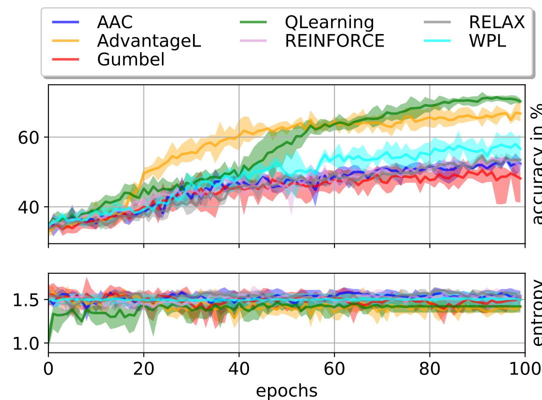
Figure 7.3 shows results for different decision-making algorithms when relying on meta-information. The starkest result is that policy gradient-based approaches (including the related reparameterization algorithms Gumbel and RELAX, but excluding WPL) are consistently outperformed by value-based reinforcement learning based approaches. As can be seen when comparing Figures 7.3(a) with (b), this difference grows with routing depth, i.e., number of routed layers. We stipulate that this difference results from the effect of exploration on interference, as discussed in section 4.3. We will further investigate this argument in section 7.1.6.

In contrast to policy gradient strategies, ϵ-greedy strategies guarantee only limited path interference, even for paths of near-identical value, because exploration is limited by ϵ. This also explains why the weighted policy learner is the best performing policy gradient



(a) CIFAR 100 MTL, with depth of 1

(b) CIFAR 100 MTL, with depth of 2



(c) SCI MTL, with depth of 3

Figure 7.3: Multi-task results for different decision-making algorithms.

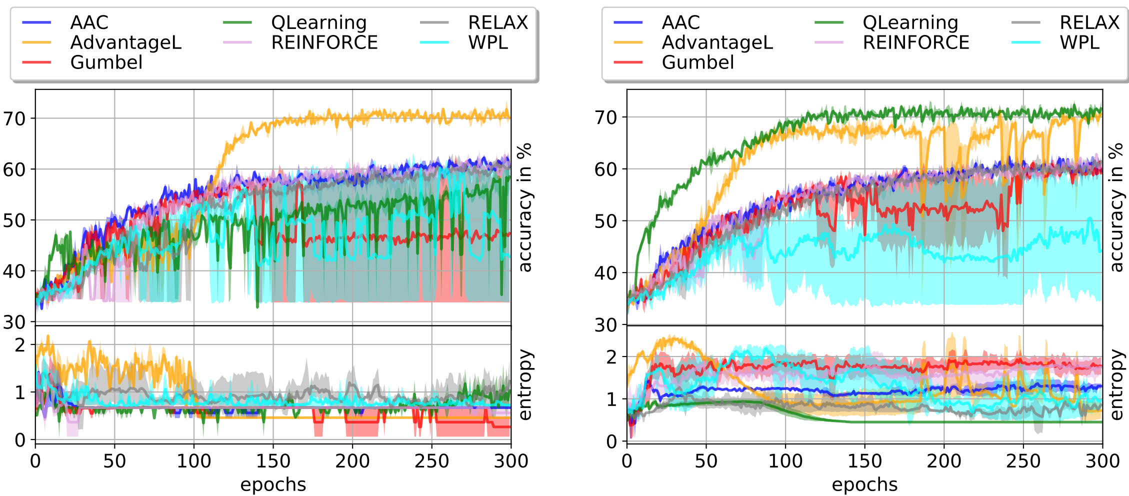
algorithm. It was explicitly designed for stochastic games, and thus has specific properties to emphasize exploitation even for similar-value actions. This effectively lowers its exploration and thereby its interference. We will re-visit how exploration impacts routing in section 7.1.6.

In the language inference experiments in Figure 7.3(c), it is apparent how different algorithms stabilize (or not) over time. As for CIFAR, the value-based approaches Q-Learning and Advantage learning have a clear edge over the policy gradient based approaches. It is particularly interesting to see how both algorithms have nearly the same learning behavior for the first 20 epochs, until Advantage learning, and 20 epochs later, Q-Learning,

stabilize to then quickly leave the other algorithms behind. We speculate that the benefit of Advantage learning stems from the argument in Section 3.1.3, because it is able to offset the general increase in value of the different transformations.

7.1.5 Routing without Meta-Information but with Dispatching

Figure 7.4 shows the same comparison over decision-making algorithms for dispatched architectures on SCI. These re-iterate the previous results that value-based approaches to learning reliably outperform policy gradient based approaches, including reparameterization appraoches. However, within the two value-based approaches evaluated, Q-Learning and Advantage learning, the picture is less clear. While Advantage learning learns more reliably and reaches comparable performance in nearly all cases, as e.g., in Figure 7.4(a), Q-Learning may outperform Advantage learning for specific combinations of hyperparameters, as for Figure 7.4(b). Another interesting result of the plots in Figure 7.4 is that for the dispatched architecture, which trains a dispatcher relying on a neural network function approximator



(a) Attention Dispatcher; Negloss Reward; Collab Reward -0.1

(b) Onehot Dispatcher; Correct Classification Reward; Collab Reward -0.1

Figure 7.4: Comparison of decision-making algorithms over different dispatched architectures and hyperparameter settings.

and the tabular subagents, Gumbel and WPL have problems learning. Additionally, AAC, REINFORCE and RELAX generally work comparatively.

7.1.6 RL vs Reparameterization and the Impact of Exploration

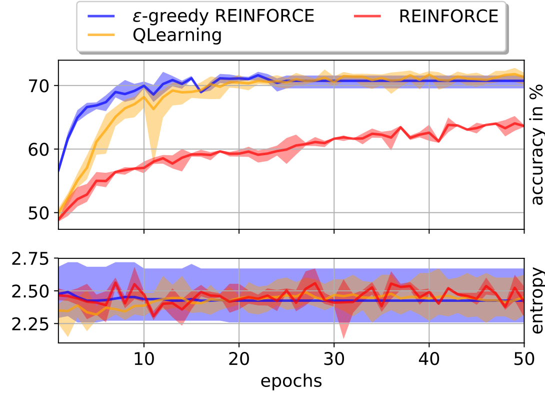


Figure 7.5: An ε-greedy version of REINFORCE on CIFAR 100 MTL

In general, the results for reparameterized routing, i.e., using Gumbel and RELAX, are very similar to the results for policy gradient approaches. We even found that for deeper architectures, the reparameterization techniques suffer from the same decrease in performance as other PG algorithms, as shown in Figure 7.3(b). As mentioned above, we assume that this stems from inferior exploration behavior for on-policy stochastic sampling. To verify, we designed a version of REINFORCE that samples actions using an ε-greedy policy, i.e., that takes the best action ε of the time, and that samples from the actual policy $1 - \varepsilon$ of the time. Given an existing policy $\pi_\theta(s)$, we define $\pi_\theta^\epsilon(s)$ as follows:

$$\pi_\theta^\epsilon(s,a) = \begin{cases} (1-\epsilon) + \epsilon\pi_\theta(s,a), & \text{if } a = \text{argmax}(\pi_\theta(s)). \\ & \text{(For ties, choose the lower-indexed action.)} \\ \epsilon * \pi_\theta(s,a), & \text{otherwise.} \end{cases} \tag{7.3}$$

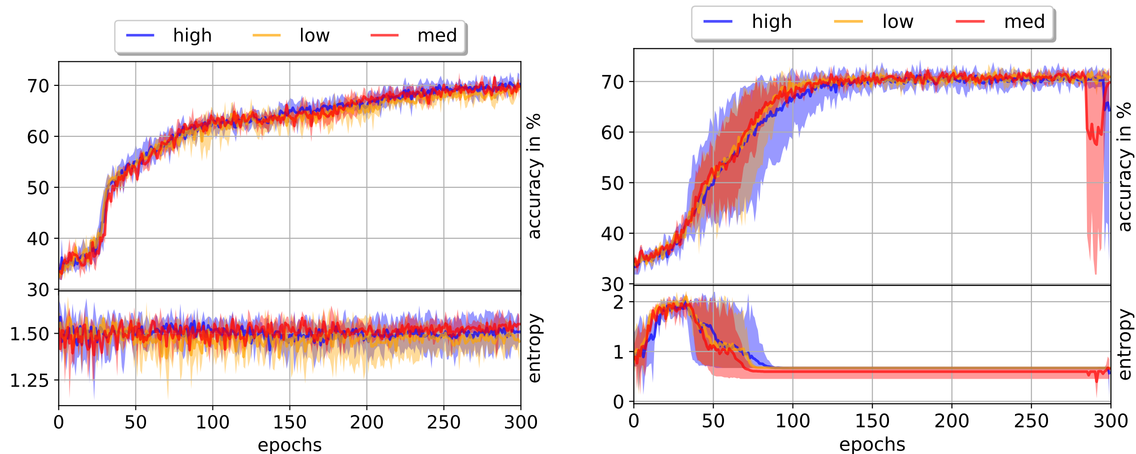
When training, we sample from π_θ^ϵ but update π_θ, and compensate for the difference by scaling the magnitude of the gradient updates using importance weighting [Precup et al., 2000]. As shown in Figure 7.5, this indeed changes the behavior of REINFORCE to act more like a value-based approach. Also note that exploration for the Q-Learning experiments is

very high, with over 0.4 for the first 5 epochs, which suggests that the amount of exploration may play a role, but that the exploitation strategy dominates the results.

These results suggest that routing may greatly benefit from a targeted decision-making algorithm. In particular shaping exploration, with its added complexity of causing interference, seems to be a promising direction for future research. For example, it may be useful to consider algorithms that have separate exploitation and exploration policies, along the lines of [Garcia and Thomas, 2019], or algorithms that can incorporate more complex exploration mechanics into reparameterization techniques.

7.2 Decision Making Hyperparameters

7.2.1 Exploration



(a) Exploration on SCI using signatures as task labels with an advantage learning router.

(b) Exploration on SCI with a dispatched advantage learning router.

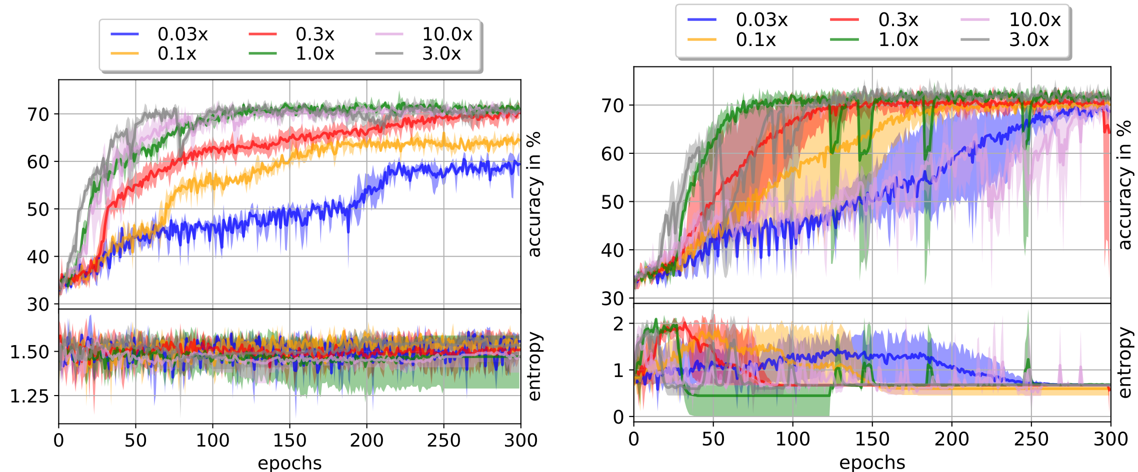
Figure 7.6: Results for different levels of exploration. High starts at 0.5 and anneals every 5 epochs by a factor of 4, until it reaches 0.01. Med starts at 0.25 and anneals every 5 epochs by a factor of 4, until it reaches 0.005. Low starts at 0.1 and anneals every 5 epochs by a factor of 4, until it reaches 0.002.

As already mentioned in section 4, modeling exploration is one of the fundamental problems of reinforcement learning, and of any kind of hard decision-making over unknown

distributions. For routing, this problem is exacerbated in theory by the correlation between exploration and interference.

In practice, however, exploration seems to have less of an impact than expected. Consider Figure 7.6, that shows training for a multi-task and a dispatched architecture. For both architectures, the performance only varies mildly for three different exploration schedules. Interestingly, exploration hardly affects the selection entropy, either. While the expected result, i.e., that higher exploration yields higher entropy policies, holds, it only does so by a negligible margin.

7.2.2 Learning Rates



(a) Router learning rate on SCI using signatures as task labels with an advantage learning router.

(b) Router learning rate on SCI with a dispatched advantage learning router.

Figure 7.7: Results for different routing learning rates, in ratio to the module learning rate of 0.001.

We mentioned before that modeling the learning rates of the modules and of the router separately can be beneficial for the overall learning stability. Because the module learning rate is mostly determined by the problem to be solved, it suffices to analyze the ratio of the routing learning rate to the module learning rate. Consider Figure 7.7, that shows the impact of the router learning rate for a multi-task and a dispatched architecture. Here, the pattern is

clear: the learning rate of the router should be at least as large as the learning rate of the modules. Smaller learning rates delay learning, and may even be detriment to performance. However, the learning rate does not seem to affect the selection entropy.

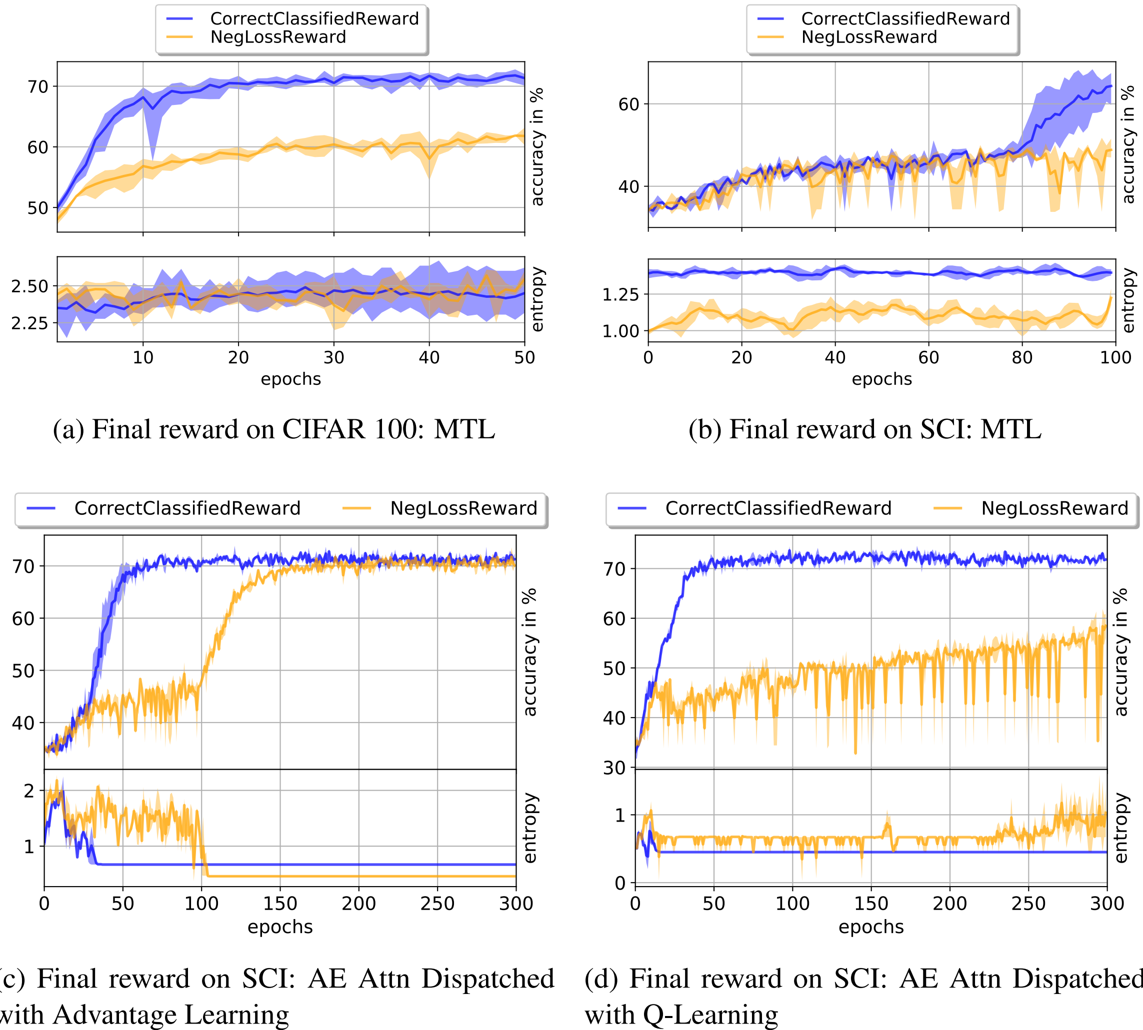
This is a particularly important result, because it allows us to draw some conclusions about how a routing network stabilizes. Intuitively, there are two options: The modules stabilize first, and the router follows, or the router stabilizes its paths, and the modules follow. Because a lower router learning rate makes the paths less flexible, it would benefit the latter, while a larger router learning rate would benefit the former. Consequently, the results in Figure 7.7 strongly suggest that the modules should stabilize first, so that the router can adapt its paths accordingly.

7.3 Reward Design

7.3.1 Final Rewards

Figure 7.8 shows results for different final reward functions. In general, the correct/incorrect final reward strategy, ± 1 for correct and incorrect classification, appears clearly superior to the negative downstream task loss reward $r_f = -\mathcal{L}(\hat{y}, y)$. While this seems initially surprising because the negative loss reward contains more information, we believe that this is exactly what makes learning more difficult, for two reasons. First, a negative downstream task loss may overemphasize outliers, because, relative to other samples, it may add a much larger part to the total loss. While this may not be harmful to supervised learning, a reinforcement learning router may update to accommodate these samples at the cost of other, non-outlier samples.

Second, as suggested by the high fluctuation in the learning on SCI, the finer granularity of the negative classification loss reward may reduce the difference between the value of different transformations, resulting in even small updates changing the greedy strategy. Because this reduces stability, it will also decrease overall performance.



(a) Final reward on CIFAR 100: MTL

(b) Final reward on SCI: MTL

(c) Final reward on SCI: AE Attn Dispatched with Advantage Learning

(d) Final reward on SCI: AE Attn Dispatched with Q-Learning

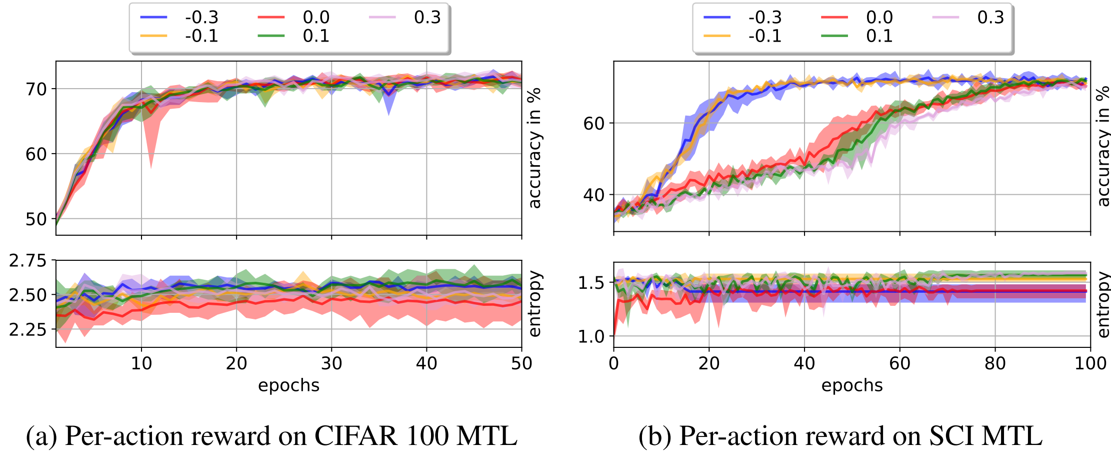
Figure 7.8: Results for different reward functions

This would also explain the stark difference between the choice of decision-making algorithm, as shown for the results on a dispatched architecture in Figures 7.8(c) and (d). While the bias-corrected Advantage Learning can learn a good routing strategy even for the negative loss reward, Q-Learning fails to do so completely.

7.3.2 Regularization Rewards

Figure 7.9 shows plots for different intermediate reward values. These rewards are computed as a fraction of the overall probability of choosing a particular transformation.

Positive values are expected to increase transfer and lower entropy, while negative values are expected to decrease interference and increase entropy. Comparing the results for CIFAR in Figure 7.9(a) with the results for SCI in Figure 7.9(b) it becomes apparent that the domain – or the architecture – plays a major role in the effect of this regularization reward. While the reward has no discernable effect on CIFAR, it dramatically changes the convergence behavior – if not the final performance – on SCI. Interestingly, negative reward values, meant to increase diversity, appear to stabilize learning as the model learns much faster, but



(a) Per-action reward on CIFAR 100 MTL

(b) Per-action reward on SCI MTL

Figure 7.9: Multi-task results for different intermediate reward values

have no effect on the routing entropy. While a complete interpretation of this result is very difficult, we speculate that it results from the higher potential for interference on SCI, and maybe even on language domains in general. The potential for interference is higher when training on SCI because the effective input dimensionality is nearly an order of magnitude lower than it is for CIFAR, which leads to a larger average overlap between examples in the input space. Additionally, as is also the case for us here, models used for NLP tend to use smaller hidden representations than computer vision models, resulting in a larger average overlap between examples in the activation space as well. This would not only explain why 'pushing' the router to diversify stabilizes learning, but also why the results with the negative

loss reward function have such a high variance, because any change in routing decision may end up with a much higher amount of interference.

These results suggest that future research should consider different new reward functions. In particular, it could be worthwhile to find a final reward function that can rely on the information contained in the negative loss, but that is less susceptible to its problems. Additionally, it would be interesting to investigate an adaptive intermediate reward that incentivizes diversity as long as it is needed for stability, but that eventually anneals to zero, so that the router only optimizes for the overall model performance.

7.4 Other Router Design Choices

7.4.1 Entropy Regularization

Considering the danger of collapse, another idea we tried is to regularize the router by adding a negative entropy term to the decision-making loss. The goal is that this term would push the router into making more diverse decisions, until it's routing decisions stabilize. At this point, the term would anneal out, to allow the router to find an unregularized decision-making strategy. A similar idea was introduced to incentivize exploration for policy gradient algorithms [Williams and Peng, 1991]. However, such entropy regularization did simply not work. For small ratios for the entropy term, it did not prevent collapse, and for larger values, it made the training of the router unstable.

7.4.2 Exploratory Actions

Another possibly useful change to the training procedure is to limit the updates of the transformations if they were chosen by an exploratory action. The intuition is that we do not want the network to add interference to the training of modules if they were just evaluated and found not fitting to a particular sample. Because, however, any exploratory action has an effect on any return from the entire trajectory, we simply squash the optimization step size *of the modules* for the particular trajectory using the following formula:

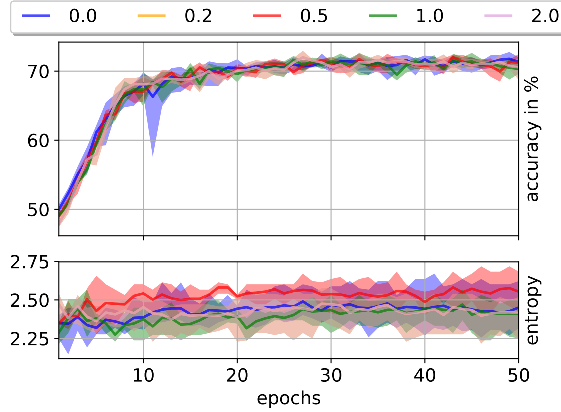


Figure 7.10: Squashing training for exploratory trajectories on CIFAR100 MTL, with values for α_{exp}.

$$\alpha_{exp} = \left(1 - \frac{\#exploratoryactions}{trajectorylength}\right)^{\kappa} \tag{7.4}$$

where α_{exp} is factor to the learning rate, and κ is a hyperparameter. For $\kappa = 0$, exploratory actions will be treated no differently, while for a very large κ, the transformations are effectively not trained on the entire trajectory if even only one action was taken non-greedily.

Figure 7.10 shows the effect of lowering the training of transformations chosen non-greedily. As one can see, κ has very little effect on the overall performance and only mild effect on the entropy. This is a very interesting result, because it tells us that the learned modules are resistant to small injections of noise in their training process. This suggests that they are sufficiently distinct from other modules. If they were not, even a small change could change the router policy, at the very least adding noise to the learning process.

7.4.3 Splitting training data

One of the challenges of training a routing network is overfitting (see Section 4.4 for a thorough explanation). To prevent this, we investigated splitting the training data into a part for training the transformations, and a part to train the router. This has a clear effect on the performance, as shown in Figure 7.11. Here, the curves show the percentage of the data used

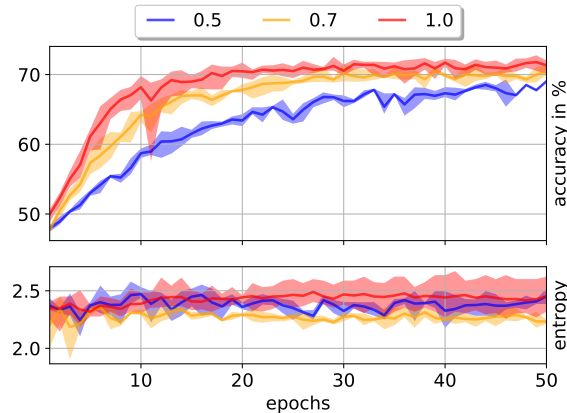


Figure 7.11: Stochastic router-transformation training on CIFAR100 MTL

to train the transformations. As one can see, using the full data benefits both performance and entropy.

7.4.4 Optimization Algorithm / SGD

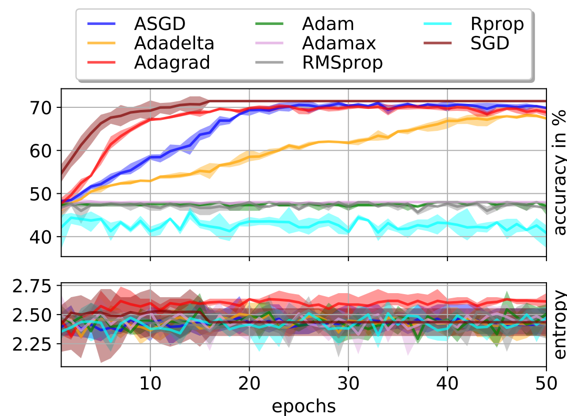


Figure 7.12: The effect of the optimizer choice on CIFAR100 MTL

A very interesting and important result is the impact of the choice of optimization algorithm. Results are depicted in Figure 7.12. As already reported by [Rosenbaum et al., 2017, Cases et al., 2019], routing becomes unstable for many optimizers, and consistently yields the best performance for 'plain' SGD. We stipulate that similar problems can be observed for any architecture with an adaptive computation graph. Consider a simple example, where

we route a sample x through a two-layer routing network. The first routing decision can choose from transformations $\{t_{1,1}, t_{1,2}\}$ and the second from $\{t_{2,1}, t_{2,2}\}$. Now consider the gradients for $t_{1,2}$ over two different routing paths, $t_{1,2}(t_{2,1}(x))$ and $t_{1,2}(t_{2,2}(x))$. It is obvious that for those two paths, $t_{2,1}$ and $t_{2,2}$ may yield vastly different activations, and thereby vastly different inputs to $t_{1,2}$. If an optimizer relies on parameter-specific approximations, such as, e.g., momentum, an approximation for $t_{2,2}$ may be completely incompatible with $t_{2,1}$, thereby making training difficult, or even impossible. This explains why plain optimization strategies that do not compute parameter-specific information generally do better in the context of dynamic computation graphs.

7.4.5 Gradient Flow from Router to Activations

Another idea we evaluated early [Rosenbaum et al., 2017] is to not only train the activations at different routing depths on the external provided loss, but also on their impact on the decision-making. Because this is only applicable to routing with function approximators, the hope was that the modules could learn representations that are both useful for the downstream problem, but also to the router. In terms of a deep architecture, we simply allow the gradient flow from the router through the modules. In practice, this turned out to not work well. Instead of learning good representations for both problems, their training appears to interfere, so that the representations are useless for either.

7.5 Analyzing the Challenges

7.5.1 Stability

As we argued before, stability is a consequence of the routing 'chicken-and-egg' problem: Upon initialization of a routing network, both the modules and the router do not yet have any information on the problem, and act randomly. The router cannot stabilize, because it cannot discern which selected module caused any increase in performance, and the modules cannot stabilize as they are trained with samples and activations so different that interference

can destroy any learning that may happen. This, in turn, may destroy any progress the router may have made with the credit assignment. This problem is never worse than for the 'single router' architecture (compare Figure 3.4), for the reasons described in Section 7.1.3. Consequently, a routing network may fail to learn anything, as depicted in Figure 7.1.

7.5.2 Module Collapse

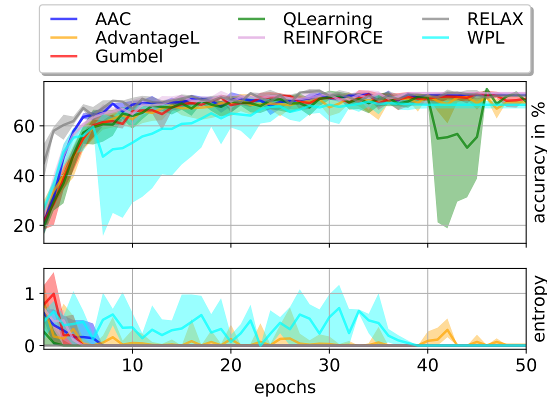


Figure 7.13: Collapse on CIFAR 100 with a dispatched architecture

Figure 7.2 already showed how different router training algorithms can lead to collapse in different domains. Additionally, consider Figure 7.13, where the experiment is on CIFAR with a naïve dispatched architecture that is not regularized by any external losses. The same fc layers are routed as in the other experiments. However, the dispatching action is based on the activation after the convolutional layers. The dispatched subrouters are tabular, and do not consider the intermediate activations. For this architecture, collapse is not a consequence of the routing algorithm, but of the general architecture, because *all* algorithms lead to collapse. However, in difference to the results in Figure 7.2, the results achieved are good, reaching standard non-routed performance.

This is not a surprising result, because collapse *will* stabilize the router, and because there are no later activations to consider. This is reflected by the test-time selection entropy of nearly all algorithms going to zero within ten epochs. What is surprising, however, is that this extends to the stochastic reparameterization algorithms Gumbel and RELAX which do

not rely purely on rewards, further establishing that in the context of routing, Gumbel and RELAX behave 'just as' any other PG algorithm.

The only algorithms that do not collapse completely are WPL and Advantage learning. We assume that WPL does not collapse as much because it was specifically designed for non-stationary returns, and that Advantage learning performs well as a consequence of the argument put forth in 3.1.3.

7.5.3 Overfitting

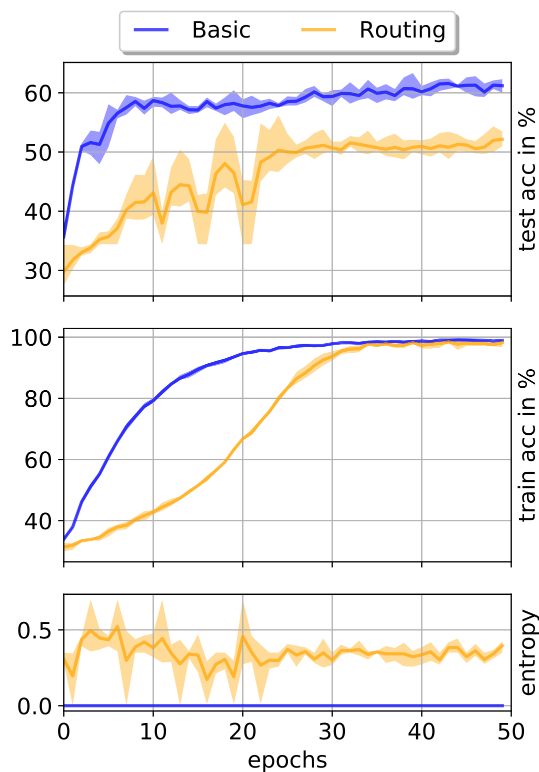


Figure 7.14: Overfitting on SCI

The main reference architectures used until this point – routing the classifier of a convolutional network, and routing the word representations of a natural language inference model – were chosen because of their performance, and overfit only little.

However, if we consider an sequence-to-sequence architecture for language inference with a dispatched routed classifier, we can show how routing can lead a model to overfit.

(We made similar claims for different architectures throughout this work.) Consider Figure 7.14, which shows from top to bottom the test accuracy, train accuracy and test entropy on SCI. While both a basic, non-routed network and the routed network easily reach perfect train accuracy ($> 99.5\%$), the routing architecture starts flattening out even earlier than the basic architecture, reaching accuracy around 10% lower than the basic model, and over 20% lower than the routed model.

As discussed in section 4.4, overfitting is less of a problem in the presence of metainformation, because the router can ignore spurious activations in these cases. However, because the goal of modular architectures is to adaptively modularize without any kind of external information guiding the composition, future work should investigate how routing architectures can be regularized to achieve good generalization properties, possibly along the lines of hierarchical dispatching.

CHAPTER 8

CONCLUSIONS

This thesis is about modularity and compositionality. It is about the belief that any real progress towards artificial general intelligence has to compartmentalize skills to combine them ad-hoc for any new task. I do not and will not claim that routing can achieve this. It is only a first step towards this goal. I hope that even this step is able to yield important insights into not only the challenges of what lies ahead, but also into why approaches of this kind are worth pursuing.

8.1 Contributions

Generally, modular and compositional computation promises two advantages. The first is that its modularity allows the transfer-interference trade-off to be elegantly navigated. That is, it allows a learner to learn similar skills with a large overlap in modules and dissimilar skills with a small, or no, overlap in modules. Intuitively, it *specializes* specific modules to learn specific skills. Because it uses hard decisions, and not more common mixture approaches, it can explicitly model aspects like the relationship between exploration and interference, which are important factors in navigating the trade-off during learning. In comparing soft (or mixture) and hard architectures in section 5.2.3, we could verify the advantage of hard decisions in this context.

The second advantage is generalization via composition. For example, consider our ability to put together a piece of Ikea furniture. Even if we have never seen that particular piece, and if we do not study the manual, we would probably be able to put most of the simpler furniture together without problems (given some experience assembling somewhat

similar Ikea furniture). We do that so easily because each required step is compartmentalized, and either completely or partially known to us already, so the only challenge is *composing* already-known pieces into a new skill. This ability to solve specific problems without training is called *zero-shot learning*, something machine learning algorithms are still tremendously bad at. But now imagine a fully modular machine learning architecture that was able to compose in a way as we do on the fly. Then, generalization of this kind might become feasible.

In this light, this thesis consists of three major contributions. The first is conceptual, and is the fundamental idea of *routing*. Routing is arguably the most general formulation of functional composition, and generalizes several approaches introduced over the last couple of years, because it jointly learns modules, paths, and partitions of the sample distribution.

At its core, it is a very simple idea: have a so-called *router*, i.e., a decision-making algorithm, pick from a set of (compatible) modules that transform the current activation until it decides that the last activation can solve the current task, and then training both modules and router online. This allows it to navigate the transfer-interference trade-off, because the router can route samples with incompatible training information through different paths. It can also reach the second goal of compositional learning, generalization, by combining already pretrained modules in new ways, thereby covering yet unseen examples.

The second contribution is analytical, and consists of a conceptual analysis of modular and compositional approaches, with a focus on the challenges of training them. They arise from an unfortunate mingling of the transfer-interference trade-off and the exploration-exploitation dilemma.

The first is the mutual non-stationarity of the two optimization processes of tranining the modules and training the router. From the perspective of the router, the environment is

non-stationary as the modules change. From the perspective of the modules, their input and output connections change whenever the router changes the paths.

The second and third challenges revolve around routing networks being models of conditional computation, creating different implicit models for different clusters of samples. We identified the ‘flexibility dilemma for modular learning’ that describes the problems when a modular approach is too quick or too slow in creating new clusters in sample-space. On the one hand, initialization conditions can make a router collapse, using too few clusters and thereby underfitting. On the other hand, if a router creates too many clusters, it runs the risk of finding hyperlocal approximations that do not generalize well to other examples, i.e., overfitting.

The last contribution is purely empirical and consists of a suite of experiments. We evaluated the capacity of routing networks to navigate the transfer-interference trade-off, their capacity for generalization and the impact of many different design decisions.

Some highlights of this analysis are: Experiments on multi-task learning domains, in particular on MNIST-MTL and on SCI, where routing can navigate the transfer-interference tradeoff; experiments on SCI, where a routing network generalizes *much* better to complex and nested samples than comparable non-routed baselines; experiments evaluating different architectures, such as single agent, multi-agent and dispatched architectures; and an empirical analysis of the impact of optimization algorithms, among other design decisions.

8.2 Future Work

Routing, being a very recent paradigm, offers a lot of space for improvement. The first area is addressing specific issues that arise for routing networks as introduced here. The second is questioning if there is a fundamental modification to the routing paradigm that may be better suited to achieve compositional zero-shot generalization.

For the first area, I propose the following core goals of research, among others: The first are architectural extensions. We have shown in chapter 6 how including a so-called dispatcher can help to manage the fundamental challenges to routing. It is possible that similar ideas could be used to stabilize the network, to orthogonalize the transformations defined by different modules, or to prevent overfitting and collapse. The second goal is deriving a better, routing-specific decision making agent. The core problem is, as we have shown, that for a routing network, exploration correlates with interference. This means that static exploration architectures give consistently better results. However, these approaches do not not utilize the rich information that e.g., RELAX provides. The third goal is developing a full theoretical framework for routing. This may not only yield interesting insights into how routing works fundamentally, but may also guide the development of new and better architectures and decision making algorithms.

For the second area, the goals are more abstract, because they might require a re-thinking of the routing paradigm. One such goal would be to explore if it may not be useful to have multiple (non-additive) module selections per depth. The intuition would be "evidence aggregation". Instead of picking modules and applying them to an activation, and then recursively updating that activation, it may be useful to "gather evidence", by sequentially collecting activations of the same depth and combining them. This may be combined with the second goal, architectural diversity. While already discussed in the challenges (see section 4.5), it feels as if routing as it is may be too limited to overcome the 'module selection problem' without some fundamental changes or additions.

BIBLIOGRAPHY

Sherief Abdallah and Victor Lesser. Learning the task allocation game. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 850–857. ACM, 2006. URL http://dl.acm.org/citation.cfm?id=1160786.

Ferran Alet, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Modular meta-learning. *CoRR*, abs/1806.10166, 2018. URL http://arxiv.org/abs/1806.10166.

Rahaf Aljundi, Punarjay Chakravarty, and Tinne Tuytelaars. Expert gate: Lifelong learning with a network of experts. *arXiv preprint arXiv:1611.06194*, 2016.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Deep compositional question answering with neural module networks. *CoRR*, abs/1511.02799, 2015. URL http://arxiv.org/abs/1511.02799.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to compose neural networks for question answering. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1545–1554. Association for Computational Linguistics, 2016. doi: 10.18653/v1/N16-1181. URL http://aclweb.org/anthology/N16-1181.

Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. *CoRR*, abs/1606.04474, 2016. URL http://arxiv.org/abs/1606.04474.

Leemon C Baird III. Advantage updating. Technical report, WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1993.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *ICLR*, 2017.

William Bechtel and Adele Abrahamsen. *Connectionism and the mind: Parallel processing, dynamics, and evolution in networks*. Blackwell Publishing, 2002.

William Bechtel and Robert C. Richardson. *Discovering Complexity: Decomposition and Localization as Strategies in Scientific Research*. Mit Press, 2010. ISBN 9780262514736.

Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 549–558, 2018.

Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *CoRR*, abs/1511.06297, 2015. URL `http://arxiv.org/abs/1511.06297`.

Yoshua Bengio. Deep learning of representations: Looking forward. In *International Conference on Statistical Language and Speech Processing*, pages 1–37. Springer, 2013.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. *CoRR*, abs/1708.05344, 2017. URL http://arxiv.org/abs/1708.05344.

Rich Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, Jul 1997. ISSN 1573-0565. doi: 10.1023/A:1007379606734. URL https://doi.org/10.1023/A:1007379606734.

Ignacio Cases, Clemens Rosenbaum, Matthew Riemer, Atticus Geiger, Tim Klinger, Alex Tamkin, Olivia Li, Sandhini Agarwal, Joshua D. Greene, Dan Jurafsky, Christopher Potts, and Lauri Karttunen. Recursive routing networks: Learning to compose modules for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, June 2019.

Michael Chang, Abhishek Gupta, Sergey Levine, and Thomas L. Griffiths. Automatically composing representation transformations as a means for generalization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=B1ffQnRcKX.

Corinna Cortes, Xavi Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. Adanet: Adaptive structural learning of artificial neural networks. *arXiv preprint arXiv:1607.01097*, 2016.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

Martha J. Farah. Neuropsychological inference with an interactive brain: A critique of the "locality" assumption. *Behavioral and Brain Sciences*, 17(1), 43-104, 1994.

Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

Jerry A. Fodor. *The Language of Thought*. Harvard University Press, 1975.

Jerry A. Fodor. *The Modularity of Mind*. MIT Press, 1983.

Jerry A. Fodor and Zenon W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988.

Francisco M. Garcia and Philip S. Thomas. A meta-mdp approach to exploration for lifelong reinforcement learning. *CoRR*, abs/1902.00843, 2019. URL http://arxiv.org/abs/1902.00843.

John J. Godfrey and Ed Holliman. Switchboard-1 release 2. Linguistic Data Consortium, Catalog #LDC97S62, 1997.

Will Grathwohl, Dami Choi, Yuhuai Wu, Geoff Roeder, and David Duvenaud. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=SyzKd1bCW.

Nicholas Hay, Stuart Russell, David Tolpin, and Solomon Eyal Shimony. Selecting computations: Theory and applications. *arXiv preprint arXiv:1408.2048*, 2014.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

Robert A. Jacobs, Michael I. Jordan, and Andrew G. Barto. Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. *Cognitive Science*, 15(2):219 – 250, 1991a. ISSN 0364-0213. doi: https://doi.org/10.1016/0364-0213(91)80006-Q. URL `http://www.sciencedirect.com/science/article/pii/036402139180006Q`.

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991b.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

Lauri Karttunen. Implicative verbs. *Language*, 47(2):340–358, 1971. doi: 10.2307/412084. URL `\http://www.jstor.org/stable/i217134`.

Alexander J.E. Kell, Daniel L.K. Yamins, Erica N. Shook, Sam V. Norman-Haignere, and Josh H. McDermott. A task-optimized neural network replicates human auditory behavior, predicts brain responses, and reveals a cortical processing hierarchy. *Neuron*, 98(3): 630 – 644.e16, 2018. ISSN 0896-6273. doi: https://doi.org/10.1016/j.neuron.2018.03.044. URL `http://www.sciencedirect.com/science/article/pii/S0896627318302502`.

Louis Kirsch, Julius Kunze, and David Barber. Modular networks: Learning to decompose neural computation. In *Advances in Neural Information Processing Systems*, pages 2414–2423, 2018.

James Kostas, Chris Nota, and Philip S Thomas. Reinforcement learning without backpropagation or a clock. *arXiv preprint arXiv:1902.05650*, 2019.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

Jason Zhi Liang, Elliot Meyerson, and Risto Miikkulainen. Evolutionary architecture search for deep multitask networks. *CoRR*, abs/1803.03745, 2018. URL `http://arxiv.org/abs/1803.03745`.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *The European Conference on Computer Vision (ECCV)*, September 2018.

Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3994–4003, 2016.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

Rowan Nairn, Lauri Karttunen, and Cleo Condoravdi. Computing relative polarity for textual inference. In Johan Bos and Alexander Koller, editors, *Inference in Computational Semantics (ICoS-5)*, pages 67–76. University of Manchester, Manchester, UK, 2006. URL `http://www.aclweb.org/anthology/W06-39`.

Alex Nichol and John Schulman. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2018.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL `http://www.aclweb.org/anthology/D14-1162`.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Christopher Potts. A case for deep learning in semantics: Response to Pater. *Language*, 95 (1):e115–e125, 2019.

Doina Precup, Richard S Sutton, and Satinder P Singh. Eligibility traces for off-policy policy evaluation. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 759–766. Morgan Kaufmann Publishers Inc., 2000.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. OpenAI, 2018. URL https://blog.openai.com/language-unsupervised/.

Janarthanan Rajendran, P. Prasanna, Balaraman Ravindran, and Mitesh M. Khapra. ADAAPT: attend, adapt, and transfer: Attentative deep architecture for adaptive policy transfer from multiple sources in the same domain. *ICLR*, abs/1510.02879, 2017. URL http://arxiv.org/abs/1510.02879.

Prajit Ramachandran and Quoc V. Le. Diversity and depth in per-example routing models. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BkxWJnC9tX.

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. *ICLR*, 2017.

John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

Matthew Riemer, Ignacio Cases, Robert Ajemian, Miao Liu, Irina Rish, Yuhai Tu, and Gerald Tesauro. Learning to learn without forgetting by maximizing transfer and minimizing interference. In *In International Conference on Learning Representations (ICLR)*, 2019.

Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. Routing networks: Adaptive selection of non-linear functions for multi-task learning. *CoRR*, abs/1711.01239, 2017. URL http://arxiv.org/abs/1711.01239.

Clemens Rosenbaum, Ignacio Cases, Matthew Riemer, Atticus Geiger, Lauri Karttunen, Joshua D. Greene, Dan Jurafsky, and Christopher Potts. Dispatched routing networks. Technical Report Stanford AI Lab, NLP Group Tech Report 2019-1, Stanford University, 2019a.

Clemens Rosenbaum, Ignacio Cases, Matthew Riemer, and Tim Klinger. Routing networks and the challenges of modular and compositional computation. *CoRR*, abs/1904.12774, 2019b. URL http://arxiv.org/abs/1904.12774.

Sebastian Ruder, Joachim Bingel, Isabelle Augenstein, and Anders Søgaard. Sluice networks: Learning what to share between loosely related tasks. *arXiv preprint arXiv:1705.08142*, 2017.

Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.

Tim Shallice. *From neuropsychology to mental structure*. Cambridge University Press, 1988.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017. URL http://arxiv.org/abs/1701.06538.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Marijn F Stollenga, Jonathan Masci, Faustino Gomez, and Juergen Schmidhuber. Deep networks with internal selective attention through feedback connections. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3545–3553. Curran Associates, Inc., 2014.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

Philip S Thomas. Policy gradient coagent networks. In *Advances in Neural Information Processing Systems*, pages 1944–1952, 2011.

Philip S Thomas and Andrew G Barto. Conjugate markov decision processes. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 137–144, 2011.

G Tononi, O Sporns, and G M Edelman. A measure for brain complexity: relating functional segregation and integration in the nervous system. *Proceedings of the National Academy of Sciences*, 91(11):5033–5037, 1994. doi: 10.1073/pnas.91.11.5033. URL https://www.pnas.org/content/91/11/5033.

George Tucker, Andriy Mnih, Chris J. Maddison, and Jascha Sohl-Dickstein. REBAR: low-variance, unbiased gradient estimates for discrete latent variable models. *CoRR*, abs/1703.07370, 2017. URL http://arxiv.org/abs/1703.07370.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Oriol Vinyals, Charles Blundell, Timothy P. Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. *CoRR*, abs/1606.04080, 2016. URL http://arxiv.org/abs/1606.04080.

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018. doi: 10.18653/v1/N18-1101. URL `http://aclweb.org/anthology/N18-1101`.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. ISSN 0885-6125.

Ronald J Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

Yanze Wu, Qiang Sun, Jianqi Ma, Bin Li, Yanwei Fu, Yao Peng, and Xiangyang Xue. Question guided modular routing networks for visual question answering. *arXiv preprint arXiv:1904.08324*, 2019.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *CoRR*, abs/1611.03530, 2016. URL `http://arxiv.org/abs/1611.03530`.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *ICLR*, 2017.