DISPATCHED ROUTING NETWORKS

STANFORD AI LAB, NLP GROUP TECH REPORT 2019-1

Clemens Rosenbaum,[*1] **Ignacio Cases,**[*2]
Matthew Riemer,[3] **Atticus Geiger,**[2]
Lauri Karttunen,[2] **Joshua D. Greene,**[4]**, Dan Jurafsky,**[2]**, Christopher Potts,**[2]
[1]University of Massachusetts Amherst
[2]Stanford University
[3]IBM Research [4]Harvard University
cgbr@cs.umass.edu, cases@stanford.edu
[*] equal contribution

June 5, 2019

ABSTRACT

Routing and Recursive Routing Networks (RRNs) are highly expressive neural networks with modular self-assembling architectures that have proven successful in complex NLU tasks. However, this expressive power can at times be both a blessing and a curse. For many practical problems, vanilla RRNs tend to overfit to the limited data available. However, recent work has shown that high-quality meta-information can be extremely useful as a guide for routing in settings that require sample efficiency. Unfortunately, this meta-information is highly problem-dependent, and oftentimes it is not available at test-time. To compensate, we introduce an additional network that is trained jointly with the routing network to map samples to meta-information: a dispatcher. The dispatcher's goal is finding groups of samples that are as useful to the router as the groups defined by meta-information. We find that RRNs augmented with an end-to-end dispatcher achieve strong performance in multi-task learning scenarios while exhibiting high levels of generalization when adapting to new, unseen tasks.

1 Introduction

Core to human cognition are the lifelong abilities to learn new expressions from a small number of examples, adapt to new types of input, and generalize creatively. These rich and efficient learning skills exploit the capability to modularize, i.e., to decompose problems and solve them by re-composing elements from prior solutions [Lake et al., 2015]. Arguably, a key aspect of these fundamental abilities of modular learning is compositionality [Lake et al., 2015, 2016, Liang and Potts, 2015, inter alia], as it is evident in several cognitive tasks such as language understanding [Partee, 1984, Janssen, 1997]. Models that can learn to do inference and in particular natural language inference [MacCartney and Manning, 2009, Bowman et al., 2015, Bowman, 2016, Williams et al., 2018, inter alia] and at the same time are endowed with rich model-building mechanisms hold the promise to explain such extraordinary abilities [Lake et al., 2016].

Previous work has applied modular approaches to a variety of domains important to cognition, and particularly language, including composition of modules for question-answering [Andreas et al., 2015, 2016, Hu et al., 2017], language modeling [Kirsch et al., 2018], and natural language understanding[Cases et al., 2019].

In particular, we focus on routing networks [Rosenbaum et al., 2017], self-organizing networks with two components: a set of function blocks which can be applied to transform the input, and a router which makes decisions about which function block to apply next. While it has been shown that high-quality meta-information can be extremely useful for learning in scenarios that require sample efficiency [Cases et al., 2019], a common case in language understanding, this meta-information is highly problem-dependent, and oftentimes expensive to create. To compensate, we introduce an additional neural network that is end-to-end trained: a dispatcher. The dispatcher's task becomes to functionally replace the meta-information, i.e., to cluster the samples, so that the router can leverage this new new information to assign a matching path through the function modules of the routing network for each cluster. Using Recursive Routing Networks [Cases et al., 2019] (RRNs) augmented with an end-to-end dispatcher, we show strong performance in multi-task learning scenarios without explicit meta-information available, while exhibiting high levels of generalization when adapting to new, unseen tasks.

2 Dispatched Routing

Routing describes a general paradigm to compositional computation. Any routed model consists of one (or several) sets of parameterized functions or *modules*, and a decision making model that we call a *router*. Given an input, the router evaluates the input, and selects and applies a module from the according set. This yields a new activation, which will again be analyzed by the router, and again be transformed with a module, until the router decides to terminate and output the final activation. As such, every stacked architecture can be routed, in particular neural, layered architectures.

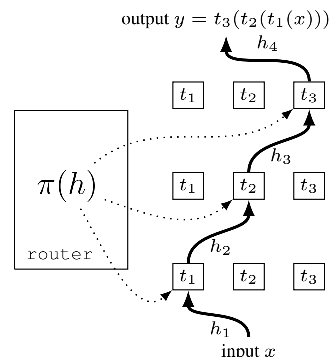


Figure 1: A *routing network*. The router consists of a parameterized decision maker that iteratively selects modules (i.e. trainable functions). Each selected function is then applied to the latest activation, resulting in a new activation, which can then again be transformed. The training of a routing 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.

This general idea of routing, selecting and applying different transformations in sequence, can be implemented in several ways. Apart from choosing the transformation within a model that will be routed, an important decision is the architecture of the router. Decision making functions such as the one utilized by the router are oftentimes called a *policy*, relating to their extensive investigation in reinforcement learning. As a policy can be encoded in many ways, from tabular lookup tables to complex function approximators, one of the most important questions when designing a routing network is how to encode that policy. The most intuitive approach is to encode the policy with one function approximator, one neural network as illustrated in see Figure 2(a), although constraints (in particular dimensionality constraints) may require multiple approximators in practice. However, different strategies are possible and have been introduced in Rosenbaum et al. [2017]. The first extension, depicted in Figure 2(b), consists of a different (sub-) policy for each decision in the routing network. The second introduces a hierarchy in the decision making process, by introducing a dispatcher policy (Figure 2(c)). This dispatcher assigns samples to different (sub-) policies, each of which could again be composed as e.g., a per-decision set of policies. Consequently, the subrouters responsible for all steps after dispatching can rely on much weaker information, not even

having access to the actual activation. Rosenbaum et al. [2017] show that both of these steps reduce the complexity of training the routing policy, a plausible claim considering their divide-and conquer nature.

While aiming to solve the same problem, a dispatched routing network can be characterized quite differently than a non-dispatched model. The difference lies in how to describe the assignment that routing does when choosing different routing paths. For networks with a single policy, this assignment happens in the space of activations. For a dispatched network, however, samples are *clustered* while still in input-space, before any future evaluation in activation space may happen. Rosenbaum et al. [2017], Cases et al. [2019] utilized this fact to assign samples based on task-labers, effectively using clusters given externally by the dataset. Such an approach could be seen as *offline* dispatching, as the data is clustered in a separate process before training a routing network, and is contrary to *online* dispatching, where the clusters are learned simultaneously with the routing network.

2.1 Advantages of dispatching

While a 'single policy' routing architecture, illustrated by Figure 2 (a), is conceptually simpler, dispatching has several advantages over this form of routing, which roughly fall into two different groups: the first concerns how dispatching addresses the challenges raised in Rosenbaum et al. [2019] (see below), and the second regards its interpretability and versatility.

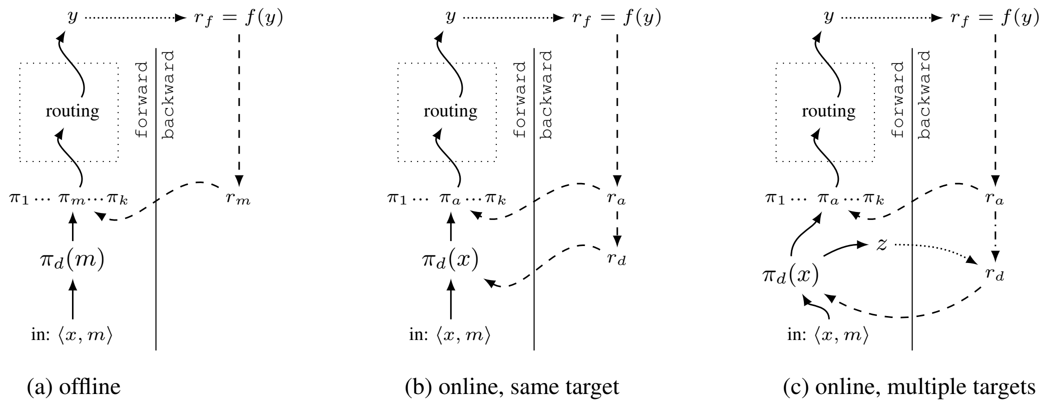
Rosenbaum et al. [2019] detail several challenges to all compositional architectures, and to routing in particular. The most important for dispatching are the extrema of the "flexibility dilemma for modular learning": collapse and overfitting. The flexibility dilemma describes problems arising from a modular architecture's ability to locally assign samples to different models, or paths in a routing network. For collapse, the modular architecture fails to find useful local approximations, and only uses a few (or even only one) coarse grouping of samples. For overfitting, it finds too many clusters, thereby possibly allowing a separate model for only a few (or even only one) samples. Balancing these is thus one of the most important challenges for designing any kind of modular architecture. However, if the right local approximations are found, routing can create highly flexible models that exactly capture the nature of the desired function, leading to impressive generalization properties.

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 2(c). If appropriately chosen, this regularization loss can prevent collapse. However, it can also prevent overfitting, as it may prevent the formation of very small clusters.

The second advantage of dispatching lies in assigning samples in input-space exclusively. 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. The fact that surface properties, and not activations, should decide on the clusters of a routing network, additionally make the routing process much easier, thereby stabilizing the learning process.

The third advantage is how dispatching can be combined with well-investigated strategies for clustering. As argued in the previous section, dispatching effectively clusters samples 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 [cf. 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. This means that we could cluster



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

Figure 2: 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.

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

2.2 Dispatching Strategies

We have 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 in the extension in Cases et al. [2019]), and the third is online dispatching with multiple objectives. We will investigate all three, where we will focus on using existing clustering algorithms for offline dispatching, as we want to move beyond relying on provided meta-information.

2.2.1 Offline Dispatching by Clustering

Offline dispatching relies on cluster-information available *before* the routing network is trained. The best investigated clusters are human annotated task-labels [Rosenbaum et al., 2017, Cases et al., 2019]. However, a natural extension are 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 the word sequences in the given samples, which will later be clustered. For Natural Language Inference (the domain we focus on in this work), previous work exclusively clustered samples based on the premise of the premise-hypothesis pairs, motivating us to do the same. 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. For this report, we consider two: K-Means clustering and Agglomerative Clustering.

2.2.2 Online Dispatching without an additional Objective Function

Both this kind of dispatching and the following one introduce a clear hierarchy in the online decision making process, as the dispatcher acts as an additional step in sample-space that assigns a sample to a secondary agent which will do the actual routing. In this case, the dispatcher is trained on the same objective as the other routing decisions – which might be final performance, or any other function of the final output of the routed model.

In contrast to Cases et al. [2019], where this kind of dispatching was shown to not generalize well, we experiment with a dispatcher that encodes the input with a sequential network, and not with a CBoW representation.

2.2.3 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. This can be achieved by optimizing the dispatcher, a reinforcement learning agent, on a weighted sum of rewards corresponding to these objectives:

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

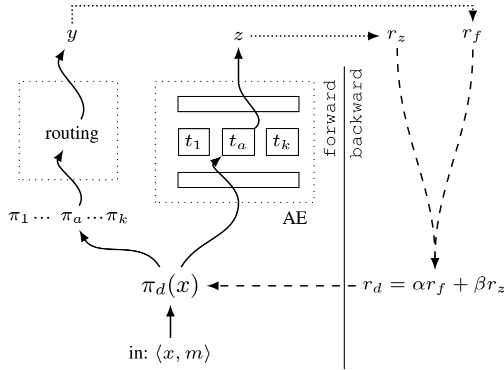


Figure 3: Autoencoder dispatching. The dispatcher chooses an action which determines both the sub-policy routing the sample and the autoencoder. The dispatcher is then trained on feedback from both the sub-policy routing and the autoencoder. The autoencoder may be replaced with any other single-step routing network, with any target z.

One very simple external objective is self-reconstruction. That is, we derive the dispatching action by routing a bottlenecked autoencoder on a sentence encoding, as depicted in Figure 4. This yields natural clusters that can be trained online, and that can be trained on an objective as a combination of the reconstruction reward and the routing reward. A natural reconstruction reward is the negative reconstruction loss, following a reward design introduced in Rosenbaum et al. [2017]:

$$\mathcal{R}_{ext} = -\mathcal{L}_{reconstruction} \tag{2}$$

Routing an autoencoder works by forcing the information contained in an encoding through a set of selectable “bottlenecks”. If these are small enough (in terms of parameter count), a router will select different transformations for different samples, thereby grouping these by reconstruction similarity. The routing component of the autoencoder can thus be defined as solving the following optimization problem (with M as the set of

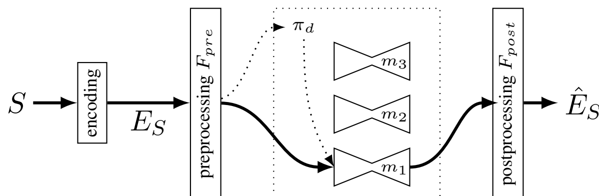


Figure 4: A (routed) bottleneck autoencoder. After the input S is encoded, the encoding E_S is projected to some dimensionality d_p, and then passed through one of a set of “bottleneck” modules, $\{m_1, m_2, m_3\}$, selected by the autoencoder dispatching policy π_d. These are of the form $fc_{d_p,d_{bn}} \rightarrow fc_{d_{bn},d_p}$, 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.

available modules, F_{pre} the encoding preprocessing, and F_{post} as the postprocessing, E the encoding, S the phrase and MSE as the mean-squared difference):

$$\underset{m \in M}{\text{argmax}} -MSE(E(S), F_{post}(m(F_{pre}(E(S))))) \tag{3}$$

However, there are several ways to design the autoencoder by choosing different encodings.

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):

$$Enc(S) = \sum_{w \in S} GloVe(w) \tag{4}$$

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):

$$Enc(S) = \sum_{w \in S} A(w) GloVe(w) \tag{5}$$

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):

$$Enc(S) = \sum_{w \in S} OneHot(w) \tag{6}$$

Sentence-level 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):

$$Enc(S) = SEQ(S) \tag{7}$$

3 Related Work

Routing Networks are an extension of early work on task-decomposition modular networks [Jacobs et al., 1991a] and mixtures of experts architectures [Jacobs et al., 1991b, Jordan and Jacobs, 1994] that make hard selections over modules in order to better navigate the transfer-interference trade-off of weight sharing during learning [Riemer et al., 2019]. Routing Networks provide a bridge between a body literature focused on adaptive weight sharing during multi-task learning [Stollenga et al., 2014, Rusu et al., 2016, Misra et al., 2016, Riemer et al., 2016, Aljundi et al., 2017, Fernando et al., 2017, Ruder et al., 2017, Rajendran et al., 2017] and a body of literature focused on neural architecture search [Zoph and Le, 2017, Baker et al., 2017, Miikkulainen et al., 2017, Liang et al., 2018]. In particular, Routing Networks provide a generalization of "one-shot" neural architecture search [Brock et al., 2017, Pham et al., 2018, Bender et al., 2018, Ramachandran and Le, 2019].

Our work is inspired by past work that has leveraged data clusters or task clusters based on natural regularities in the data as a means of guiding the transfer learning process [cf. Cases et al., 2019]. In particular our approach of using autoencoder to help control dispatching in routing networks is related to the work of Aljundi et al. [2017] who leveraged an autoencoder to make expert selection decisions during lifelong learning in application to a mixtures of experts architecture. The main conceptual difference in our work is that the autoencoder helps inform the decision of which routing sub-policy to select as opposed to helping select the modules themselves. Our work also shares similar motivation with past work that leveraged task and data clusters to improve multi-task learning [Kang et al., 2011, Kumar and Daume III, 2012, Crammer and Mansour, 2012, Barzilai and Crammer, 2015, Yang and Hospedales, 2016, Murugesan et al., 2017, Yu et al., 2017] and to improve meta-learning when no task labels are available [Hsu et al., 2019].

4 Experiments

In this section, we evaluate the proposed architectures on the Stanford Corpus of Implicatives (SCI) [Cases et al., 2019]. This corpus consists of approximately $11k$ triplets of premise, hypothesis pairs together with a label that indicates the entailment relation between the premise and hypothesis, modeled after other NLI coprus [Bowman et al., 2015, Williams et al., 2018]. We chose it as it comes in four different variations, each challenging a different aspect of generalization. For a complete description of the dataset and its properties, see Cases et al. [2019].

4.1 Quantitative Results

				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 1: Test results on SCI, averaged over five runs. 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.

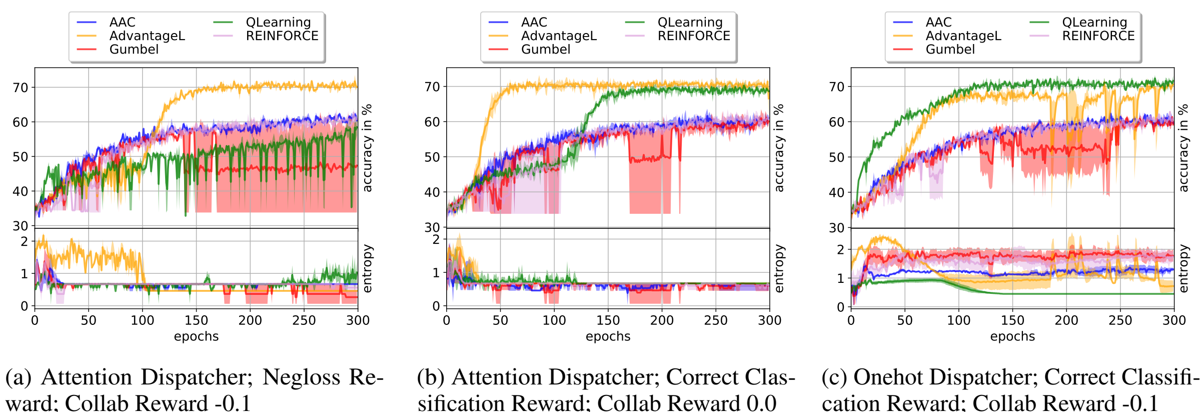
We begin by evaluating the performance on SCI over the different approaches to dispatching and the choice of router training algorithm, as these are the two most relevant design questions. To address the issues raised in Rosenbaum et al. [2019], we will not only evaluate the different architectures on test accuracy, but also on the dispatching selection entropy (to discuss the problem of *collapse*). See the appendix for the train accuracies to get a sense of the problem of *overfitting*. For space reasons, the full table of all results, including the train accuracy, is in the appendix (see Table 8), while a summary is shown in Table 1.

The baseline to these experiments are the results presented in Cases et al. [2019], i.e., the (unrouted) sequential model, the (unrouted) CBOW model and word-level CBOW routing, conditioned on signatures (the first three rows of Tables 1, 8). The architecture is the same architecture, i.e., routing the individual word projections of a CBOW encoding. The questions we try to answer 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?

In addition, we study the effect of the most important hyperparameter choices on the base dataset (joint) in Figures 5 through 6, depicting dev results over training over 300 epochs. We begin with these, as they determine the design choices for the results discussed later.

Design and Hyperparameter Choices Figure 5 shows the impact the choice of decision making algorithm has on the learning behavior, on three exemplary hyperparameter choices. These re-iterate the results from Rosenbaum et al. [2019] that value-based approaches to learning reliably outperform policy gradient based approaches, including the Gumbel reparameterization trick. 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 5(a), Q-Learning may outperform Advantage learning for specific combinations of hyperparameters, as for Figure 5(c).

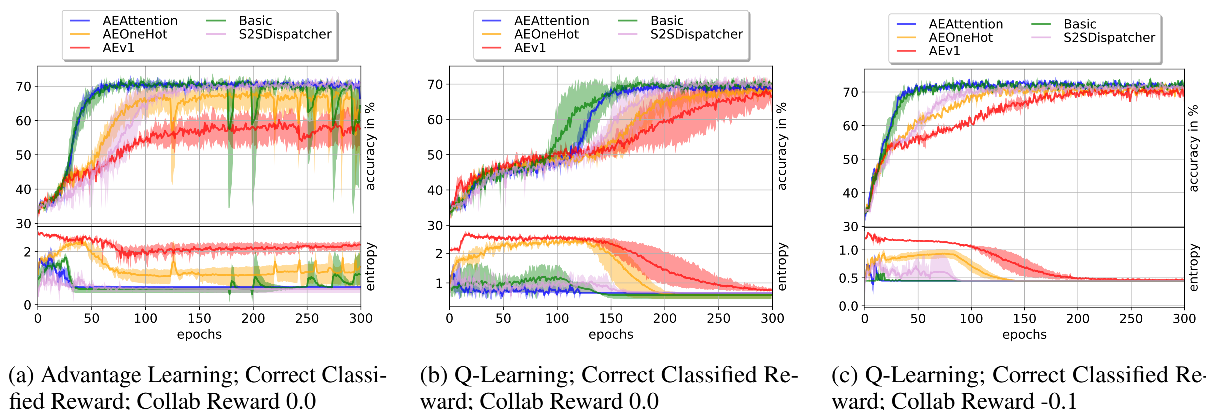


(a) Attention Dispatcher; Negloss Reward; Collab Reward -0.1

(b) Attention Dispatcher; Correct Classification Reward; Collab Reward 0.0

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

Figure 5: Comparison of decision making algorithms over different hyperparameter settings



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

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

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

Figure 6: Comparison of dispatching architectures over different hyperparameter settings

Another important design choice regards the design of the reward function used to train the router, and, consequently, also partially the dispatcher. As described in Rosenbaum et al. [2017, 2019], the quality of the routing decisions can be determined by not only the design of the final routing reward, but also by a 'collaboration' reward that regularizes diversity of the decision making within the router.

For the final routing reward, we evaluate two choices. The first is a "Correct classified reward" of $+1$ for a correct classification, and -1 for incorrect classification. The other is the negative of the classification loss (negative as we minimize losses, but maximize RL rewards). For the collaboration reward, which is computed as a fraction of the overall probability of choosing a module, we evaluate values of -0.1, incentivizing diversity as in Cases et al. [2019] and 0.0, i.e., no collaboration reward. We found that the generally best combination is using the correct classified reward with a small diversity reward of -0.1. A figure with exemplary dev curves is in the appendix, Figure 9.

Finally, Figure 6 shows the dev results of the different dispatching strategies over 300 training epochs. They, again, show that the respective learning behavior strongly depends on outside factors, as e.g., the plain autoencoder depicted in Figure 6(a) does not learn competetive predictions, while it does in the other cases. While we originally expected the one-hot autoencoder to learn the slowest, we found that the plain autoencoder-dispatcher can take noticeably longer to learn. Another important observation is that, generally, the sweet spot for entropy in regard to performance is somewhere smaller than one; if a dispatcher maintains a higher entropy, it does not achieve good performance.

Offline Dispatching We show results for offline dispatching, i.e., for clusters computed offline before training of the routing network in Table 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 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 even beat 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.

Online Dispatching Table 1 show the test scores of the different online dispatching algorithms after 300 epochs. They allow us to draw some clear and some not so clear conclusions:

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

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

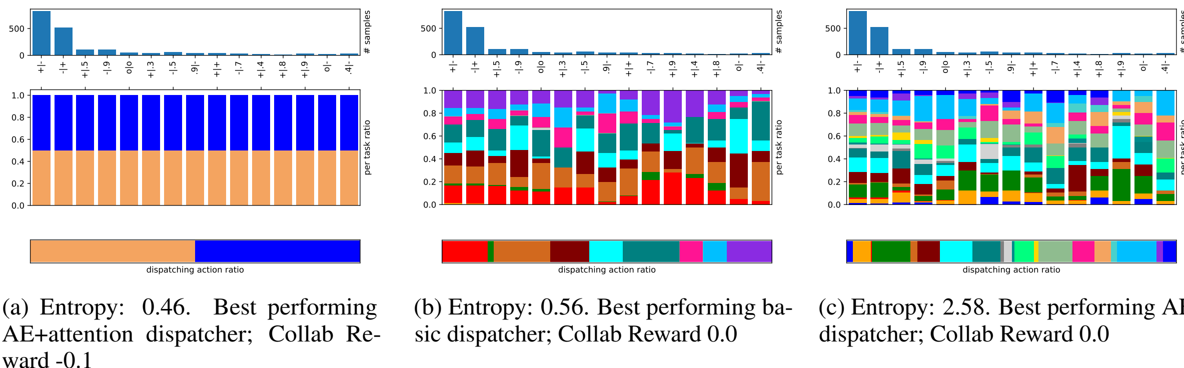
(3) With the exception of the autoencoder dispatching, most of the dispatching entropy results are in the $0.4 - 0.5$ range, which generally corresponds to two dominant dispatching clusters with minor deviations.

(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, as it allows the encoding to learn to ignore stop words, names and other words irrelevant to the implication question. However, it is surprising that the 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 our basic dispatcher does not use a BoW encoding, but a sequential encoding instead. Apparently, the different in expressivity between these two is sufficient to learn meaningful clusters.

(2) does not only hold for the accuracies, but generally also extends to the entropy results. Their confidence intervals are shown in the appendix. A particularly interesting result is the plain autoencoder result on joint

& match, as 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.



(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 7: 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.

(3) holds to some degree for all dispatching strategies, except for the plain autoencoder. However, it is clearly impacted by the collaboration reward value. Consider Figure 7 that depicts the dispatching distributions for three of the best performing models. Interestingly, a negative collaboration reward, which should theoretically increase entropy, causes a fairly even split between two dispatching clusters only. Only no collaboration reward will create naturally distributed behavior. These results generalize to all other experimental runs we did. Unfortunately, there appears to be only little correlation between signatures and learned dispatching clusters.

(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.

5 Conclusion

In this report, we evaluated several strategies to design a dispatcher, a hierarchical routing element that clusters samples into similarity groups that share a routing path. Considering the main challenges described in Rosenbaum et al. [2019], overfitting and collapse, we designed two main groups of dispatchers: a "plain" version that is purely trained on the routing rewards, and a version that is trained on an additional outside signal. We found that a natural signal is self-reconstruction in form of an autoencoder. This allows the network to both stabilize, and to learn naturally occurring clusters. Additionally, this external signal makes these clusters less prone to collapse, and additionally regularizes sufficiently to not overfit too badly.

We evaluated on the Stanford Corpus of Implicatives (SCI), as it comes in several variations, each of which testing a different way to generalize. Compared with the existing baselines published in Cases et al. [2019], we found that our proposed dispatched routing architectures mostly fall short of the performance of routing architectures relying on high-quality, hand-labeled clusters, but still considerably outperform their respective base architectures. In particular, for the 'nested' variation of SCI which composes different signatures, our dispatching architectures were able to learn impressively useful clusters, outperforming even hand-labeled clusters.

References

Rahaf Aljundi, Jay Chakravarty, and Tinne Tuytelaars. 2017. Expert gate: Lifelong learning with a network of experts. In *Proceedings CVPR 2017*, pages 3366–3375.

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

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. 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.

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

Aviad Barzilai and Koby Crammer. 2015. Convex multi-task learning by clustering. In *Artificial Intelligence and Statistics*, pages 65–73.

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

Samuel R. Bowman. 2016. *Modeling natural language semantics in learned representations*. Ph.D. thesis, Stanford University.

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. 2015. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 632–642. Association for Computational Linguistics.

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

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. 2019. 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)*.

Koby Crammer and Yishay Mansour. 2012. Learning multiple tasks using shared hypotheses. In *Advances in Neural Information Processing Systems*, pages 1475–1483.

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

Kyle Hsu, Sergey Levine, and Chelsea Finn. 2019. Unsupervised learning via meta-learning.

Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. 2017. Learning to reason: End-to-end module networks for visual question answering. *CoRR*, abs/1704.05526.

Robert A. Jacobs, Michael I. Jordan, and Andrew G. Barto. 1991a. Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. *Cognitive Science*, 15(2):219 – 250.

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

Theo M. V. Janssen. 1997. Compositionality. In Johan van Benthem and Alice ter Meulen, editors, *Handbook of Logic and Language*, pages 417–473. MIT Press and North-Holland, Cambridge, MA and Amsterdam.

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

Zhuoliang Kang, Kristen Grauman, and Fei Sha. 2011. Learning with whom to share in multi-task feature learning.

Louis Kirsch, Julius Kunze, and David Barber. 2018. Modular networks: Learning to decompose neural computation. *arXiv preprint arXiv:1811.05249*.

Abhishek Kumar and Hal Daume III. 2012. Learning task grouping and overlap in multi-task learning. *arXiv preprint arXiv:1206.6417*.

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

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. 2016. Building machines that learn and think like people. *CoRR*, abs/1604.00289.

Jason Zhi Liang, Elliot Meyerson, and Risto Miikkulainen. 2018. Evolutionary architecture search for deep multitask networks. *CoRR*, abs/1803.03745.

Percy Liang and Christopher Potts. 2015. Bringing machine learning and compositional semantics together. *Annual Review of Linguistics*, 1(1):355–376.

Bill MacCartney and Christopher D. Manning. 2009. An extended model of natural logic. In *The 8th International Conference on Computational Semantics (IWCS-8)*, pages 140–156. University of Tilburg, Tilburg, Netherlands.

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

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

Keerthiram Murugesan, Jaime Carbonell, and Yiming Yang. 2017. Co-clustering for multitask learning. *arXiv preprint arXiv:1703.00994*.

Barbara H. Partee. 1984. Compositionality. In Fred Landman and Frank Veltman, editors, *Varieties of Formal Semantics*, pages 281–311. Foris, Dordrecht.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

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

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

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

Prajit Ramachandran and Quoc V. Le. 2019. Diversity and depth in per-example routing models. In *International Conference on Learning Representations*.

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

Matthew Riemer, Aditya Vempaty, Flavio Calmon, Fenno Heath, Richard Hull, and Elham Khabiri. 2016. Correcting forecasts with multifactor neural attention. In *International Conference on Machine Learning*, pages 3010–3019.

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

Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. 2017. Routing networks: Adaptive selection of non-linear functions for multi-task learning. *arXiv preprint arXiv:1711.01239*.

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

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive neural networks. *arXiv preprint arXiv:1606.04671*.

Marijn F Stollenga, Jonathan Masci, Faustino Gomez, and Juergen Schmidhuber. 2014. 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.

Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. 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.

Yongxin Yang and Timothy Hospedales. 2016. Deep multi-task representation learning: A tensor factorisation approach. *arXiv preprint arXiv:1605.06391*.

Mo Yu, Xiaoxiao Guo, Jinfeng Yi, Shiyu Chang, Saloni Potdar, Gerald Tesauro, Haoyu Wang, and Bowen Zhou. 2017. Robust task clustering for deep many-task learning. *arXiv preprint arXiv:1708.07918*.

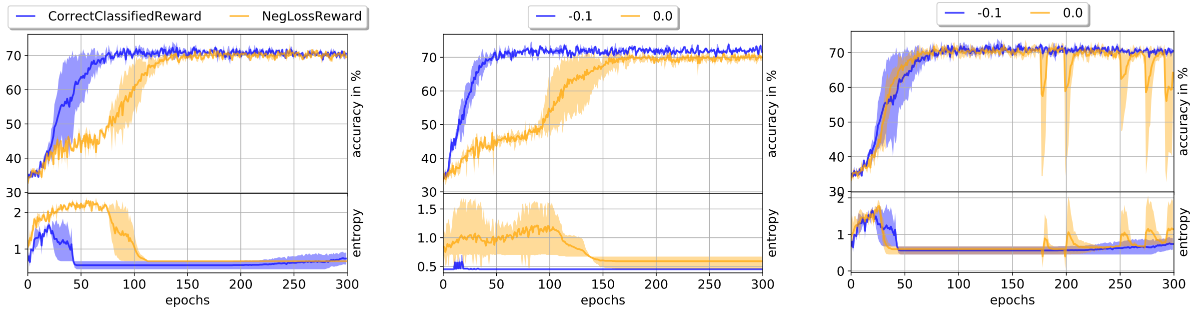
Barret Zoph and Quoc V Le. 2017. Neural architecture search with reinforcement learning. *ICLR*.

A Appendices

A.1 Full Results

Enc	Dispatching		class	Joint & Match			Disjoint			Mismatch			Nested		
				Test Acc	Train Acc	Entropy	Test Acc	Train Acc	Entropy	Test Acc	Train Acc	Entropy	Test Acc	Train Acc	Entropy
Seq	n/a	n/a	1	67.04±2.36		0	63.52±2.00		0	59.32±2.87		0	57.69±2.75		0
CBOW	n/a	n/a	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±0.0	73.91±0.54		1.8±0.0	71.08±0.52		1.82±0.0	75.43±0.29		0.46±0.0
		K-Means	5	62.26±2.79	98.22±0.01	0.45±0.04	61.68±1.4	98.07±0.33	0.35±0.02	59.29±0.59	97.86±0.32	0.42±0.05	71.67±4.03	97.57±0.23	0.39±0.0
		Aggl	5	64.63±0.62	97.62±0.36	0.44±0.04	60.86±0.75	98.05±0.32	0.46±0.06	61.56±0.55	97.68±0.29	0.43±0.04	72.45±1.79	97.45±0.33	0.59±0.0
	Online	Basic	≤ 20	71.4±1.39	96.29±0.58	0.56±0.11	64.13±0.3	97.68±0.29	0.45±0.0	67.77±0.61	97.76±0.19	0.45±0.0	83.24±0.58	95.65±0.38	0.62±0.0
		AE	≤ 20	70.42±0.83	74.44±3.33	2.58±0.01	61.31±0.97	95.59±1.07	0.84±0.14	63.08±0.94	96.2±0.7	0.59±0.13	79.38±0.99	95.94±0.99	0.31±0.2
		AE Attention	≤ 20	72.7±0.21	95.24±0.05	0.46±0.01	65.05±0.78	97.78±0.21	0.45±0.0	67.74±0.23	97.77±0.25	0.44±0.01	80.35±1.42	97.47±0.22	0.41±0.0
		AE OneHot	≤ 20	70.89±0.98	93.76±0.8	0.71±0.18	61.94±1.12	96.9±0.55	0.49±0.06	64.01±0.53	95.31±0.99	0.68±0.1	79.77±2.87	96.74±0.59	0.4±0.3
		AE Sequential	≤ 20	70.49±0.7	96.81±0.3	0.45±0.0	63.3±1.12	95.87±0.53	0.55±0.11	65.43±0.53	97.48±0.22	0.45±0.0	82.37±2.6	97.57±0.01	0.41±0.0

Figure 8: Results on SCI. Each entry consists of the test accuracy with confidence intervals, the train accuracy with confidence intervals, and the entropy of the learned dispatching policy at test time, also with confidence intervals. All models are trained with an Advantage learning router.



(a) Basic Dispatcher; Advantage Learning; Collab Reward -0.1

(b) Basic Dispatcher; Q-Learning; Correct Classified Reward

(c) Basic Dispatcher; Advantage Learning; Correct Classified Reward

Figure 9: Comparison of reward type (subfig a) and collaboration reward values (subfigs b, c) over different hyperparameter settings