Systematic Generalization in Connectionist Models

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by

Róbert Csordás

under the supervision of
Prof. Jürgen Schmidhuber

September 2023

Dissertation Committee

Prof. Cesare Alippi	Università della Svizzera italiana, Switzerland
Prof. Rolf Krause	Università della Svizzera italiana, Switzerland
Prof. Dzmitry Bahdanau	McGill University/MILA/ServiceNow, Canada
Prof. Jacob Andreas	Massachusetts Institute of Technology, USA
Prof. Marco Baroni	Pompeu Fabra University, Spain

Dissertation accepted on 18 September 2023

Research Advisor	PhD Program Director
Prof. Jürgen Schmidhuber	**Prof. Walter Binder/ Prof. Stefan Wolf**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Róbert Csordás
Lugano, 18 September 2023

Abstract

In recent years, neural networks (NNs) revolutionized computer science, solving many problems out of reach of classical methods. Thanks to their flexibility, they can process raw data, such as images, audio, or text, and defeat humans in games. However, a critical challenge remains: they often fail on test data that follow the same underlying rules as the training data but present superficial differences, like longer inputs or unseen word combinations. Generalization to such structurally related data is called systematic generalization. Analysis suggests that NNs often learn a smart interpolation between their training data points and rarely learn a generally applicable rule-based solution. This limits both their applicability and their trustworthiness. Thus, systematic generalization is of utmost importance. This work consists of multiple parts. First, we improve the performance of differentiable neural computers in algorithmic and reasoning tasks. Then we analyze the implicit modularity of neural networks and show that it does not support compositionality. Motivated by compositionality, we introduce architectural changes to transformers, significantly boosting generalization on multiple well-known datasets. Pushing this idea further, we introduce the purely connectionist NDR architecture that can generalize to longer inputs on algorithmic tasks. Then we move our focus to systematicity, and we propose a new dataset to analyze the behavior of the model. Finally, we focus on scaling up NDRs to real-world tasks and improving the Mixture of Experts models, matching the performance of the parameter-equivalent dense baselines. We hope that the high-level ideas outlined in this thesis can provide guidance for further research aiming to achieve compositional generalization.

Acknowledgements

I am deeply grateful to my supervisor, Jürgen Schmidhuber, for giving me the opportunity to pursue my PhD in his lab at IDSIA. He encouraged me to explore my own research ideas, which was crucial to the direction of this PhD thesis. He also provided financial support for my studies and the hardware we used in our lab. Our discussions deeply influenced my view about neural networks as general-purpose computers that should execute algorithms.

I would also like to thank my loving wife Dorottya Cserpán for all the support she gave me during the difficult parts of this journey. I am also grateful for her patience for the long weekends I sometimes had to spend working on different projects. I am thankful to my parents, Mária and István, for supporting my passion for science and supporting me in my childhood despite their difficult financial situation.

If I have to nominate a single person who impacted me the most during my PhD, then this is Kazuki Irie. His way of approach to research, managing, and executing projects had a deep influence on me. His support was very valuable to me. His positive attitude is very inspiring and motivating.

I am grateful for all my colleagues, Aleksandar Stanic, Francesco Faccio, Imanol Schlag, Vincent Hermann, Anand Gopalakrishnan, Sjoerd van Steenkiste, Louis Kirsch, Mikhail Andronov, Aditya Ramesh, Quinhan Hou, and Dylan Ashley, for all the wonderful time spent together. We not only engaged in scientific discussions and learned from each other, but also had a lot of fun hiking, playing board games, and traveling. I would also like to emphasize our discussions with Imanol Schlag, who not only has a research interest aligned with mine, but also helped me maintaining the hardware infrastructure of the lab.

I would like to thank my hosts at DeepMind, Jordan Hofman and Michela Paganini for having me as their intern. I would also like to thank my collaborators, especially Petar Veličković, Andrea Banino, Grégoire Delétang and Anian Ruoss.

Finally, I would like to thank my friend Kis-Benedek Ágnes for the initial push that initiated this wonderful journey.

Contents

Contents **v**

1 Introduction **1**

1.1 Systematic Generalization 2

1.1.1 Characterizing Systematic Generalization 3

1.2 The Difficulty 5

1.3 Learning More Algorithmically 5

1.3.1 The Memory Architecture of Modern Neural Networks . 6

1.3.2 The Need for Sequential Computation 9

1.3.3 Breaking Down the Problem in Subproblems 11

1.4 Systematic Generalization in the Age of Large Language Models 12

1.5 Related Concepts 13

1.6 Related Work Overview 17

2 Improving Differentiable Neural Computers **20**

2.1 Brief Overview of DNC 21

2.2 Method 23

2.3 Experiments 25

2.3.1 The Effect of Modifications 26

2.3.2 bAbI Experiments 28

2.4 Conclusion 30

3 Inspecting the Implicit Modularity of Neural Networks **33**

3.1 Discovering Modules via Weight-Level Introspection 35

3.2 Analyzing Fundamental Properties of Modules 37

3.2.1 Addition/Multiplication Experiments 38

3.2.2 Double-Addition Experiments 39

3.2.3 Transfer Learning Experiments 42

3.2.4 A Potential Explanation for Lack of Weight Sharing . . . 43

3.3 Analyzing Systematic Generalization on Algorithmic Tasks . . . 44
3.4 Analyzing Convolutional Neural Networks 46
3.5 Related Work 47
3.6 Conclusion 48

4 Improving the Systematic Generalization of Transformers 49
4.1 Datasets and Model Architectures for Systematic Generalization 50
4.1.1 Datasets 50
4.1.2 Model Architectures 52
4.2 Improving Transformers on Systematic Generalization 52
4.2.1 Addressing the EOS Decision Problem with Relative Positional Embedding 53
4.2.2 Model Selection Should Be Done Carefully 55
4.2.3 Large Impacts of Embedding Scaling 59
4.3 Results Across Different Datasets 60
4.3.1 SCAN 60
4.3.2 CFQ 60
4.3.3 PCFG 62
4.3.4 COGS 62
4.3.5 Mathematics Dataset 63
4.4 Related Work 63
4.5 Conclusion 64

5 Achieving Length Generalization with Transformers 65
5.1 Improving Transformers for Learning Adaptive Control Flow . . 66
5.1.1 Copy Gate: Learning to Skip Operations (Vertical Flow) 67
5.1.2 Geometric Attention: Learning to Attend to the Closest Match (Horizontal Flow) 68
5.2 Experiments 70
5.2.1 Compositional Table Lookup 71
5.2.2 Simple Arithmetic 74
5.2.3 ListOps 75
5.3 Analysis 76
5.4 Discussion 79
5.5 Related Work 79
5.6 Conclusion 81

6 Inspecting Systematicity of Neural Networks 82

6.1 Original CTL 83

6.2 Extensions for Systematicity: CTL++ 83

6.3 Results 86

6.3.1 Results on Variants 'A' and 'R' 86

6.3.2 Results of Staged Variant 'S' 88

6.4 Limitations 89

6.5 Conclusion 89

7 Accelerating Transformer MLP Layers: a Path Towards Scalable NDRs 90

7.1 Background 92

7.2 Approximating 2-layer MLPs 93

7.2.1 Top-K Activation Function 94

7.2.2 Product-Key Memories (PKMs) 94

7.2.3 Mixture of Experts (MoE) 95

7.3 Existing MoE Variants 96

7.4 Improving Mixture of Experts 99

7.5 Experiments 101

7.5.1 Top-K 101

7.5.2 Product-Key Memory (PKM) 102

7.5.3 Mixture of Experts (MoE) 103

7.6 Limitations 106

7.7 Conclusion 108

8 Conclusion and Future Work 109

8.1 Future Directions 110

A Further Details on Improving Differentiable Neural Computers 111

A.1 Implementation Details 111

A.2 Hyperparameters for the Experiments 114

B Additional Details for Inspecting the Implicit Modularity of Neural Networks 115

B.1 Derivations 115

B.1.1 From Gumbel-Softmax to Gumbel-Sigmoid 115

B.1.2 Straight-Through Estimator 116

B.1.3 The Expected Value of the Samples 117

B.1.4 Choosing the Temperature 118

B.2 Additional Discussion 119

B.2.1 Stability of the Masks 119
B.2.2 Does Masking Change the Performed Operation? 119
B.2.3 Choosing Target Functionality 121
B.2.4 Is Attention the Solution? 122
B.2.5 Explicitly Modular Networks 123
B.3 Additional Results and Experimental Details 124
B.3.1 Sanity Checking the Mask Discovery Process 124
B.3.2 Common Hyperparameter Choices 124
B.3.3 Choosing the Regularization Hyperparameter 125
B.3.4 Addition/Multiplication Experiments 126
B.3.5 Double Addition Experiments 127
B.3.6 Transfer Learning Experiments 130
B.3.7 Experiments on Algorithmic Tasks 130
B.3.8 CNN Experiments on CIFAR10 133

C Additional Details on Improving the Systematic Generalization of Transformers 142
C.1 Evaluation Metrics 142
C.2 Hyperparameters 142
C.3 Relative Positional Embedding 143
C.4 Embedding Scaling 143
C.5 Analyzing the Positively Correlated Loss and Accuracy 144
C.6 Additional Results 144

D Additional Details for Achieving Length Generalization with Transformers 153
D.1 Ablations 153
D.2 Details of Attention with Combined Absolute/Relative Positional Encoding 156
D.3 Implementation Details 157
D.3.1 Choosing the number of layers 158
D.3.2 Dataset Details 158
D.3.3 Model Details 159
D.4 Additional Analysis 161
D.4.1 Compositional Table Lookup 161
D.4.2 ListOps 161

E More Details on Inspecting Systematicity of Neural Networks 175
E.1 Experimental Details 175
E.1.1 Modified NDR architecture 175
E.1.2 Hyperparameters 176
E.2 More Analyses and Plots 177
E.2.1 Quantitative Analysis of Incompatibility 177
E.2.2 Representative Cosine Similarities 177

F Additional Details of Accelerating Transformer MLP Layers: a Path Towards Scalable NDRs 181
F.1 Further Details and Analyses 181
F.1.1 Definition of normalised Top-K 181
F.1.2 Measuring the Number of Active Channels in 181
F.1.3 More Details and Results on PKM 182
F.1.4 Further Analyses of Our σ-MoE 183
F.1.5 More on Resource Efficiency 184
F.2 Implementation Details 184
F.2.1 A Few Words on the CUDA Kernel 185
F.2.2 Additional Results on MoEs 186

Bibliography 192

Chapter 1

Introduction

Artificial Intelligence (AI) has seen enormous progress in recent years, thanks to deep learning and artificial neural networks (NNs) [Schmidhuber, 2015, 2022]. More and more cognitively intensive tasks can be automated, such as vision [Ciresan et al., 2012; Krizhevsky et al., 2012; Srivastava et al., 2015b,a; He et al., 2016], language processing [Hochreiter and Schmidhuber, 1997; Bahdanau et al., 2015; Vaswani et al., 2017; Brown et al., 2020; OpenAI, 2022, 2023] or playing games [Mnih et al., 2013; Silver et al., 2016b]. NNs also proved to be a useful tool to accelerate scientific discoveries [Jumper et al., 2021; Degrave et al., 2022].

Despite their recent success, NNs still suffer from significant limitations. One of the most important is the lack of systematic generalization [Fodor et al., 1988]. *Systematic generalization* is the ability to generalize to novel data outside the training distribution, but governed by the same underlying rules. The difference between the working and failing cases is usually superficial, such as different input lengths or a different combination of known ingredients. The generalization capabilities of humans are widely assumed to be based on composition: we can recombine knowledge in novel ways, solving previously unseen tasks. *Compositional generalization* is sometimes used as a synonym for systematic generalization. The lack of systematic generalization has many implications: to cover the possible input space with sufficient density, a very large number of training examples are needed. Even in simple cases of composing unary functions, the number of training examples has to grow exponentially with the number of compositions. This is unrealistic in real-world situations. The deployed models cannot be trusted to perform well because the network might fail on any unseen composition. Although this can be greatly offset by a large amount of training data [OpenAI, 2022, 2023], NNs are generally weak in reasoning tasks, which usually require long and diverse recombinations of elementary operations. Current language models excel in sophisticated explanations of factual knowledge but often fail even on the simplest arithmetic operations [Rae et al., 2022; Dziri et al., 2023].

In this PhD thesis, we will take an algorithmic approach: we aim to make it easier for neural architectures to represent a general-purpose computer that is able to learn easily. This intuition leads to significant improvements in the generalization performance of the models. In this chapter, we briefly overview systematic generalization and introduce the main intuitions that will guide us throughout all of our work. In the following chapters, we present six different papers related to the topic. We start by improving the Differentiable Neural Computer [Graves et al., 2016] in Chapter 2. Then, in Chapter 3, we analyze whether the usual neural architectures develop an implicitly modular behavior, which seems to be necessary for compositionality. After concluding with a negative answer, we focus on the most promising architecture, the transformer [Schmidhuber, 1991, 1992c, 1993; Vaswani et al., 2017], and improve its generalization ability in Chapter 4. However, it turns out that even this is not enough to achieve length generalization on algorithmic tasks. Thus, in Chapter 5 we introduce the Neural Data Router (NDR) architecture that is able to generalize in ListOps for the first time, to the best of our knowledge. However, recombining operations in novel ways (systematicity) is still elusive, so in Chapter 6 we propose a new dataset to test the systematicity of the models, and conclude that the current ones, including the NDR, fail miserably. Finally, we focus on solving the main bottleneck that prevents NDRs from large-scale training: the execution time of shared layer transformers is very slow due to the increase d_{model} or d_{ff} if the parameter count is matched. As a first step towards this, in Chapter 7 we demonstrate that, with some modifications, the MoE models can match the performance of their parameter-matched dense counterparts.

1.1 Systematic Generalization

A historic debate began in the 1980s about the ability of connectionist models (neural networks) to model higher-level human cognition appropriately. The argument revolves around the structural symmetry of human thought: understanding certain sentences seems to be inherently related to understanding sentences with similar structures. For example, everyone who understands the sentence ″John loves Mary″ also understands ″Mary loves John″ [McLaughlin, 2009]. Standard NNs with standard objectives (e.g next-word prediction) do not have a built-in structure to support these kinds of symmetrical relations. It is unclear how they can be learned from data without seeing an exhaustive combination of elementary constituents in combination (e.g seeing both John and Mary as both subject and object); thus, neural networks are not systematic. Fodor et al. [1988] argued that this is a fundamental issue of connectionist systems, and thus they cannot be considered a good model of cognition. Instead, Fodor was defending his Language Of Thought hypothesis [Fodor, 1975], claiming that human thought is based on a language-like structure that has grammar and can be processed systematically by some rule-based system. The core components of this symbol processing system are innate [Chomsky, 1962]. Pylyshyn [1984] argues that such a computation is a fundamental property of the mind.

The debate is still far from settled [McLaughlin, 2009; Lake and Baroni, 2018; Hupkes et al., 2020]. Some researchers claim that the thought is not actually systematic [Wason, 1968; Johnson-Laird et al., 1972], others improve the systematicity of neural networks [Korrel et al., 2019; Li et al., 2019; Russin et al., 2019; Gordon et al., 2020; Herzig and Berant, 2020; Andreas, 2020; Herzig et al., 2021; Csordás et al., 2021, 2022a]. The original argument only took into account simple neural networks without additional structure. Built-in structures, such as the one in Fast Weight Programmers [Schmidhuber, 1991; Schlag and Schmidhuber, 2018; Schlag et al., 2019] and attention [Schmidhuber, 1991, 1992c, 1993; Bahdanau et al., 2015] could dramatically change the picture. Fodor et al. [1988] seems to ignore the possibility of improving the models or the optimization process in a way sufficient to solve the problems present in the NNs of the time.

We believe that systematicity is a spectrum: both human minds and the current neural models are systematic to a certain extent, as they generalize in certain cases, while not in others. It also seems to be obvious that, as for now, humans are much more systematic than artificial neural networks. The goal of our research should be to reduce the gap between the two.

1.1.1 Characterizing Systematic Generalization

There is no consensus in the community on formalizing systematic generalization. It is often studied in the context of language-based problems, which avoids the additional complexity of parsing the raw input signal (such as images) into more symbolic entities. In practice, the generalization ability of the models is tested by generating test splits that differ systematically from the train set. For example, they can have longer sequences or novel combinations of input symbols.

Despite the lack of consensus, the properties associated with systematic generalization are studied [Fodor et al., 1988; Fodor and McLaughlin, 1990; Pagin and Westerståhl, 2010; Hupkes et al., 2020]. Some of the most important such properties are the following:

1. **Systematicity**. Recombining known constituents in novel ways. For example, if the model can perform arithmetic expressions in the form of $(a + b) * c$, it should also be able to do $(a * b) + c$. It requires zero-shot generalization to unseen combinations of the elementary building blocks of the problem.

2. **Productivity**. Generalizing to arbitrarily large instances of a problem of the same type as seen during the training. The only limitation is the available memory. Generalization to longer sequences (length generalization) is a special form of this. For example, if the model learned to add up to five numbers, it should also work for six. Productivity does not necessarily require unseen combinations of elementary operations but could also just need longer chains of operations of the same type to be performed.

3. **Substitutivity**. Ability to handle synonyms.

4. **Localism**. Many problems are hierarchical: their general solution relies on solving subproblems and combining their results. Localism states that the solution of the subproblems should not depend on the global structure of the problem, but should be solved locally, independent of the global context. For example, to calculate $a + b$ in $(a + b) * c$, the addition operation should not be influenced by what is outside the parentheses. Localism is a controversial property that does not necessarily hold for natural language, but it is helpful to investigate it in other contexts, such as synthetic languages.

In realistic settings, such as natural language, a model that generalizes well can find it challenging to handle exceptions. This property is called **overgeneralization**, and it can be a useful indicator of the generalization ability of the models.

Systematic generalization is far from being language-exclusive: Vision models often fail to generalize to different colors, clothes, hairstyles, number of objects, etc. [Lake et al., 2017] and they are sensitive to adversarial attacks [Szegedy et al., 2014]. Reinforcement Learning (RL) agents often fail catastrophically due to trivial differences, such as changing colors [Kirk et al., 2021].

1.2 The Difficulty

The problem of generalization is challenging because there is no direct optimization pressure to generalize[1]. There is no information in the training set that guides what behavior we expect from the model outside the training data distribution. All solutions found by the optimization process are equally good, even pure memorization. Regularization and capacity bottlenecks can improve on this, but in a way that is indirect and hard to control. There could be multiple ways around this: the most obvious is the addition of inductive biases. Alternatives might include introducing splits of multiple difficulties to guide optimization (for example, by meta-learning [Schmidhuber, 1987]), using data augmentation methods to generate new compositional training data [Andreas, 2020; Akyürek and Andreas, 2022].

Adding inductive biases should be done with caution to keep the model's flexibility because, in real-world scenarios, the NN should be able to handle exceptions, which rely on memorization. Thus, the model should be biased, but not restricted to learning generally applicable rules instead of pure pattern matching.

In addition, there could be multiple issues that prevent generalization in current models. If only *some*, but not all, are fixed, the model can always fall back to memorization. Therefore, it is challenging to measure progress. There may be no progress made on real-world datasets for a long time [Furrer et al., 2020; Dankers et al., 2021], until all issues are solved. This might make the research field less attractive and more difficult to justify.

We propose approaching the problem iteratively: first, build some intuition about what might be missing for the real-world tasks. Then distill it to a synthetic dataset isolating the most essential problems. We can measure progress on such a dataset, and new models could be developed. It is important to always keep the end goal in mind and not commit to dataset-specific models. Finally, test whether the improvements transfer to more realistic tasks, and if not, return to the first step.

1.3 Learning More Algorithmically

In this chapter, we would like to informally build up the intuition that guides our research. Our research goal is to enable learning in purely neural architectures

[1] Assuming a standard setup; we do not exclude the possibility of construction of such an optimizer or objective.

in an algorithmic way, which we believe will lead to a more systematic behavior. Specifically, we would prefer that the NNs pick up algorithmic patterns whenever possible. For example, if trained on scientific papers [Lewkowycz et al., 2022], the calculations in them follow clear arithmetic rules, which should ideally be learned by the network. On the other hand, we would not want to impede the network's memorization and pattern-matching ability either, but we would like to achieve a fuzzy mixture of the two. We believe that this is a powerful combination that can adhere to rules while handling exceptions gracefully. We think a way of achieving this is to make it easier for the NNs to learn proper algorithms than to memorize the exponentially many combinations of inputs necessary to cover the input space sufficiently densely.

Our main hypothesis is that neural architectures capable of algorithmic processing must be able to represent general-purpose computers[2]. This aligns with the classical view of the theory of computation [Hopcroft and Ullman, 1979]. Note that Turing completeness of the architecture does not mean that such behavior is learnable by gradient descent in practice. Additional changes to the architecture or training method might be necessary to encourage learning algorithmic solutions instead of pure pattern matching. For example, most RNNs [Elman, 1990; Hochreiter and Schmidhuber, 1997] with the right parameterization can simulate a general-purpose computer with bounded memory[3], but when it comes to learning, they often fail. For example, RNNs are easily outperformed in practice by more complex architectures on reasoning tasks [Graves et al., 2014, 2016]. We will see in Sec. 4 and 5 that adhering to these principles can indeed improve the generalization capabilities of our neural models. In the following, we will discuss what we believe are the main architectural bottlenecks in current-day neural networks that prevent them from learning more algorithmically. To facilitate understanding, we will first discuss some of them in detail: the memory architecture and the lack of recurrence.

1.3.1 The Memory Architecture of Modern Neural Networks

In order to justify our design decisions, it is helpful to consider the memory architecture of modern neural networks in detail. In this work, we care only

[2] In practice, we only care about the bounded memory case, which corresponds to an FSA. However, we would like to think in higher lever abstractions, like loops, recursion, conditional execution, etc. Therefore, it is more useful to think of it as a Turing machine with bounded memory, just as the practically realizable hardware computers are.

[3] With infinite numerical precision, RNNs with continuous state [Elman, 1990] are known to be Turing-complete [Siegelmann and Sontag, 1991, 1995].

about practical models with a bounded amount of memory. When we discuss productivity, we only want to generalize to the point where the available amount of memory is still not the bottleneck. In this section, we will discuss the difference between RNNs [Lenz, 1920; Ising, 1924; Amari, 1972; Kohonen, 1972; Elman, 1990], memory-augmented NNs (typically RNNs with external memory banks), and transformers. In the case of transformers, we think of the layers as individual steps of an RNN, which is true for the shared layer case. Note that in general, there are multiple variables (″memories″) that the network must read, write, and make decisions on, and the number of memories might differ between train and test time (e.g consider a learned stack for parsing context-free grammars). Therefore, having a learning-friendly memory architecture is crucial. In the following, we cover 3 important aspects.

How are the Different Memories Stored and Accessed?

The simplest form of memory is the state of an RNN, which is a single vector. However, this vector can be arbitrarily big; thus, in theory, it is possible to store and retrieve an arbitrary number of memories in this single vector. However, it is unclear how the NN controller can learn a mechanism that can generalize to store more memories than was seen during training. The common assumption is that making the state large enough should help. We would like to challenge this assumption. If we assume that the network stores all stored variables in a different linear subspace of the state vector[4] we run into a problem: accessing a ″new″ linear subspace of the state vector (that is orthogonal to what is used during training) in test time seems difficult. To see why, consider the dual formulation of the layer [Irie et al., 2022] that receives such a vector as input. Since this new input is orthogonal to every input seen during training, no update made by gradient descent has an effect on it (see Eq. 11 by Irie et al. [2022]), and the layer will appear as ″untrained″ for this specific input. The existence of an alternative solution cannot be excluded (e.g. storing variables in some non-linear fashion), but it seems to be difficult to learn (our preliminary experiments confirm this). The superior performance of NNs with external memory [Graves et al., 2014; Joulin and Mikolov, 2015; Graves et al., 2016; Schlag and Schmidhuber, 2018; Suzgun et al., 2019] supports this view.

Memory-augmented NNs overcome this issue by having an explicit mechanism built in to manage the repeated structure of their memories. This guar-

[4] There is an increasing amount of evidence that NNs prefer to store features in linearly separable subspaces. See the probing literature, e.g. Alain and Bengio [2017]; Hupkes et al. [2018]; Lakretz et al. [2019].

antees that an arbitrary number of cells can be stored and retrieved (given a specific limit set explicitly and memory constraints). It also guarantees that all retrieved memories are in the same format and that no additional transformation is applied to them. Transformers behave in a similar manner: Each column of a transformer maintains its own independent state, thus acting like a memory in which each cell is transformed at each step.

How Easy is it to Make Decisions Based on the Memory Content?

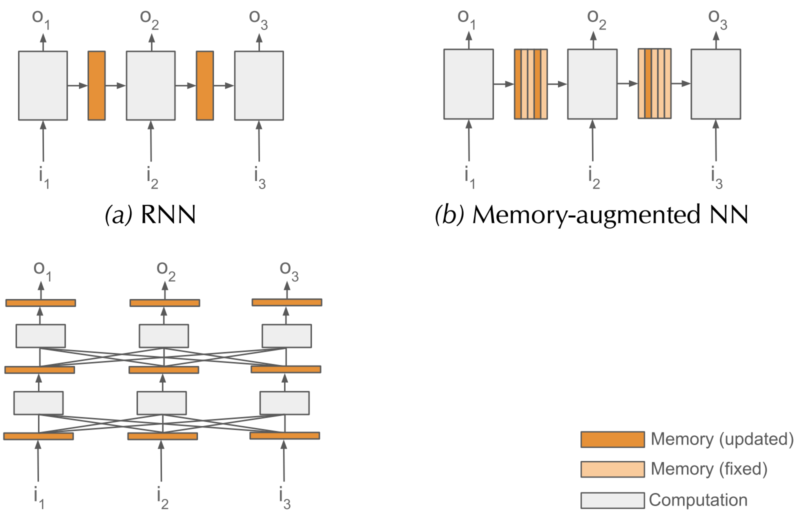
It is not enough to be able to store and retrieve memories, but the network should be able to make decisions based on the contents of the memory. This acts as a "control flow" for the algorithm they are learning to execute. RNNs and transformers have access to each memory cell in each of their processing steps (which are layers in the case of the transformer). However, memory-augmented neural networks typically have access only to a single or eventually a few memory cells at once. This makes decisions based on memory content difficult as the controller has to memorize the sequence of operations it has to perform based on the restricted amount of information available. We hypothesize that this can lead to easier overfitting on certain tasks.

How Many Memories Can be Updated Simultaneously?

Memory-augmented NNs update only one memory cell at a time. On the contrary, transformers and RNNs can update all of their memory. This enables implementing a parallel version of the algorithms they are potentially performing, shortening the path of the gradients, thus making the learning problem easier. There is an additional caveat: the way the input and output are usually handled in RNNs in practice makes them closer to the Memory-augmented NNs and not well suited for parallel processing (see Sec. 1.3.3). Transformers, on the other hand, process all inputs and outputs in parallel in practice as well. This, in theory, could enable the learning of parallel algorithms, similarly to a GPU.

Summary

Fig. 1.1 shows the memory architecture of popular NNs visually. In summary, of the architectures considered, based on their memory architecture, transformers seem to be best suited for learning parallel algorithms: They have separate memory cells that are operated on in parallel with identical processing structures that act on them. Each cell can be processed in parallel. The downside



(a) RNN

(b) Memory-augmented NN

(c) Transformer

Figure 1.1: Memory architecture of modern neural networks. Orange blocks denote memory cells. Dark orange denotes updated memory cells, while light orange denotes memory cells that are left unchanged in the given step. Light gray denotes a computation step. (a) An RNN [Elman, 1990] (e.g LSTM [Hochreiter and Schmidhuber, 1997]). It has a single memory cell that is updated at each step. (b) A memory-augmented RNN, such as NTM [Graves et al., 2014] or DNC [Graves et al., 2016]. The memory is extended to multiple cells, of which only some are updated. (c) (universal) transformers [Schmidhuber, 1991, 1992c, 1993; Vaswani et al., 2017; Dehghani et al., 2019]. Memory consists of multiple cells. Each cell is updated in each step (the step here corresponds to layers).

is the extensive computational demand if the algorithm to be considered is not parallelizable: In each step, each memory cell is updated even if only some of them should. As we shall see in Sec. 5.1.1, this can pose additional challenges beyond wasted computation, affecting the consistency of the data. Because all cells are updated at the same time, no global decision about the next computation step should be made, facilitating generalization.

1.3.2 The Need for Sequential Computation

Compositional behavior could be naturally achieved by breaking the problem into subproblems that are easy to solve (or memorized before) and combining the subresult [5]. Each of these steps represents an unsolved problem alone. Let us focus on combining the subresults. To have something to combine, the subresults should already have been calculated. This necessitates multiple steps of computation. Since, in general, it should be possible to combine operations in any order, knowledge of how to perform them must be available at each step. This requires recurrence. Classical results from the theory of computation also support this hypothesis: all well-known models of universal computation are either recurrent (Turing machines, cellular automatons) or recursive (Lambda calculus). RNNs, memory-augmented NNs, and Universal Transformers [Dehghani et al., 2019] are recurrent by their definition. However, the behavior of non-shared layer transformers in this regard is unclear.

Transformers and Recurrence

Transformers without shared layers are not recurrent by default. However, language models are trained for next-word prediction, providing a recurrent connection from the current output to the next input (see Fig. 1.2). The success of recent large language models [Radford et al., 2019; Brown et al., 2020; OpenAI, 2023] in reasoning tasks, especially with scratchpads [Nye et al., 2022] and chain-of-thought prompting [Wei et al., 2022] shows that these models are capable of reusing the subresults produced by them in an earlier timestep [Anil et al., 2022].

However, this kind of recurrence is limited. Only discrete symbols can be passed from one step to the next, and the distribution of such symbol sequences must match the distribution of the training data. On one hand, this can be helpful: The teacher forcing [Williams and Zipser, 1989] provides step-by-step supervision on the sequence of steps that the reasoning process should take. However, the granularity of this supervision might be insufficient. For example, Minerva [Lewkowycz et al., 2022] was trained on 60B tokens of arXiv papers related to mathematics, but sometimes it still fails in simple addition [6]. We hypothesize that one of the reasons is that in the training data (scientific papers), even though many examples of additions were seen, it is rare to write out the individual steps of the long addition algorithm. Even if the network could learn to break down the algorithm into its elementary steps, there is no way to incor-

[5] Other solutions might also be possible, for example generating the weights of a network that solves the problem in a single step. However, the composition of subresults in different timesteps, as CPUs and GPUs do, seems more natural. We decided to focus on this type of solution in our research.

[6] `https://minerva-demo.github.io/#category=Algebra&index=62`

porate their subresults in the input/output of the model.

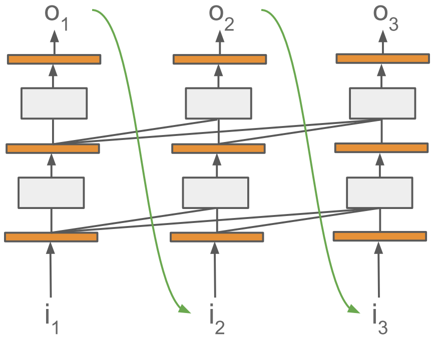
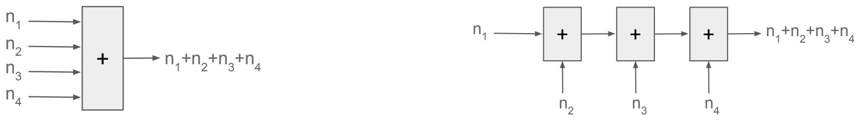


Figure 1.2: Autoregressive transformers feed back their output to their input in the next step (green arrow). This provides a form of a recurrence.

1.3.3 Breaking Down the Problem in Subproblems

Unlike transformers, RNNs (including memory-augmented NNs) can pass arbitrary memory content between individual computation steps, without restrictions. However, the number of computation steps performed is usually dictated by the input and output of the model: In each step, they consume one input and produce an output. This makes it impossible to break down a problem into a long list of subproblems. Methods of adaptive computation time (ACT; Schmidhuber [2012]; Graves [2016]; Banino et al. [2021]) are proposed as a remedy. However, using ACT with RNNs would require learning a fully sequential algorithm, which could require many steps and could result in vanishing or exploding gradients [Hochreiter, 1991], and it is not parallelizable. Introducing recurrence in transformers, by sharing their layers [Dehghani et al., 2019; Csordás et al., 2022a] seems to be a better solution: many algorithms can be (at least partially) parallelized, significantly reducing the number of the required steps, while also retaining a sufficient amount of sequential computation (in the layer dimension) to be able to compose subresults.

As an alternative view of this decomposition problem, consider the goal of avoiding fusing elementary operations together. For example, Fig. 1.3 shows an example of adding four numbers together. It is possible to do this in a single step with a 4-way adder or in 3 steps with a 2-way one. If the network learned to compose 2-way adders, it is likely that it will work with any number of operands (as long as the range of the input and result is within the one seen during the training). In the case of a fused operation, such a generalization is not possible. Unfortunately, the lack of adaptive computation time (see above) makes the models prefer fused solutions, since they are always doable in a single step.



(a) Short and parallel computation

(b) Long and sequential computation

Figure 1.3: An example of solving a problem (adding 4 numbers) in different ways. (a) Parallel solution: a 4-way adder. It can calculate the result in a single step, but it does not generalize to adding a different amount of numbers. (b) Serial solution with a 2-way adder. The same model should be able to add any amount of numbers.

1.4 Systematic Generalization in the Age of Large Language Models

Large Language Models [Radford et al., 2019; Brown et al., 2020; OpenAI, 2022, 2023], especially with instruction tuning [Ouyang et al., 2022], show remarkable reasoning capabilities [Bubeck et al., 2023]. This raises the question of whether any effort to improve systematic generalization or inducing algorithmic behavior has legitimacy.

First, we note that these models are trained on a cleaned-up version of the whole Internet [Brown et al., 2020; Gao et al., 2021; Raffel et al., 2020; Lewkowycz et al., 2022], including web crawls, books, open source code, synthetically generated data (typically for math), arXiv papers, etc. Most of these datasets are not publicly available. Thus, it is impossible to know whether some task is out-of-distribution. Also, some publicly available datasets are often part of the training data; thus benchmarking models that are trained after the dataset was released is delusive.

Currently, it is not clear what is the limit of these models. The papers released by the companies that develop them tend to exaggerate their capabilities [Bubeck et al., 2023; OpenAI, 2023]. Independent analysis suggests that they still struggle with generalization. Tian et al. [2023] analyzes ChatGPT on coding problems and finds that it struggles to generalize to novel problems, contrary to the claims of the original papers. He [2023] found that GPT-4 solves all 10 easiest problems on Codeforces before 2021 (which corresponds to their knowledge cutoff date[7]), but fails on all the newer ones. A more rigorous evaluation also shows a disappointing performance of GPT-4 on programming tasks [Enryu, 2023]. Weiss [2022] found that when ChatGPT is used as a code interpreter to ″run″ Python programs, it often ignores the details of the provided code and simply repeats a memorized sequence (e.g. the first 10 primes). Liu et al. [2023] demonstrated that ChatGPT and GPT-4 struggle with reasoning datasets developed after their knowledge cut-off date. Wu et al. [2023] generates a modified counterfactual version of common reasoning tasks and shows that the performance of the model is significantly reduced.

Overall, there is not enough evidence to conclude to what extent these models generalize. However, based on the independent evaluations and on the fact that they are not recurrent, there is good reason to believe that their fundamental limitations match those of the much smaller models. With a large enough training set, these limitations can be offset so that the models become usable for practical tasks, especially eliminating somewhat repetitive tasks like writing boilerplate code. Their factual knowledge is also very high, thanks to the enormous amount of data they were trained for. They could be used as an advanced, fuzzy search engine, but caution must be taken due to hallucinations. However, scaling by simply increasing the size of the dataset has limits. Chinchilla's scaling laws [Hoffmann et al., 2022] suggest that most of the improvements to the models come from the dataset size as opposed to the model size. But finding more good quality data can be challenging, given that we already train on most of the Internet [Nostalgebraist, 2022].

We believe that making the models more systematic would push their limits significantly. We also think that it would enable learning from a much smaller amount of data, building on previously learned knowledge, and reducing the necessary density of the coverage of the input space. Thus, we think that research on systematic generalization is of utmost importance.

1.5 Related Concepts

Systematic generalization is closely related to many other areas of machine learning. In the following, we will discuss what we consider the most important.

[7] Knowledge cutoff date is when the dataset was created. It does not contain any newer information.

Memory-augmented neural networks. Neural networks capable of reasoning and algorithmic processing need some form of memory. One of the early attempts to introduce memory into connectionist models was RNNs [Amari, 1972; Kohonen, 1972; Elman, 1990], but they suffered from vanishing and exploding gradients. LSTMs [Hochreiter and Schmidhuber, 1997] reduced the severity of this problem and made RNNs widely applicable. However, these methods have a significant limitation: the number of their parameters is proportional to the square of their memory size. To extend the available memory size without the explosion in the number of parameters, memory-augmented neural networks were proposed, such as Fast Weight Programmers [Schmidhuber, 1991, 1992c], Neural Turing Machines [Graves et al., 2014] and DNC [Graves et al., 2016]. These architectures are capable of performing more complex reasoning tasks than their predecessors. However, they typically struggle with generalization. Neural GPUs [Kaiser and Sutskever, 2016] are a more parallel approach that uses two-dimensional memory and convolutions. They claim to be able to generalize to very long sequences in simple algorithmic tasks, but they suffer from severe instabilities: Only very few seeds generalize out of hundreds [Kaiser and Sutskever, 2016]. Fast weight programmers are an alternative way to increase memory capacity [Malsburg, 1981; Schmidhuber, 1991, 1992c, 1993; Schlag and Schmidhuber, 2018; Irie et al., 2021]. Attention-based models [Schmidhuber, 1992a, 1993; Bahdanau et al., 2015], such as transformers [Vaswani et al., 2017; Dehghani et al., 2019] can also be considered memory-augmented networks, where the memory is directly initialized with the input and updated by every layer application. Due to their very parallel nature, they can be trained significantly faster, becoming the workhorse of the recent era of large language models [Brown et al., 2020; Devlin et al., 2019; Rae et al., 2022; Chowdhery et al., 2022; OpenAI, 2022, 2023].

Causality. focuses on disentangling the cause and its effects [Pearl, 2009; Pearl and Mackenzie, 2018]. Causal models avoid reliance on spurious correlations, which is a major limitation of current deep learning models. However, classically, these models require building in knowledge about the problem. Recently, the deep learning community has started to adopt ideas from causality [Schölkopf, 2019; Goyal et al., 2021b; Mitrovic et al., 2021], but learning causal relations is still an unsolved problem. In some sense, learning causal relations is similar to decomposing problems into a sequence of more elementary operations, which is the basis of compositionality.

Statistical learning theory. Statistical learning theory [Vapnik, 1998] can provide error bounds on unseen IID data. Neural tangent kernels [Jacot et al., 2018] extend the theory to deep neural networks in an infinite-width limit. However, none of these theories focuses on out-of-distribution (OOD) generalization and compositionality.

Domain adaptation. focuses on making the resulting model robust to domain shifts [Blitzer et al., 2006; Ben-David et al., 2010; Koh et al., 2021]. This literature is usually concerned about raw sensor inputs, not the reasoning core alone. Typical sources of domain shifts can be changes in the acquiring machine, camera, or data from geographically different places. It is not focused on compositions.

Continual and curriculum learning. Models that can generalize systematically should rely on the solutions of the elementary problems to build solutions for the more complex ones. This might be explicitly enforced, such as in curriculum learning [Elman, 1993; Schmidhuber, 2004; Bengio et al., 2009]. It is also related to continual learning: more and more complex skills must be learned based on existing ones without destroying old knowledge [Schlimmer and Fisher, 1986; McCloskey and Cohen, 1989; Ring, 1991; French, 1999; Kirkpatrick et al., 2017a; Mallya and Lazebnik, 2018].

Symbolic processing. Many theories of generalization are based on symbols and objects and their compositions [Whitehead, 1928; Fodor, 1975]. Others [Solomonoff, 1964a,b; Hutter, 2000] are based on Kolmogorov complexity, which is defined in the space of computer programs that are inherently symbolic and compositional. Symbolic AI (sometimes also called Good Old Fashioned AI, or GOFAI [Haugeland, 1985]) was the dominant approach in early AI systems used for theorem proving [Newell et al., 1959], expert systems [Shortliffe and Buchanan, 1975] and early NLP systems [Weizenbaum, 1966]. They had excellent generalization capabilities within their knowledge base by construction. However, many of them are incapable of learning, relying on a large number of rules defined by humans, which is very costly to obtain. Maintaining a consistent set of rules is prohibitively hard as the size of the knowledge bases grows. Symbolic machine learning systems are typically capable of learning by using some kind of search [Newell and Simon, 1961; Levin, 1973; Sussman, 1973; Deville and Lau, 1994; Schmidhuber, 2004]. For example, inductive logic programming methods [Plotkin, 1972; Shapiro, 1981] were capable of learning from positive and negative examples. These systems typically require formalizing the problem in some formal language. The expensive nature of discrete search also makes these methods impractical for large problems. However, if such a search could be done in an efficient way on raw data, the Kolmogorov complexity [Kolmogorov, 1965] should provide a way to select the best program out of multiple possible solutions: according to the Minimal Description Length principle [Rissanen, 1978], the model with the shortest description length should be chosen. These attractive properties of symbolic processing motivate the effort to combine them with neural networks in the so-called neuro-symbolic methods.

Neuro-symbolic methods. Symbolic methods are excellent in generalization but are incapable of learning. On the other hand, NNs are good at learning, but suffer in generalization. Neurosymbolic methods [Towell et al., 1990; Sun and Bookman, 1994; Roli et al., 1995; Chaudhuri et al., 2021] try to combine the advantages of the two worlds. However, the resulting methods are often limited and are usually dataset-specific. They require a significant amount of engineering for new applications [Silver et al., 2016a; Li et al., 2019; Chen et al., 2020; Liu et al., 2020; Li et al., 2022b]. A widely applicable and flexible neurosymbolic method has yet to be developed, although significant progress has been made recently [Ellis et al., 2021].

Meta-learning. The role of meta-learning for compositional generalization is underexplored. However, there have been some recent attempts [Deng and Zhang, 2020; Conklin et al., 2021b]. It is a promising approach: for example, it can make the search space more aligned to symbolic computation instead of pattern matching. Alternatively, multiple difficulty training splits can potentially be used to extract knowledge on what behavior is expected outside the training regime.

Representation learning. Many works focus on learning good representations [Ivakhnenko and Lapa, 1965b; Amari, 1967; Ivakhnenko, 1968, 1971; Hinton, 1984; Bengio et al., 2013a] that can be used efficiently for downstream tasks. This often includes disentangled representations [Barlow et al., 1989; Schmidhuber, 1992b; Higgins et al., 2018]. Here, the core of the underlying problem is to identify symbolically meaningful objects and relations between them and use them compositionally [Greff et al., 2020].

Modular neural networks. In these networks, specialized modules are componsed together explicitly. The structure of the network reflects the composition of the elementary operations. The way modules are composed is often learnable [Clune et al., 2013; Andreas et al., 2016; Kirsch et al., 2018; Chang et al., 2019; Goyal et al., 2021b; Ruis and Lake, 2022]. These methods usually generalize better but are hard to train. There can be multiple bottlenecks, for example, the generalization of the composer network or, in the case of no supervision over the structure, whether the learned modules correspond to semantically meaningful units.

Geometric deep learning. Learning on graph-structured data can be threated specially for learning more efficiently and better performance [Pollack, 1987; Sperduti, 1993; Baldi and Chauvin, 1996; Goller and Küchler, 1996; Küchler and Goller, 1996; Bronstein et al., 2017, 2021]. Most of the reasoning problems can be represented as graphs by explicitly specifying their structure. Thus, they can also be considered to be similar to the neurosymbolic methods. As the structure is known, reasoning problems expressed as graph networks usually generalize well [Velickovic et al., 2020; Velickovic and Blundell, 2021; Dudzik and Velickovic, 2022]. However, learning the structure of the graph together with the algorithm is an unsolved problem. Existing methods typically assume a fully connected graph, as in transformers.

1.6 Related Work Overview

Connectionist models are often criticized for their lack of systematicity and compositionality [Fodor et al., 1988; Fodor and McLaughlin, 1990; Marcus, 1998, 2003; Lake and Baroni, 2018; Hupkes et al., 2020; Greff et al., 2020]. Many papers propose datasets to analyze the generalization properties of NNs. Most of them are based on synthetic data with systematic differences between the train and the test set. Probably one of the most well-known such datasets is SCAN [Lake and Baroni, 2018; Loula et al., 2018], which remains unsolved by general-purpose architectures. Others include PCFG [Hupkes et al., 2020], CFQ [Keysers et al., 2020], COGS [Kim and Linzen, 2020] in the language domain, or CLEVR [Johnson et al., 2017] and CLOSURE [Bahdanau et al., 2019a] in the visual domain.

Countless papers analyze aspects of the generalization behavior of different, generally applicable neural networks [Bahdanau et al., 2019b; Klinger et al., 2020; Zhang et al., 2020; Bau and Andreas, 2021; Kharitonov and Chaabouni, 2021; Nogueira et al., 2021; Schwarzschild et al., 2021; Zhang et al., 2021]. In addition, some works compare specialized architectures to pre-trained models, and they usually find limited advantages of specialized architectures [Furrer et al., 2020; Dankers et al., 2021].

Some methods separate the prediction of alignment between source and target sequences from the generation of output tokens [Korrel et al., 2019; Li et al., 2019; Russin et al., 2019]. These methods typically improve on SCAN, except for the length generalization split. More general versions allow for permutation of the input sequence while maintaining an unconstrained transformation [Wang et al., 2021]. Other methods exploit the symmetries in the specific dataset to generalize better [Gordon et al., 2020].

Meta-learning [Schmidhuber, 1987] is also applied in the context of systematic generalization. Some methods are specialized [Lake, 2019], while others are generally applicable [Conklin et al., 2021a].

Among neuro-symbolic methods, Chen et al. [2020] achieve an impressive 100% generalization on all SCAN splits. However, it is not clear whether such architectures are generally applicable. Some of them leverage the structure of SCAN to achieve significant gains [Liu et al., 2020]. Liu et al. [2021] achieve excellent generalization on semantic parsing tasks, but requires a task-specific hardcoded tree decoder. Guo et al. [2020] design a multistage architecture. The stages include generating sketches, filling them, and ranking them. This leverages the structure of the semantic parsing problem and shows big improvements. Weißenhorn et al. [2022] show that compositional neural parsers are very beneficial for parsing tasks, but Kim [2021] shows that his quasi-synchronous grammar-based approach is too restrictive for more general problems. Sartran et al. [2022] integrate syntactic inductive bias in the attention of a transformer, but their method requires a linearized parse tree as input. Sometimes, more general architectures are trained with immediate supervision to perform more algorithmically [Yan et al., 2020]. The approach of Shaw et al. [2021] uses a symbolic solver and falls back to a neural backend if it fails. Some approaches generate a (partially) discrete program in some form [Vani et al., 2021].

Generalizing to longer sequences in algorithmic tasks has proven to be especially difficult. Currently, only hybrid task-specific neuro-symbolic approaches made significant progress [Nye et al., 2020; Chen et al., 2020; Liu et al., 2020].

Universally applicable architectures are usually designed with reasoning as the primary focus. Sometimes, they are explicitly tested for generalization on algorithmic tasks [Graves et al., 2014; Kaiser and Sutskever, 2016; Graves et al., 2016; Freivalds et al., 2019; Schlag et al., 2019; Herzig and Berant, 2020]. Mathematics [Saxton et al., 2019; Charton et al., 2021; Lewkowycz et al., 2022] is an especially popular domain. A parallel line of work is motivated by causality [Mittal et al., 2020; Goyal et al., 2021b]. Others use certain restrictions motivated by compositionality [Hudson and Manning, 2018; Akyürek and Andreas, 2021; Chaabouni et al., 2021; Liu et al., 2022]. Alternatives extend the expressiveness of well-known architectures [Dubois et al., 2020; Mittal et al., 2021], or use different tricks to boost generalization [Oren et al., 2020; Ontañón et al., 2021]. Iterative processing is also shown to help generalization [Ruiz et al., 2021]. Varying the length of iterative processing has also been shown to be beneficial [Schmidhuber, 2012; Graves, 2016; Banino et al., 2021].

Some architecture-agnostic methods to improve generalization include data augmentation [Andreas, 2020; Akyürek and Andreas, 2022]. Zheng and Lapata [2022] iteratively re-encode the input concatenated with the partial output. Qiu et al. [2021] learns a grammar to generate new data. Curriculum learning can also have a significant effect on generalization [Zaremba and Sutskever, 2015].

With the advent of big language models [Devlin et al., 2019; Brown et al., 2020; Rae et al., 2022; OpenAI, 2022, 2023] a new data-driven approach appeared. They provide input prompts or finetune the network so that the output includes the result of each processing step to be performed [Nye et al., 2022; Wei et al., 2022]. It has been empirically shown to significantly improve generalization, although it is rarely tested on OOD data.

Chapter 2

Improving Differentiable Neural Computers[1]

The Differentiable Neural Computer (DNC) [Graves et al., 2016] is one of the first models to demonstrate strong reasoning capabilities. It combines large external memory with advanced addressing mechanisms, such as content-based look-up and temporal linking of memory cells. Unlike specific approaches that achieve state-of-the-art performance only on specific tasks, e.g. MemNN [Sukhbaatar et al., 2015] or Key-Value Networks [Miller et al., 2016] for the bAbI dataset [Weston et al., 2016], the DNC consistently reached near state-of-the-art at the time. This generality made the DNC worth further study.

We discovered three problems with standard DNC. They revolve around the *content-based lookup mechanism*, which is the main memory addressing system, and the *temporal linking* used to read memory cells in the same order in which they were written. First, the lack of key-value separation negatively impacts the accuracy of content retrieval. Second, the current deallocation mechanism fails to remove deallocated data from memory, which prevents the network from erasing outdated information without explicitly overwriting the data. Third, with each write, the noise from the write address distribution accumulates in the temporal linking matrix, degrading the overall quality of temporal links.

Here, we propose a solution to each of these problems. We allow for dynamic key-value separation through masking of both the key and data, which is more general than a naive fixed key-value memory, yet does not suffer from loss of accuracy in addressing content. We propose to wipe the content of a

[1] For the full paper please see our work "Improving Differentiable Neural Computers Through Memory Masking, De-allocation, and Link Distribution Sharpness Control" [Csordás and Schmidhuber, 2019]

memory cell in response to a decrease of its usage counter to allow for proper memory de-allocation. Finally, we reduce the effect of noise accumulation in the temporal linking matrix through exponentiation and renormalization of the links, resulting in improved sharpness of the corresponding address distribution.

These improvements are orthogonal to other previously proposed DNC modifications. Incorporation of the differentiable allocation mechanism [Ben-Ari and Bekker, 2017] or certain improvements to memory usage and computational complexity [Rae et al., 2016] might further improve the results reported in this paper. Certain bAbI-specific modifications [Franke et al., 2018] are also orthogonal to our work.

We empirically evaluated each of the proposed modifications on a benchmark of algorithmic tasks and on bAbI [Weston et al., 2016]. In all cases, we find that our model outperforms the DNC. In particular, on the bAbI task, we observe a 43% relative improvement in terms of mean error rate. We find that improved de-allocation together with sharpness enhancement leads to zero error and 3x faster convergence on the large repeated copy task, while DNC is not able to solve it at all.

Sec. 2.1 provides a brief overview of the DNC. Sec. 2.2 discusses identified problems and proposed solutions in more detail. Sec. 2.3 analyzes these modifications one by one, demonstrating their positive effects.

2.1 Brief Overview of DNC

Here we provide a brief overview of the Differentiable Neural Computer (DNC). Presenting the model in full detail is out of the scope of this thesis. More details can be found in the original work of Graves et al. [2016] and in Appendix A.1.

The DNC combines a neural network (called controller) with an external memory that includes several supporting modules: to read and write memory, to allocate new memory cells, to chain memory reads in the order in which they were written, and to search memory for partial data. A simplified block diagram of the memory access is shown in Fig. 2.1. Please note that DNCs use a different notation in their attention mechanism than the Transformers models popular nowadays. We try to respect the original notation introduced by Graves [2016] in this work.

External memory. The main component of DNC is an external 2D memory organized in cells ($\boldsymbol{M}_t \in \mathbb{R}^{N \times W}$, where N is the number of cells and W is the size of the memory word) N is independent of the number of trainable parameters.

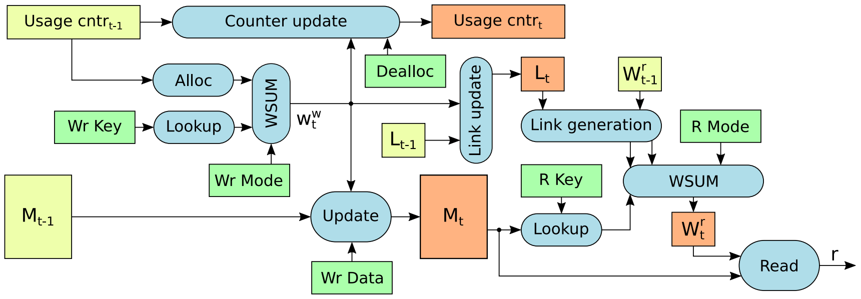


Figure 2.1: Simplified block diagram of DNC's memory access module with single read head. Yellow boxes denote the inputs from the previous time step, orange boxes are the corresponding outputs to the next time step. Green boxes are the control inputs from the controller. Blue, rounded boxes are modules responsible for a specific function. $\boldsymbol{w}_t^w$ denotes the write address, $\boldsymbol{w}_t^r$ the read address, $\boldsymbol{L}_t$ the temporal linkage matrix. $\boldsymbol{M}_t$ is the memory. Arrow "r" denotes the output of the memory read.

The controller (typically an LSTM) is responsible for producing control signals for the gates and for the memory transactions. Memory is accessed through multiple read heads and a single write head. Cells are addressed in a soft way using attention, which provides a soft distribution over the whole address space. Each cell is read and written at each time step to an extent determined by the address distributions, resulting in a differentiable procedure.

Memory addressing. The DNC uses three different addressing methods. The most important one is content-based look-up. It compares every memory cell with a key ($\boldsymbol{k}_t^* \in \mathbb{R}^W$) produced by the controller, resulting in a score, which is later normalized to get an address distribution over the whole memory. The second is temporal linking, which has 2 types: forward and backward. It is a soft adjacency matrix ($\boldsymbol{L}_t \in \mathbb{R}^{N \times N}$) that shows which cell is being written after and before the one read in the previous time step. It is used to project any address distribution to the address that follows ($\boldsymbol{f}_t^i \in [0,1]^N$) or precedes it ($\boldsymbol{b}_t^i \in [0,1]^N$). Temporal linkage is useful for processing continuous sequences of data and is used only for read heads. The third addressing method is the allocation mechanism, which is used only by the write heads for allocating new memory cells.

Memory allocation. Memory allocation works by maintaining usage counters for every cell. These are incremented on memory writes and optionally decremented on memory reads (de-allocation). When a new cell is allocated, the one with the lowest usage counter is chosen. Deallocation is controlled by a gate: weighted by the gate and the address distribution of the previous read, the usage counter of each cell is decreased.

Read / Write. In each step, the memory is first written and then read. A write address is generated as a weighted average of the write content-based lookup and the allocation distribution. The update is gated for each channel separately by the "erasing vector" $\boldsymbol{e}_t \in [0,1]^W$. Parallel to the write, the temporal linkage matrix is also updated. Finally, the memory is read. The read address distribution ($\boldsymbol{w}_t^{r,i} \in [0,1]^N$) is the weighted average of the read content-based lookup distribution and forward and backward temporal links. The weighted sum of the memory cells is calculated using this address, resulting in a single vector, which is the retrieved data. This is combined with the output of the controller to produce the final output of the model.

2.2 Method

Masked content-based addressing. Content-based addressing aims to find memory cells similar to a given query vector. This query contains partial information (it is a partial memory), and the content-based memory read completes its missing part based on previous memories. However, controlling which part of the query vector to search for is difficult because there is no key-value separation: The entire query is compared with all memory cells to produce the similarity score. The part of the cell that is unknown during the search time and should be retrieved is also used in the normalization part of the cosine similarity, resulting in an unpredictable score. With less information in the query vector (the longer the part to be retrieved), the problem becomes worse. This could result in less similar cells having higher scores and could flatten the resulting address distribution because the extra data generate an offset. Due to normalization before softmax, it behaves similarly to an increased temperature parameter.

We propose solving the problem by providing a way to explicitly mask the unknown part which should not be used in the query. This is more general than key-value memory since key-value separation can be controlled dynamically and does not suffer from the incorrect score problem. We achieve this by producing a separate mask vector $\boldsymbol{m}_t^* \in [0-1]^W$ by the controller and multiplying both the query and the memory content by it before comparing (β is the write query strength controlling the temperature of the softmax and

$D(\boldsymbol{u},\boldsymbol{v})$ is the cosine distance (see Eq. 2.4), h is the head):

$$C(\boldsymbol{M},\boldsymbol{k},\beta,\boldsymbol{m}) = \mathrm{softmax}(D\left(\boldsymbol{k}\odot\boldsymbol{m},\boldsymbol{M}\odot\mathbb{1}\,\boldsymbol{m}^\top)\beta\right) \tag{2.1}$$

$$\boldsymbol{c}_t^w = C\left(\boldsymbol{M}_{t-1},\boldsymbol{k}_t^w,\beta_t^w,\boldsymbol{m}_t^w\right) \qquad \boldsymbol{c}_t^{r,h} = C\left(\boldsymbol{M}_t,\boldsymbol{k}_t^{r,h},\beta_t^{r,h},\boldsymbol{m}_t^{r,h}\right) \tag{2.2}$$

De-allocation and content-based look-up. The DNC tracks the allocation states of memory cells by so-called usage counters, which are increased on memory writes and optionally decreased after reads. When allocating memory, the cell with the lowest usage is chosen. Decreasing is done by elementwise multiplication with a so-called retention vector ($\boldsymbol{\psi}_t$), which is a function of previous read address distributions ($\boldsymbol{w}_{t-1}^{r,i}$) and of scalar gates. The vector $\boldsymbol{\psi}_t$ indicates to what extent the current memory should be kept. The problem is that deallocation affects only the usage counters and not the actual memory $\boldsymbol{M}_t$. Memory content plays a central role in both read and write address generation: the content-based lookup still finds deallocated cells, resulting in memory aliasing. Thus, we propose to clear the memory contents by multiplying every cell of the memory matrix $\boldsymbol{M}_t$ with the corresponding element of the retention vector. Then the memory update equation becomes:

$$\boldsymbol{M}_t = \boldsymbol{M}_{t-1}\odot\boldsymbol{\psi}_t\mathbb{1}^\top\odot\left(\boldsymbol{E}-\boldsymbol{w}_t^w\boldsymbol{e}_t^\top\right)+\boldsymbol{w}_t^w\boldsymbol{v}_t^\top \tag{2.3}$$

where $\odot$ is the element-wise product, $\mathbb{1}\in\mathbb{R}^N$ is a vector of ones, $\boldsymbol{E}\in\mathbb{R}^{N\times W}$ is a matrix of ones. Note that the cosine similarity (used to compare queries with the memory content) is normalized by the length of the memory content vector, which would normally cancel the effect of Eq. 2.3. However, in practice, due to numerical stability, cosine similarity is implemented as

$$D(\boldsymbol{u},\boldsymbol{v}) = \frac{\boldsymbol{u}\cdot\boldsymbol{v}}{|\boldsymbol{u}||\boldsymbol{v}|+\epsilon} \tag{2.4}$$

where ϵ is a small constant. In practice, free gates f_t^i tend to be almost 1, so $\boldsymbol{\psi}_t$ is very close to 0, making the stabilizing constant ϵ dominate the norm of the erased memory content vector. This will assign a low score to the erased cell in the content addressing: the memory is removed.

Sharpness of temporal link distributions. With temporal linking, the model can sequentially read memory cells in the same or reverse order as they were written. For example, traversing a sequence is possible without content-based lookup: forward links $\boldsymbol{f}_t^i$ can be used to jump to the next cell. Any address distribution can be projected to the next or previous one by multiplying it by a so-called temporal link matrix ($\boldsymbol{L}_t$) or its transpose. $\boldsymbol{L}_t$ can be understood as a continuous adjacency matrix. On every write, all elements of $\boldsymbol{L}_t$ are updated to the extent controlled by the write address distribution ($\boldsymbol{w}_t^w$). Links related to previous writes are weakened; the new links are strengthened. If $\boldsymbol{w}_t^w$ is not one-hot, the order information for all nonzero addresses will be (partially) replaced in $\boldsymbol{L}_t$ with noise from the current write. This is done repeatedly in each step. Thus, the forward ($\boldsymbol{f}_t^i$) and backward ($\boldsymbol{b}_t^i$) distributions of long-term-present cells are becoming increasingly noisy and flatten out with time. When chaining multiple reads with temporal links, the new address is generated by repeatedly multiplying by $\boldsymbol{L}_t$, making the blurring effect exponentially worse.

We propose introducing a sharpness enhancement step $S(\boldsymbol{d}, s)$ to the generation of the temporal link distribution. By exponentiating and renormalizing the distribution, the network can adaptively control the importance of non-dominant elements. This does not fix the noise accumulation in the link matrix $\boldsymbol{L}_t$, but significantly reduces the effect of exponential blurring behavior when following temporal links, making noise in $\boldsymbol{L}_t$ less harmful.

$$\boldsymbol{f}_t^i = S\left(\boldsymbol{L}_t \boldsymbol{w}_{t-1}^{r,i}, s_t^{f,i}\right) \qquad \boldsymbol{b}_t^i = S\left(\boldsymbol{L}_t^\top \boldsymbol{w}_{t-1}^{r,i}, s_t^{b,i}\right) \qquad S(\boldsymbol{d}, s)_i = \frac{(\boldsymbol{d}_i)^s}{\sum_j (\boldsymbol{d}_j)^s} \tag{2.5}$$

The scales $s_t^{f,i} \in \mathbb{R}$ and $s_t^{b,i} \in \mathbb{R}$ should be generated by the controller ($\hat{s}_t^{f,i}$ and $\hat{s}_t^{b,i}$). The oneplus nonlinearity is used to bound them in the range $[0, \infty)$: $s_t^{f,i} = oneplus(\hat{s}_t^{f,i})$ and $s_t^{b,i} = oneplus(\hat{s}_t^{b,i})$.

2.3 Experiments

To analyze the effects of our modifications, we used simple synthetic tasks designed to stress test all critical parts of the DNC while leaving the internal dynamics somewhat human-interpretable. These tasks allow us to analyze the contributions of individual components. We also conducted experiments on the significantly more complex bAbI dataset [Weston et al., 2016].

We tested several variants of our model. For clarity, we use the following notation: DNC is the baseline, DNC-D has modified deallocation, DNC-S has sharpness enhancement, and DNC-M has masking in content-based addressing. Multiple modifications (D, M, S) can be present simultaneously.

Copy Task. A sequence of length L of random binary vectors of size W is presented to the network, and the task is to repeat them. After all the inputs are presented, a special token indicates the start of the repeat phase. To solve this task, the network must remember the sequence, which requires allocating memory and recalling it. However, this alone does not require memory de-allocation and reuse. To force the network to demonstrate its deallocation capabilities, N instances of such data are generated and concatenated. Because the total length $N \cdot L$ of the sequences exceeds the number of cells in memory, the network must reuse its memory cells. An example is shown in Fig. 2.3a.

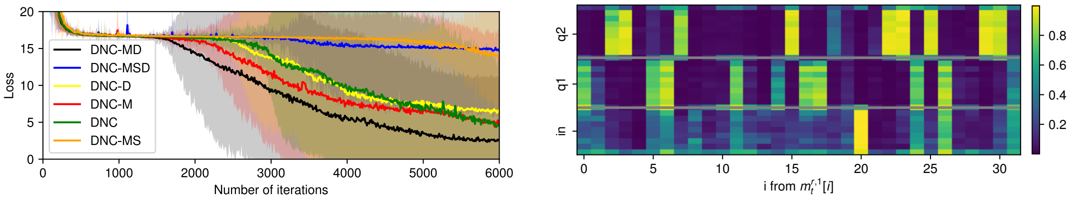
Associative Recall Task. In the associative recall task [Graves et al., 2014] B blocks of W_b words of size W are presented to the network sequentially, with special bits indicating the start of each block. After presenting the input to the network, a special bit indicates the start of the recall phase, where a randomly chosen block is repeated. The network needs to output the next block in the sequence.

Key-Value Retrieval Task. The key-value retrieval task demonstrates some properties of memory masking. L words of size $2W$ are presented to the network. The words $\boldsymbol{w}$ are divided into two halves of equal size, $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$. In the first phase, all the words are presented to the network. After that, the words are shuffled and the $\boldsymbol{w}_1$ halves are fed to the network, requiring it to complete the missing part $\boldsymbol{w}_2$. Next, the words are shuffled again, $\boldsymbol{w}_2$ is presented, and the corresponding $\boldsymbol{w}_1$ has to be completed. The network must be able to query its memory using either part of the words to complete this task.

2.3.1 The Effect of Modifications

Masking. Fig. 2.2a shows the performance of various models on the associative recall task. The two models that perform best use memory masking. Masking also improves the convergence speed. Sharpening negatively affects performance on this task (see Sec. 2.3.2 for further discussion).

De-allocation. The authors of the original DNC paper [Graves et al., 2016] successfully trained the model on the repeat copy task with a small number of repetitions (N) and relatively short length (L). We found that increasing N causes the DNC to fail (see Fig. 2.3a). On the contrary, Fig. 2.3b shows that our model solves the task perfectly. Its outputs are clean and sharp. Furthermore, it converges much faster than the baseline DNC, reaching a near-zero loss very quickly. We hypothesize that the reason for this is the modified de-allocation:



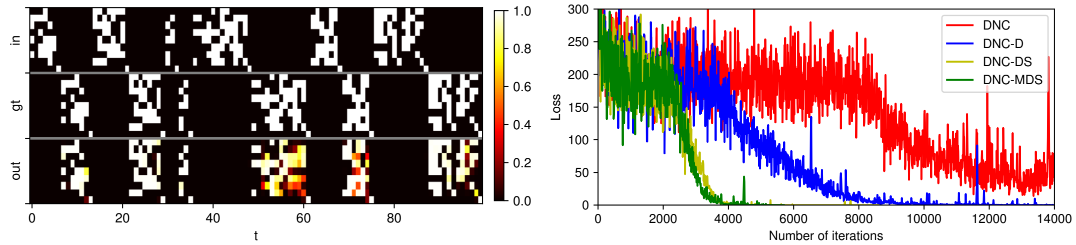
(a) Effect of masking on convergence speed

(b) A sample mask from DNC-M

Figure 2.2: (a) Mean training loss on the associative recall task. The shaded area shows the $\pm 2\sigma$ mark (12 seeds/model). Masking improves convergence speed. (b) An example read mask of DNC-M on the key-value retrieval task. The y-axis shows individual time steps, starting from the bottom. Yellow values indicate parts of the key the network searches for. When the query switches from $\boldsymbol{w}_1$ to $\boldsymbol{w}_2$, the mask changes accordingly. In the bottom third (in) the input is stored (look-up is not used). For in the middle third (q1) $\boldsymbol{w}_1$ is presented in random order and $\boldsymbol{w}_2$ is retrieved. In the last third (q2) $\boldsymbol{w}_2$ is presented in random order and $\boldsymbol{w}_1$ is retrieved. When the query switches from $\boldsymbol{w}_1$ to $\boldsymbol{w}_2$, the mask switches accordingly, as intended.

The network can store the beginning of every sequence with a similar key without causing look-up conflicts, as it is guaranteed that the previously present key is erased from memory. DNC seems to be able to solve the short version of the problem by learning to use different keys for every repeat step, which is not a general solution. This hypothesis, however, is difficult to prove, as neither the write vector nor the lookup key is easily human-interpretable.

Sharpness enhancement. To analyze the problem of temporal link degradation after successive updates of the link matrix, we examined the forward and backward link distributions ($\boldsymbol{f}_t^i$ and $\boldsymbol{b}_t^i$) of the model with modified deallocation (DNC-D). The forward distribution is shown in Fig. 2.4a. The problem presented in Sec. 2.2 is clearly visible: the distribution is blurry and the problem becomes worse with each iteration. Fig. 2.4c shows the read mode ($\boldsymbol{\pi}_t^1$) of the same run. It is clear that only the content-based addressing (middle column) is used. When the network starts repeating a block, the weight of forward links (last column) increased only marginally, but as the distribution becomes blurrier, the network falls back to pure content-based lookup. Probably it is easier for the network to perform a content-based look-up with a learned counter as a key rather than to learn to restore the corrupted data from blurry reads. Fig. 2.4b shows the for-



(a) Input, ref output, net output (repeated copy)

(b) Train loss of repeated copy task

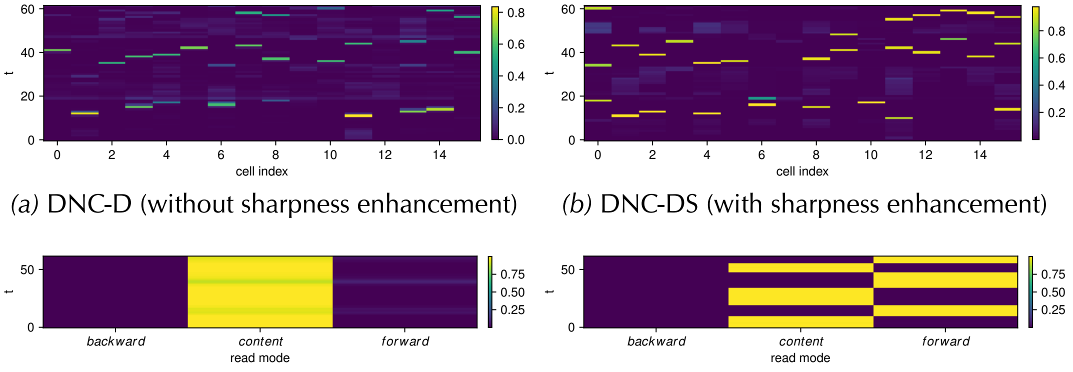
Figure 2.3: (a) Input (top), ground truth (middle), and network output (bottom) of DNC on a long repeat copy tasks. DNC fails to solve the task; the output is blurry. The problem is especially apparent starting from $t = 50$. (b) De-allocation and sharpness enhancement substantially improve convergence speed. The improvement caused by the masking is marginal, probably because the model uses temporal links to solve the task.

ward link distributions of the model with sharpness enhancement as suggested in Sec. 2.2 for the same input. The distribution is much sharper, staying sharp until the very end of the repeat block. The read mode ($\boldsymbol{\pi}_t^1$) for the same run can be seen in Fig. 2.4d. The network prefers to use the links in this case.

2.3.2 bAbI Experiments

bAbI [Weston et al., 2016] is a synthetic question answering dataset containing 20 different tasks. The data is organized in sequences of sentences called stories. The tokenization is word-level. When a question mark is encountered, the network must output a single word that represents the answer. A task is considered solved if the error rate (the number of incorrect answer words divided by the number of total predictions) decreases below 5%, as usual for this task.

Manually analyzing bAbI tasks suggests that some are difficult to solve in a single timestep. Consider the sample from task QA16: "Lily is a swan. Bernhard is a lion. Greg is a swan. Bernhard is white. Brian is a lion. Lily is gray. Julius is a rhino. Julius is gray. Greg is gray. What color is Brian? *A: white*" The network should be able to "think for a while" about the answer: it needs to perform multiple memory searches to link the clues. This cannot be done in parallel as the result of one query is needed to produce the key for the next. One solution would be to use adaptive computation time [Schmidhuber, 2012; Graves, 2016]. However, that would add an additional level of complexity. So, we decided to



(a) DNC-D (without sharpness enhancement) *(b)* DNC-DS (with sharpness enhancement)

(c) DNC-D (without sharpness enhancement) *(d)* DNC-DS (with sharpness enhancement)

Figure 2.4: (a), (b) Example forward link distribution on the copy task with 3 repeated blocks. Each row is an address distribution across all memory cells. Yellow cells indicate memory cells taking part in the read operation, while blue cells are ignored. (a) DNC-D: without sharpness enhancement the distributions are blurry, rarely having peaks near 1.0. The problem becomes worse over time. Note that only $t \in [9, 18], [34, 46]$ and $[54, 62]$ are memory reads, the other parts store the input, where no meaningful read distribution is expected. (b) Sharpness enhancement (DNC-DS) makes the distribution sharp during the read, peaking near 1.0. Note that (a) and (b) have identical input data. (c), (d) The $\boldsymbol{\pi}_t^1$ distribution for (a) and (b). Columns are the weighting of the backward links, the content-based lookup, and the forward links, respectively. (c) The forward links are not used without sharpness enhancement. (d) With sharpness enhancement, the forward links are used for every block.

insert $T = 3$ thinking steps before reading the answer—a difference from what was done previously [Graves et al., 2016]. We also use a word embedding layer, instead of one-hot input representation, as is typical for NLP tasks. The embedding layer is a learnable lookup table that transforms word indices to a learnable vector of length E.

In Tab. 2.1, we present the experimental results of our modifications after $0.5M$ training iterations with batch size 2. The performance of the original DNC [Graves et al., 2016] is also shown (column "Graves et al"). Our best-performing model (DNC-MD) reduces the mean error rate by 43%, while also having a lower variance. This model does not use sharpness enhancement, which penalizes mean performance by only 1.5% absolute. We hypothesize that this is due to the nature of the task, which rarely needs step-to-step transversal of words but requires many content-based look-ups. When following the path of an object, many words and even sentences might be irrelevant between the clues, so the sequential linking in the order of writing is of little to no use. Compare this to the work of Franke et al. [2018], where the authors completely removed the temporal linking for bAbI. However, we argue that for other types of tasks, link distribution sharpening might be crucial (see Fig. 2.3b, where sharpening helps and masking does not).

The mean test error curves are shown in Fig. 2.5. Our models converge faster and have both lower error and lower variance than DNC. (Note that our goal was not to achieve state-of-the-art performance at the time [Santoro et al., 2017; Henaff et al., 2017; Dehghani et al., 2019] on bAbI. It was to exhibit and overcome certain shortcomings of the DNC.)

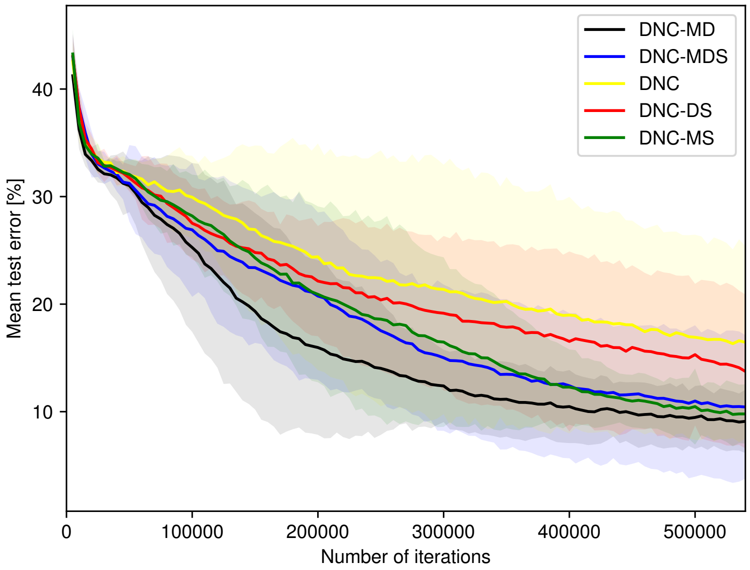


Figure 2.5: Mean test error of various models during the training. The shadowed area shows $\pm 2\sigma$.

2.4 Conclusion

We identified three drawbacks of the traditional DNC model and proposed fixes for them. Two of them are related to content-based addressing: (1) Lack of key-value separation yields uncertain and noisy address distributions resulting from content-based look-up. We mitigate this problem with a special masking method.

Task	DNC (ours)	DNC-MDS	DNC-DS	DNC-MS	DNC-MD	Graves et al
1	2.5 ± 4.4	0.4 ± 1.2	0.7 ± 1.6	0.0 ± 0.1	**0.0 ± 0.0**	9.0 ± 12.6
2	29.0 ± 19.4	8.6 ± 10.1	18.6 ± 15.1	7.8 ± 5.9	**6.9 ± 4.7**	39.2 ± 20.5
3	32.3 ± 14.7	10.8 ± 9.5	16.9 ± 13.0	**7.9 ± 7.8**	12.4 ± 5.1	39.6 ± 16.4
4	**0.8 ± 1.5**	0.8 ± 1.5	6.4 ± 10.0	0.8 ± 1.0	**0.1 ± 0.2**	0.4 ± 0.7
5	1.5 ± 0.6	1.6 ± 1.0	**1.3 ± 0.5**	1.7 ± 1.1	1.3 ± 0.7	1.5 ± 1.0
6	5.2 ± 6.8	1.1 ± 2.1	2.4 ± 3.8	**0.0 ± 0.1**	0.1 ± 0.1	6.9 ± 7.5
7	8.8 ± 5.8	3.4 ± 2.3	7.6 ± 5.1	**2.5 ± 2.0**	3.0 ± 5.0	9.8 ± 7.0
8	11.6 ± 9.4	4.6 ± 4.5	10.9 ± 7.9	**1.8 ± 1.6**	2.5 ± 2.1	5.5 ± 5.9
9	4.5 ± 5.8	0.8 ± 1.9	2.0 ± 3.3	**0.1 ± 0.2**	**0.1 ± 0.2**	7.7 ± 8.3
10	9.1 ± 11.5	2.6 ± 3.9	4.1 ± 5.9	0.6 ± 0.6	**0.5 ± 0.5**	9.6 ± 11.4
11	11.6 ± 9.4	0.1 ± 0.1	0.1 ± 0.2	**0.0 ± 0.0**	**0.0 ± 0.0**	3.3 ± 5.7
12	1.1 ± 0.8	**0.2 ± 0.2**	0.5 ± 0.4	0.3 ± 0.4	**0.2 ± 0.2**	5.0 ± 6.3
13	1.1 ± 0.8	**0.1 ± 0.1**	0.2 ± 0.2	0.2 ± 0.2	**0.1 ± 0.1**	3.1 ± 3.6
14	24.8 ± 22.5	8.0 ± 13.1	20.0 ± 19.4	1.8 ± 0.9	**2.0 ± 1.6**	11.0 ± 7.5
15	40.8 ± 1.4	26.3 ± 20.7	42.1 ± 6.3	33.0 ± 15.1	**23.6 ± 18.6**	27.2 ± 20.1
16	**53.1 ± 1.2**	54.5 ± 1.8	53.5 ± 1.4	53.2 ± 2.3	53.9 ± 1.2	53.6 ± 1.9
17	**37.8 ± 2.5**	39.9 ± 3.2	40.1 ± 2.0	41.2 ± 3.0	39.8 ± 1.2	32.4 ± 8.0
18	7.0 ± 3.0	6.3 ± 4.1	9.4 ± 0.9	3.3 ± 2.2	**2.0 ± 2.6**	4.2 ± 1.8
19	67.6 ± 8.6	48.6 ± 32.8	67.6 ± 7.9	48.1 ± 26.7	**40.7 ± 34.9**	64.6 ± 37.4
20	**0.0 ± 0.0**	0.9 ± 0.9	1.5 ± 1.0	5.3 ± 12.5	0.1 ± 0.1	0.0 ± 0.1
mean	16.9 ± 5.2	11.0 ± 3.8	15.3 ± 3.5	10.5 ± 1.9	**9.5 ± 1.6**	16.7 ± 7.6

Table 2.1: bAbI error rates of different models after 0.5M iterations of training [%]

(2) De-allocation results in memory aliasing. We fix this by erasing the memory contents in parallel to decreasing usage counters. (3) We try to avoid blurring the temporal linkage address distributions by sharpening the distributions.

We experimentally analyzed the effect of each novel modification on synthetic algorithmic tasks. Our models achieved convergence speed-ups on all of them. In particular, modified de-allocation and masking in content-based look-up helped in every experiment we performed. The presence of sharpness enhancement should be treated as a hyperparameter as it benefits some, but not all, tasks. Unlike DNC, DNC+MDS solves the long repeated copy task. DNC-MD improves the mean error rate on bAbI by 43%. The modifications are easy to implement, add only a few trainable parameters, and barely affect the execution time.

Chapter 3

Inspecting the Implicit Modularity of Neural Networks[1]

Modularity provides an intuitive way to achieve compositionality, which appears to be essential for systematic generalization. Explicitly modular NNs tend to significantly improve generalization [Clune et al., 2013; Andreas et al., 2016; Kirsch et al., 2018; Chang et al., 2019; Bahdanau et al., 2019b; Goyal et al., 2021b]. Previous work showed that NNs are clusterable [Watanabe, 2019; Filan et al., 2020], but not if these clusters correspond to functionally meaningful modules that support compositionality. The emergence of such functional modules would show that NNs have a natural tendency towards breaking down the problem into subproblems and learning a modular solution.

In this work, we contribute new insights into the generalization capabilities of popular neural networks by investigating whether modules implementing specific functionality emerge and to what extent they enable compositionality. This calls for a *functional* definition of modules, which has not been studied previously in prior work. In particular, we consider functional modules given by subsets of weights (i.e. subnetworks) responsible for performing a specific 'target functionality', such as solving a subtask of the original task. By associating modules with performing a specific function, they become easier to interpret. Moreover, depending on the target functionality chosen, modules at multiple different levels of granularity can be considered.

To unveil whether a NN has learned to acquire functional modules, we propose a novel analysis tool that works on pre-trained NNs. Given an auxiliary task corresponding to a particular target function of interest (e.g., train only on a

[1] For the full paper please see our work "Are Neural Nets Modular? Inspecting Functional Modularity Through Differentiable Weight Masks" [Csordás et al., 2021]

specific subset of the samples from the original dataset), we train probabilistic, binary, but differentiable masks for all weights (while the NN's weights remain frozen). The result is a binary mask that unveils the module necessary to perform the target function. Our approach is simple yet general, which readily enables us to analyze several popular NN architectures on a variety of tasks in this way, including recurrent NNs (RNNs), Transformers [Vaswani et al., 2017], feedforward NNs (FNNs) and convolutional NNs (CNNs).

To investigate whether the discovered functional modules are part of a compositional solution, we analyze whether the NN has the following two desirable properties: ($\mathbf{P}_{\mathbf{specialize}}$) *it uses different modules for very different functions*, and ($\mathbf{P}_{\mathbf{reuse}}$) *it uses the same module for identical functions* that may have to be performed multiple times[2]. Here, we treat $\mathrm{P}_{\mathrm{specialize}}$ and $\mathrm{P}_{\mathrm{reuse}}$ as continuous quantities, which lets us focus on the degree to which functional modularity emerges. Furthermore, since for many tasks it is unclear what precise amount of sharing is desirable, we will measure $\mathrm{P}_{\mathrm{specialize}}$ and $\mathrm{P}_{\mathrm{reuse}}$ by considering the *change* in performance as a result of applying different masks corresponding to a target function. This yields an easy-to-interpret metric that does not assume precise knowledge about the desired level of weight sharing. We experimentally show that many typical NNs exhibit $\mathrm{P}_{\mathrm{specialize}}$ but not $\mathrm{P}_{\mathrm{reuse}}$. By additionally analyzing the capacity for transfer learning, we provide further insight into this issue. We offer a possible explanation: While simple data routing between modules in standard NNs is often highly desirable, it requires learning a special structure that is hard to separate out from the desired transformation induced by the loss. Indeed, our findings suggest that standard NNs have no bias towards separating these conceptually different goals of data transformation and information routing.

We also demonstrate how the functional modules discovered by typical NNs do not tend to encourage compositional solutions. For example, we analyze encoder-decoder LSTMs [Hochreiter and Schmidhuber, 1997] and Transformers [Vaswani et al., 2017] on the SCAN dataset [Lake and Baroni, 2018] designed to test systematic generalization based on textual commands. We show that combination-specific weights are learned to deal with certain command combinations, even when they are governed by the same rules as the other combinations. The existence of such weights indicates that the learned

[2] We emphasize the distinction between the ability to *reuse* modules and the ability to compose them: a compositional solution may fail to reuse a module to implement the same behavior multiple times. Similarly, weights can be reused without being composed to yield a compositional solution. Furthermore, we consider *specialization* of modules a special case of *modularization* where modules are specialized to implement a particular functionality that is semantically meaningful.

solution is non-compositional and fails at performing the more symbolic manipulation required for systematic generalization on SCAN. To demonstrate that this issue is present even in more real-world scenarios, we highlight identical behavior on the challenging Mathematics Dataset [Saxton et al., 2019].

Finally, we study whether functional modules emerge in CNNs trained for image classification, which are thought to rely heavily on shared features. Surprisingly, we can identify subsets of weights solely responsible for single classes: when removing these weights the performance on its class drops significantly. By analyzing the resulting confusion matrices, we identify classes relying on similar features.

3.1 Discovering Modules via Weight-Level Introspection

To investigate whether functional modules emerge in neural networks, one must perform a weight-level analysis. This precludes the use of existing methods, which discover the modular structure in NNs based on the clustering of individual *units* according to their similarity [Watanabe, 2019; Filan et al., 2020] and that may not always be enough to draw meaningful conclusions. Units can be shared even when their weights, which perform the actual computation, are not. Indeed, units can be viewed as mere "wires" for transmitting information. Consider, for example, a gated RNN, such as an LSTM, where gates can be controlled either by the inputs or the state, yet it makes use of different weights to project to the same gating units. To overcome this limitation, we propose a novel method to inspect pre-trained NNs at the level of individual weights. It works as follows. First, we formulate a target task corresponding to the specific function for which we want to investigate if a module has been learned. For example, this can be a subset of the original problem (i.e. a subtask), or based on a particular dataset split, e.g. to test generalization. Next, we train a weight mask on this target task while keeping the weights themselves frozen. The resulting mask then reveals the module (subnetwork) responsible for the target task.

To train the mask, we treat all N weights separately from each other. Let $i \in [1, N]$ denote the weight index. The mask's probabilities are represented as learned logits $l_i \in \mathbb{R}$, which are initialized to keep the weights with high probability (0.9). If continuous masks were applied to the weights, it would be possible to scale them arbitrarily, potentially modifying the function the network performs. To prevent this, we binarize masks, which allows only keeping or removing individual weights. Binarization is achieved using a Gumbel-Sigmoid with a straight-through estimator, which we derive from the Gumbel-Softmax [Jang et al., 2017; Maddison et al., 2017] in Appendix B.1.1. A sample $s_i \in [0, 1]$ from the mask can be drawn as follows:

$$s_i = \sigma\left(\left(l_i - \log\left(\log U_1 / \log U_2\right)\right) / \tau\right) \text{ with } U_1, U_2 \sim \mathrm{U}(0, 1), \tag{3.1}$$

where $\tau \in (0, \infty)$ is the temperature and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. Next, we can use a straight-through estimator [Hinton, 2012; Bengio et al., 2013b] to obtain a binarized sample $b_i \in \{0, 1\}$:

$$b_i = [\mathbb{1}_{s_i > 0.5} - s_i]_{\text{stop}} + s_i, \tag{3.2}$$

where $\mathbb{1}_x$ is the indicator function and $[\cdot]_{\text{stop}}$ is an operator for preventing backward gradient flow. In this case, the b_i are samples from a $\mathrm{Bernoulli}(\sigma(l_i))$ random variable (proof in Appendix B.1.3). Masks are applied element-wise: $w'_i = w_i \cdot b_i$. Training is done by applying the loss function defined by the target task and backpropagating [Kelley, 1960; Linnainmaa, 1970; Werbos, 1982] into logits l_i. Typically, multiple binary masks (between 4—-8) are sampled and applied to different parts of a batch to improve the quality of the estimated gradient.

The goal of masking is to remove weights that are not necessary to perform the target function. Therefore, the logits l_i should be regularized such that the probability that the weight w_i is active is small unless w_i is necessary for the task. This is achieved by adding a regularization term $r = \alpha \sum_i l_i$ to the loss, where $\alpha \in [0, \infty)$ is a hyperparameter responsible for the strength of the regularization. How to best choose α is described in detail in Appendix B.3.3. At the end of the training process, deterministic binary masks $M_i \in \{0, 1\}$ for weights i are obtained via thresholding $M_i = \mathbb{1}_{\sigma(l_i) > 0.5}$[3]. Applying the full mask M then uncovers the module responsible for the target task. A preliminary study confirmed that the mask training process is stable and therefore suitable for inspection (Appendix B.2.1).

In the following sections, we will analyze several standard NNs using this technique of mask-training[4][5]. Throughout our experiments, we will avoid drawing conclusions based on the measured amount of sharing alone as much as possible, since it is unclear what degree of sharing can be expected or is desirable. Rather, we will analyze the performance drop caused by removing weights

[3] In general we find that l_i concentrates at either 0 or 1 and so thresholding is safe (see also Fig. B.3).

[4] A complete overview of all experimental details is available in Appendix B.3. The mean and standard deviations shown in the figures are calculated over 10 runs unless otherwise noted.

[5] Code for all experiments is available at `https://github.com/robertcsordas/modules`.

corresponding to certain functionality, which offers a more consistent and easier-to-interpret metric[6]. For example, to show that a module *is* responsible for a particular subtask (A) but not for another (B), we train masks on A and test on both. A performance drop is expected on task B only. In contrast, to show that this module *is exclusively* needed for a particular subtask, we can invert the masks and test on both tasks. The inverted masks are expected to perform well on the complementary task, but not on the original one. However, we note that this mask inversion method is limited to analyzing entirely disjoint weights.

We analyze weight sharing *between* two tasks using two different metrics: one is *Intersection over Union (IoU)*, which measures how much the weights used for solving the tasks overlap. The other is the Szymkiewicz-Simpson coefficient (also known as the "Overlap coefficient"). This coefficient measures the number of overlapping weights (intersection) divided by the minimum of the total number of weights used for each task. For consistency with IoU, we call this metric *Intersection over Minimum (IoMin)*. Intuitively, IoMin is a measure of "subsetness". If no weights are shared, both IoU and IoMin are zero. If all weights are shared, both IoU and IoMin are one. However, when the weights needed for one task are a strict subset of the other, then IoMin is one, while IoU < 1.

3.2 Analyzing Fundamental Properties of Modules

Let us consider $\mathrm{P}_{\mathrm{specialize}}$ and $\mathrm{P}_{\mathrm{reuse}}$ (defined in Sec. 3) in more detail, as they reflect the advantages of the modular compositional design. According to our notion of functional modularity, an NN is not modular without $\mathrm{P}_{\mathrm{specialize}}$. Moreover, disjoint modules prevent catastrophic interference [McCloskey and Cohen, 1989; Rosenbaum et al., 2019], since changing the weights responsible for a specific function does not affect the others. $\mathrm{P}_{\mathrm{reuse}}$ also has multiple advantages. It increases data efficiency by processing all relevant data using the same module, which thus receives additional training when a module can be reused. It also helps with generalization. For example, consider processing the expressions $a * b$ and $(c + d) * e$ where a, b, c and d are sampled from the same range. By reusing the multiplier, it will be able to perform $a * b$ on a wider range of inputs than it would otherwise be trained for.

In this section, we conduct several experiments using synthetic datasets

[6] Exceptions only include cases where the observed amount of weight sharing can be clearly interpreted. However, even in these cases, our analysis will focus on general trends rather than the precise amounts observed.

to test whether NNs have a natural inductive bias supporting $P_{specialize}$ and P_{reuse}. These experiments are designed to be as simple as possible to isolate the property of interest. Let's assume the network consists of compositional modules. The input of such modules can come from multiple sources within the network. Similarly, their output could be connected to different parts of the network. For example, in the previous arithmetic expression, the first operand of the multiplier can come directly from the input or the output of the adder. The same holds for the outputs. Therefore, we consider cases where the inputs and outputs are shared between the modules of the ideal solution (*shared I/O*) and where they are separated (*separate I/O*).

We construct two different datasets for analyzing $P_{specialize}$ and P_{reuse}. For $P_{specialize}$, we use shared I/O and two different target functions (addition/multiplication task in Sec. 3.2.1). The shared I/O biases the network towards weight sharing by default. Thus, we use this dataset to test whether there is a bias to specialize different computations (functions) to separate weights. In contrast, to test for P_{reuse}, we construct a dataset where the same function should be performed twice, but using separate I/O (double addition task in Sec. 3.2.2). Since separate I/O biases the network to not share weights at initialization time, we will make use of this dataset to test whether NNs exhibit a bias for reusing computation. Here reusing weights is expected since information routing is assumed to be easier to learn than the actual function (addition). We emphasize that this initial bias due to different choices for I/O arises naturally in any network composing multiple different internal modules to arrive at a solution.

The conclusions are surprising: typical NNs tend to satisfy $P_{specialize}$ but not P_{reuse}. Our experiments suggest that weight sharing across tasks is mostly driven by shared I/O rather than task similarity, resulting in redundancies and lack of data efficiency.

3.2.1 Addition/Multiplication Experiments

The addition/multiplication dataset is designed to test $P_{specialize}$. The task is to add or multiply numbers (modulo 100). The input and output units are the same for both operations. An additional one-hot input specifies the operation. The numbers are two-digit and encoded as two 10-way one-hot vectors, each representing a digit. Thus, the total input is 42-dimensional and the output is 20.

First, we train the network to perform this task without any masking. Once the performance is nearly perfect, we freeze its weights. We perform two stages of mask training: first, we train a mask on addition (multiplication examples excluded), then we repeat this procedure for multiplication.

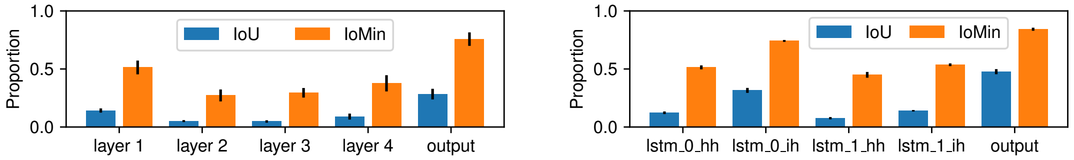


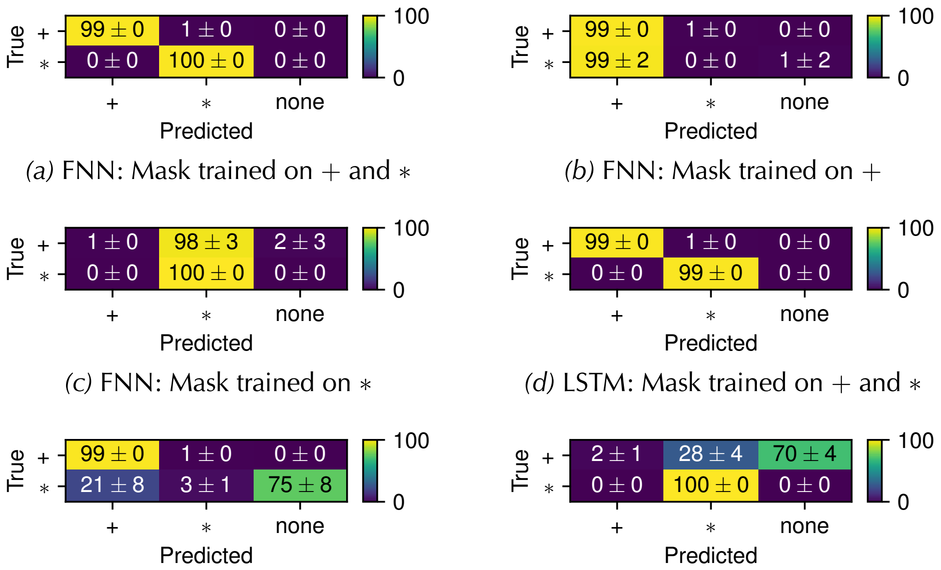
Figure 3.1: Proportion of shared weights per layer on addition/multiplication. Left: FNN, right: LSTM.

We analyze FNN and LSTM on this task. For LSTM, we present the full input for a fixed number of timesteps. The result is the output at the final step. No loss is applied in intermediate steps. Regardless of the architecture, we found the same general tendencies: There is more sharing in the input and output layers and less in the hidden layers (Fig. 3.1). We also found that the multiplication uses 3.8 times more weights than the addition (Fig. B.6), which partially explains the low IoU in this case. We conclude that there does appear to be some bias towards specializing weights according to different functions. However, the separation might still be inadequate to prevent interference and catastrophic forgetting. Increased sharing in I/O layers could be due to a switching/routing procedure used to select which operation to perform.

We also analyze how performance breaks down on the task for which the mask was *not* trained on. Here, the behavior of the FNN and the RNN differ. The FNN tends to ignore the function description and performs the operation for which the mask was trained, while the LSTM tends to produce invalid outputs, suggesting that it learned a solution where the two operations are more intertwined (Fig. 3.2).

3.2.2 Double-Addition Experiments

The double-addition experiment is designed to test property P_{reuse}. The task is to perform modulo 100 addition twice using separate I/O (different units) for each of the two instances. Using the inputs a, b, c and d, the network should output $a+b$ and $c+d$. This simulates the realistic scenario of having different data sources within a network when composing modules dynamically, without considering the additional problem of finding the right composition. Since the operation is the same and the operands' data distributions are exact matches, this simple setup encourages sharing. The encoding is the same as in Sec. 3.2.1, resulting in 80 input and 40 output units.



(a) FNN: Mask trained on + and ∗

(b) FNN: Mask trained on +

(c) FNN: Mask trained on ∗

(d) LSTM: Mask trained on + and ∗

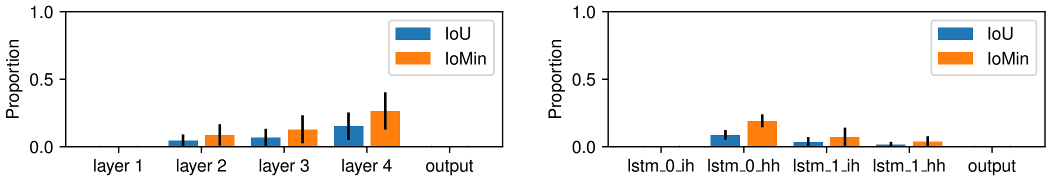
(e) LSTM: Mask trained on +

(f) LSTM: Mask trained on ∗

Figure 3.2: Analysis of FNN (a, b, c) and LSTM (d, e, f) performance degradation on the addition/ multiplication task. The y-axis shows the target operation. The x-axis shows the actual operation performed. "none" means the predicted number is neither the result of addition nor multiplication. The FNN ignores the operator specification and performs the one corresponding to the mask; in contrast, the LSTM tends to perform invalid operations.

		Full	Pair 1	¬Pair 1	Pair 2	¬Pair 2
FNN	Pair 1	100 ± 0.0	100 ± 0.0	20 ± 12.7	1 ± 0.1	92 ± 10.5
	Pair 2	100 ± 0.0	1 ± 0.1	94 ± 6.7	100 ± 0.0	21 ± 11.0
LSTM	Pair 1	100 ± 0.0	100 ± 0.0	2 ± 0.5	1 ± 0.1	99 ± 3.0
	Pair 2	100 ± 0.0	1 ± 0.1	100 ± 0.2	100 ± 0.0	2 ± 0.3
LSTM (forced)	Pair 1	100 ± 0.0	100 ± 0.0	4 ± 0.8	1 ± 0.1	99 ± 0.7
	Pair 2	100 ± 0.1	1 ± 0.1	96 ± 4.1	100 ± 0.0	3 ± 0.6

Table 3.1: Double-addition task: accuracy [%] of LSTMs and FNN on the two pairs. In case of LSTM (forced) only one input is presented at a time (to prevent interference). The header shows on which pair the mask was trained on. ¬ denotes an inverted mask.



(a) FNN

(b) LSTM

Figure 3.3: Double addition task: proportion of weights shared per operation in case of (a) feedforward network, (b) LSTM, both inputs presented together. The first and last layers have no shared weights.

We first train the network until convergence on the full task, then freeze its weights. We train a mask on $a + b$, followed by $c + d$. We analyze both FNN and LSTM architectures. FNN needs special care to avoid activation interference. When both operations have to be performed simultaneously, sharing is impossible. Thus, for the FNN, we perform two forward passes. In each pass, we feed only one pair of numbers to the network (either a, b or c, d), while zeroing out the other. With LSTM, we investigate two different settings. In the first, both pairs are presented together for a fixed number of steps, and the result is the final output. Therefore, the LSTM is allowed to schedule the execution of the operations freely. In the second setting, called LSTM (forced), we remove any incentive for solving the pairs simultaneously by feeding a single pair for multiple steps with the other zeroed out, and then read its output. This procedure is then repeated for the second pair without resetting the state.

The results of all experiments are consistent: the weight sharing is low (Fig. 3.3). To assert the modules' independence, we invert masks trained on pair 1, removing all weights needed for pair 1. We test the resulting network on pair 2, where the performance decreases only slightly, suggesting that they are independent (Tab. 3.1, further analysis is provided in Appendix B.3.5). No differences were observed between the two LSTM variants.

These observations show that P_{reuse} is violated even in this simple case. Realistic scenarios tend to be more complex, as the data distribution for different operation instances might be different (with overlaps), providing even fewer incentives to share. Furthermore, comparing the results to those of Sec. 3.2.1, it is apparent that sharing depends more on the location of the inputs/outputs than on the similarity of the performed operations. This behavior is undesired and calls for further research.

3.2.3 Transfer Learning Experiments

Let us now consider a more complex setting to assess the degree to which property $\mathrm{P}_{\mathrm{reuse}}$ is violated (see Sec. 3.2.2). Here, we will measure the amount of possible transfer in a continual learning setup using the popular permuted MNIST benchmark [Kirkpatrick et al., 2017b; Golkar et al., 2019; Kolouri et al., 2019]. A sequence of tasks is created by applying different permutations to MNIST images [LeCun et al., 2010]. Spatially close pixels may no longer be observed in nearby locations in this case, which leads us to train a FNN (as opposed to a CNN) sequentially on all permutations (tasks).

Continual learning is closely related to *transfer learning*, which is additionally concerned with transferring knowledge between tasks to improve learning and use fewer parameters. Typical approaches revolve around freezing used weights via masking when a new task is added [Fernando et al., 2017; Mallya and Lazebnik, 2018; Golkar et al., 2019]. We adjust our method accordingly: We train on a single task and freeze the occupied weights. In particular, to be able to bias the network towards weight sharing, we train masks and weights simultaneously in this case. The free weights are then reinitialized, and a new mask is allocated for the next task to obtain a mask for each permutation.

Note that since each task differs only by the input's permutation, it suffices to retrain a new 'first layer' to undo the permutation so that later layers can be reused. Indeed, since a significant portion of the weights is in the hidden layers, knowledge transfer between the permutations is possible and expected to be beneficial. However, relearning the first layer may not always be possible in practice, since the required weights could already have been occupied to address a previous permutation. To ensure that this does not happen, we always reset the first layer and do not freeze any of its elements. Notice how, while this departs from the standard transfer learning setting, it still provides us with an *upper bound* on the amount of transfer that is possible when no such conflict occurs. Even with these modifications, we observed only a small weight sharing when there was sufficient free space available (Fig. 3.4). Only once all the capacity is saturated, the weights become shared. This effect is especially apparent for the output layer.

We also conducted an experiment in which we explicitly bias the network toward sharing. We initialized elements of new masks corresponding to the occupied weights with a significantly higher probability ($P \approx 0.88$) compared to the unused ones ($P \approx 0.27$). Intuitively, this encourages reusing the old, frozen network and adds new weights with low probability. This was able to force the network to share significantly in later layers (see Fig. B.7). However,

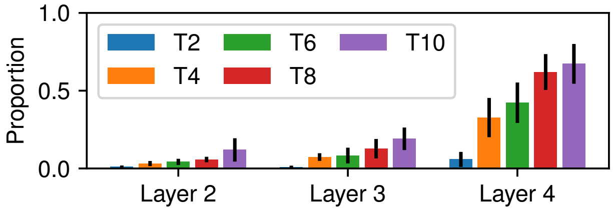


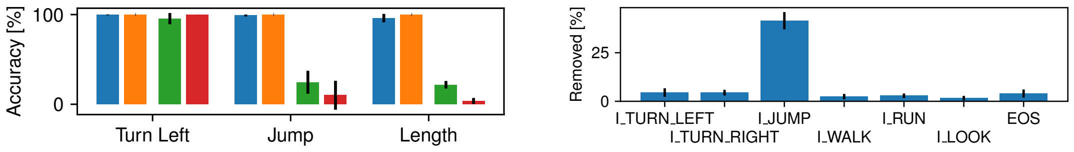
Figure 3.4: Proportion of the weights of a task shared with any of the previous tasks. Every second task on permuted MNIST. Each task corresponds to a permutation. The last layer has the lowest capacity, filling up first, forcing subsequent runs to share weights.

we emphasize that knowledge about which weights have to be reused between which samples is usually not available and therefore explicitly biasing the network in this way is generally not possible.

Together, these observations reaffirm that P_{reuse} does not emerge naturally and that the same functionality is re-learned. This is both redundant and potentially harmful, as we investigate in Sec. 3.3.

3.2.4 A Potential Explanation for Lack of Weight Sharing

Let us consider a possible explanation for the lack of weight-sharing observed in sections 3.2.2 and 3.2.3, which is that data routing is difficult in standard NNs. Indeed, inputs and outputs must be correctly routed to different sources/targets to reuse modules in different compositions. In routing networks [Kirsch et al., 2018; Rosenbaum et al., 2019; Chang et al., 2019], this is achieved through hand-designed mechanisms. Without those, routing can only occur through the weights of the NN. However, such a 'routing transformation' would change the data representation alongside the routing unless the weights have a special structure that we empirically find is hard to learn. Indeed, our experiments suggest that NNs find it hard to learn to represent data similarly along different information routes that can in principle be processed by a single module. We argue that this is an important issue and that additional research on suitable inductive biases is needed to address this. A further discussion of the potential role of attention to mitigate this is provided in Appendix B.2.4.



(a) Performance on different splits

(b) Weights removed from last layer on "Add jump"

Figure 3.5: Results of experiments on SCAN. (a) Test accuracy on split shown on x-axis with masks trained on the full problem (blue, orange) and with masks trained on split shown on x-axis (green, red). LSTM: blue, green, Transformer: orange, red (b) Percentage of weights removed per token from the output layer of the LSTM decoder trained on the "Add jump" split.

3.3 Analyzing Systematic Generalization on Algorithmic Tasks

Let us now consider the known issue of systematic generalization in light of our previous observations. To be able to combine modules in novel ways P_{reuse} should hold. The SCAN dataset [Lake and Baroni, 2018] is designed to analyze the degree to which NNs can generalize systematically. It consists of compositional commands (e.g. "jump twice"), to be translated into primitive output moves (e.g. "JUMP JUMP"). The "simple" data split is IID, the "length" split has shorter training samples than test samples, and the "add primitive" splits have a particular command presented in the training set but no compositions of this command with others (as in the test set).

It was previously shown that typical NNs generalize poorly on data splits systematically different from the train set [Lake and Baroni, 2018; Saxton et al., 2019]. However, the root of the problem is unclear. In fact, there might be two explanations: (a) The NN might have learned the correct algorithm for solving the problem but failed to pick up on certain symmetries between concepts due to the scarce evidence in the train set. For example, in the "add primitive" split, the NN might not be able to form an analogy between the additional primitive and the well-performing ones. This can also be understood as a representation problem: in this case, the NN has failed to represent the new primitive in a way that allows them to be used for problem solving in a similar manner following the acquired solution. However, the NN is not pressured to improve, since the learned solution suffices to solve the training set. (b) Alternatively, the NN may not have learned the correct algorithm to solve the problem. For example, it may have learned to recognize patterns determining when an output token should be produced in place of reusable rules. In this case, new weights are required to solve new problems of the same kind since they correspond to different patterns. Only in this case, we argue, has the NN failed to leverage the problem's compositional nature. Note that (a) requires P_{reuse} to hold so that the weights responsible for performing each individual operation are *shared* between different samples, while (b) does not.

We have tested two networks: the baseline 2 layer LSTM encoder-decoder model by Lake and Baroni [2018] and a Transformer [Vaswani et al., 2017] (see Appendix B.3.7). We pretrain the model on the IID data, which *ensures that the learned weights are capable of solving the full problem* and that a potential absence of sufficient evidence for learning about the correct symmetries between concepts is not an issue. Hence, this rules out explanation (a) being the only issue. For each split, we train a mask on its train set and measure the discovered subnetwork's performance on the corresponding systematically different test set. This process removes the weights that are not required to solve the train set for a given split. However, all splits' train sets contain sufficient information about the *full* set of rules required to perform well on *any* split. Hence, if the masking process removes any important weights, then we argue that the solution is likely pattern-recognition-like rather than based on reusable rules, providing evidence for explanation (b). Indeed, our experimental results demonstrate precisely this behavior, as can be seen from the large generalization gap in Fig. 3.5a. Note that while this gap is consistent with the findings of Lake and Baroni [2018], we are additionally able to provide evidence that the learned algorithm is likely inherently non-compositional, i.e. by eliminating explanation (a) being the only issue as a possibility.

To assess if the same behavior can be observed in a more complex setting, we conduct a similar experiment on the challenging Mathematics Dataset [Saxton et al., 2019]. Here, we generated difficulty-based splits for tasks like differentiation, solving linear equations, sorting, etc. (further details in Appendix B.3.7). In Fig. 3.6 a consistent performance drop can be observed when applying the inferred subnetwork on the "hard" split using a mask trained on the "easy" split. This demonstrates that samples in the "hard" split depend on exclusive weights, despite those being governed by the same underlying rules, which is consistent with the results on SCAN.

The weight level analysis provided by our method enables us to gain further insight. We inspect the LSTM decoder's weights on the "add jump" split of SCAN and note that the most apparent difference is in the output layer.

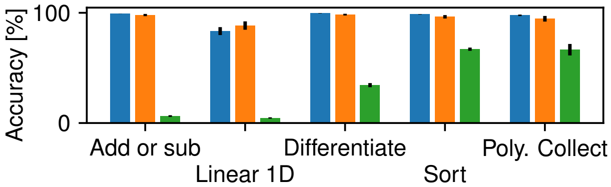


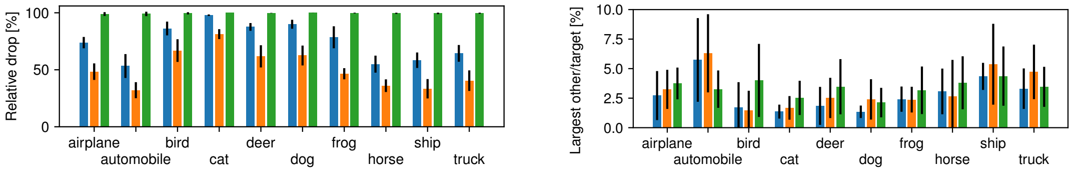
Figure 3.6: Accuracy on the "hard" test set of different tasks of the Mathematics Dataset: model without masks, masks trained on IID data and masks trained on "easy" set. A performance drop can be observed, because of the sample-specific weights. 5 seeds/task.

Almost half of the weights corresponding to "I_JUMP" are removed (Fig. 3.5b), suggesting that the network learned to detect patterns of cases when "I_JUMP" should be the output, and the last layer puzzles them together. In contrast, we hypothesize that the generalizing algorithm for solving such problems necessitates proper variable manipulation [Garnelo and Shanahan, 2019].

3.4 Analyzing Convolutional Neural Networks

As a final case study, we consider whether we are also able to observe a lack of weight-sharing in CNNs. By conducting a weight-level analysis using our tool, we are able to highlight sets of non-shared weights solely responsible for individual classes. We consider multiple CNN architectures trained on CIFAR10 [Krizhevsky et al., 2009]: a simple CNN with dropout (an ablation is provided in Appendix B.3.8) and a ResNet-110 [He et al., 2016] (based on the Highway Net [Srivastava et al., 2015b]). Full details are available in Appendix B.3.8). We proceed as follows. First, we train a 'control mask' on the full dataset to highlight all used weights. Next, we train a mask with a single class removed so that the weights solely responsible for this class will be absent from the resulting mask. Here, we avoid removing all weights responsible for the this class from the output layer (leaving no connection to the corresponding output unit) by fixing its mask to one trained on the full dataset. This corresponds to inspecting the feature detector layers as opposed to the classifier. We repeat this process for all classes to obtain a total of 11 masks.

We compute the confusion matrix on the full validation set at the end of each stage. Then we calculate the difference between the confusion matrices with and without the removed class, which unveils how the removal changes the classification. Interestingly, the performance of the target class drops signif-



(a) Relative performance drop per class

(b) Largest drop in non-target class (relative)

Figure 3.7: (a) Relative drop in performance for simple CNN, simple CNN without dropout and ResNet-110. (b) Largest performance drop in a non-target class relative to the drop in the target class.

icantly (Fig. 3.7a), only a small drop in performance (possibly due noise when mask sampling) is observed for non-target classes (Fig. 3.7b). This indicates a large dependence on class-exclusive, non-shared weights in the feature detectors. These findings, which assume that the network has sufficient capacity relative to dataset size, are in line with those observed in Sec. 3.2–3.3. Analyzing the difference in misclassification rates yields further insight: As the true positive rate drops, certain other classes are predicted instead that appear to rely on similar shared features. For example, removing "airplane" causes images to be classified as "birds" and "ships" instead, which have a blue background in common. Additional insights are reported for other classes in Fig. B.11 in Appendix B.3.8.

3.5 Related Work

There have been few other attempts at analyzing emerging modularity in NNs. Filan et al. [2020] identifies groups of neurons with strong internal and weak external connectivity via clustering, while others group neurons based on their connectivity pattern [Watanabe et al., 2018] or cluster them hierarchically based on activation statistics [Watanabe, 2019]. However, as we have argued, without considering the contribution of individual weights it is not always possible to reason about *functional* modularity. Davis et al. [2020] considers an alternative approach based on mutual information to detect salient pathways in NNs that could in principle allow for this. However, the discovered pathways are not grounded with respect to particular functionality, nor is it analyzed whether they support compositionality. Bengio et al. [2015] formulate adaptive mask learning as a reinforcement learning problem, with the main goal of accelerating inference via conditional execution. However, the masking is unit-level and trained together with network weights. Similarly, functional modularity is not considered.

Finally, we note that many transfer and continual learning methods make use of weight freezing via masking to prevent catastrophic forgetting [Fernando et al., 2017; Mallya and Lazebnik, 2018; Golkar et al., 2019; Yang et al., 2020]. Determining the importance of individual weights has been studied in network pruning [LeCun et al., 1989; Hassibi and Stork, 1992; Li et al., 2017; Frankle and Carbin, 2019; Gaier and Ha, 2019] and feature attribution [Simonyan et al., 2013; Springenberg et al., 2015; Sundararajan et al., 2017; Shrikumar et al., 2017] often using weight and/or gradients magnitudes. Differentiable binary weight masks have also been explored in the multitask setting [Mallya et al., 2018], albeit deterministically in contrast to the Gumbel-Sigmoid used here. It should also be mentioned how many explicitly modular architectures have been proposed to improve generalization [Clune et al., 2014; Andreas et al., 2016; Kirsch et al., 2018; Chang et al., 2019; Goyal et al., 2021b] and data efficiency [Purushwalkam et al., 2019]. Rather than engineering an explicitly modular solution, our goal is to let this emerge naturally. We believe that our current findings help take a step in that direction.

3.6 Conclusion

Our new method for inspecting modularity in neural networks is the first to identify modules by their functionality. It is a powerful tool for analyzing how the NNs share or separate weights based on the performed computation. By analyzing diverse sets of neural networks (FNNs, CNNs, RNNs, Transformers), we could draw significant novel conclusions: In typical current NNs, weight sharing between modules does not reflect task similarity (as desired) but can mostly be explained by rather trivial shared I/O interfaces of solution-implementing modules. The lack of weight sharing between multiple uses of the same function makes the learning data inefficient since it has to be relearned repeatedly. Moreover, NNs trained on algorithmic tasks appear to fail to learn general, modular, compositional algorithms. Rather, we have shown that they require specific subset weights to solve a particular combination of the input tokens, even when the same rules govern both the solution and the other samples. Our discoveries call for future research: function-dependent weight sharing in the neural networks should vastly improve data efficiency, and encouraging algorithmic solutions should improve generalization.

Chapter 4

Improving the Systematic Generalization of Transformers[1]

We already hinted in Sec. 1.3.1 that the structure of transformers seems to be well suited for generalization on algorithmic tasks. Additionally, the computation graph or a parse tree of the problem can be mapped to the columns of the transformer. On the contrary, in recently proposed works, including PCFG [Hupkes et al., 2020] and COGS [Kim and Linzen, 2020], baseline transformer models are typically shown to fail dramatically. However, the configurations of these baseline models are questionable. In most cases, standard practices from machine translation are applied without adaptation. Furthermore, some existing techniques relevant to the problem, such as relative positional embedding [Shaw et al., 2018a; Dai et al., 2019], are typically not included in the baseline.

In order to develop and evaluate methods to improve systematic generalization, it is necessary to have not only good datasets but also strong baselines to correctly evaluate the limits of existing architectures and to avoid a false sense of progress over bad baselines. In this work, we demonstrate that the capability of transformers [Vaswani et al., 2017] and, in particular, its universal variants [Dehghani et al., 2019] on these tasks are largely underestimated. We show that careful designs of model and training configurations are particularly important for these reasoning tasks testing systematic generalization. Guided by the intuition presented in Sec. 1.3, we revisit configurations such as layer sharing, relative positional embedding, basic scaling of the word and positional embeddings, and early stopping strategy, drastically improving the performance of the baseline transformers. We conduct experiments on five datasets: SCAN [Lake and

[1] For the full paper please see our work "The Devil is in the Detail: Simple Tricks Improve Systematic Generalization of Transformers" [Csordás et al., 2021]

Baroni, 2018], CFQ [Keysers et al., 2020], PCFG [Hupkes et al., 2020], COGS [Kim and Linzen, 2020], and Mathematic dataset [Saxton et al., 2019]. In particular, our new models improve the accuracy on the PCFG productivity split from 50% to 85%, on the systematicity split from 72% to 96%, and on COGS from 35% to 81% over existing baselines. On the SCAN dataset, we show that our models with relative positional embedding largely mitigate the so-called end-of-sentence (EOS) decision problem [Newman et al., 2020], achieving 100% accuracy on the length split with a cutoff at 26.

Also importantly, we show that despite these dramatic performance gaps, all these models perform equally well on IID validation datasets. The consequence of this observation is the need for proper generalization validation sets for developing neural networks for systematic generalization.

We thoroughly discuss guidelines that empirically yield good performance across various datasets, and we release the code[2] to make our results reproducible.

4.1 Datasets and Model Architectures for Systematic Generalization

Here we describe the five datasets and specify the transformer model variants we use in our experiments. The selected datasets include both the already popular ones and the recently proposed ones. The statistics of the datasets can be found in Tab. C.7 in the appendix.

4.1.1 Datasets

Many datasets in the language domain have been proposed to test systematic generalization. All datasets that we consider here can be formulated as a sequence-to-sequence mapping task [Sutskever et al., 2014; Graves, 2012]. Common to all these datasets, the test set is sampled from a distribution which is systematically different from the one for training: for example, the test set might systematically contain longer sequences, new combinations, or deeper compositions of known rules. We call this split the *generalization split*. Most of the datasets also come with a conventional split, where the train and test (and validation, if available) sets are independently and identically distributed samples. We call this the *IID split*. In this paper, we consider the following five datasets:

[2] `https://github.com/robertcsordas/transformer_generalization`

SCAN [Lake and Baroni, 2018]. The task consists of mapping a sentence in natural language into a sequence of commands simulating navigation in a grid world. The commands are compositional: e.g. an input `jump twice` should be translated to `JUMP JUMP`. It comes with multiple data splits: in addition to the "simple" IID split, in the "length" split, the training sequences are shorter than test ones, and in the "add primitive" splits, some commands are presented in the training set only in isolation, without being composed with others. The test set focuses on these excluded combinations.

CFQ [Keysers et al., 2020]. The task consists of translating a question in natural language into a Freebase SPARQL query. For example, `Was M0 a director and producer of M1` should be translated to `SELECT count(*) WHERE {M0 ns:film.director.film M1 . M0 ns:film.producer.film | ns:film.production_company.films M1}`. The authors introduce splits based on "compound divergence" which measures the difference between the parse trees in the different data splits. The authors experimentally show that it is well correlated with generalization difficulty. It also comes with a length-based split.

PCFG [Hupkes et al., 2020]. The task consists of list manipulations and operations that should be executed. For example, `reverse copy O14 O4 C12 J14 W3` should be translated to `W3 J14 C12 O4 O14`. It comes with different splits for testing different aspects of generalization. In this work, we focus on the "productivity" split, which focuses on generalization to longer sequences, and on the "systematicity" split, which focuses on recombining constituents in novel ways.

COGS [Kim and Linzen, 2020]. The task consists of semantic parsing that maps an English sentence to a logical form. For example, `The puppy slept.` should be translated to `* puppy ( x _ 1 ) ; sleep . agent ( x _ 2, x _ 1 )`. It comes with a single split, with a training, IID validation, and OOD generalization testing set.

Mathematics Dataset [Saxton et al., 2019]. The task consists of textual math questions at high school level, e.g. `What is -5 - 110911?` should be translated to `-110916`. The data is split into different subsets by the problem category, called modules. Some of them come with an extrapolation set, designed to measure generalization. The amount of total data is very large and thus expensive to train on, but different modules can be studied individually. We focus on "add_or_sub" and "place_value" modules.

4.1.2 Model Architectures

We focus our analysis on two transformer architectures: standard transformers [Vaswani et al., 2017] and universal transformers [Dehghani et al., 2019], and in both cases with absolute or relative positional embedding [Dai et al., 2019]. Our universal transformer variants are simply transformers with shared weights between layers, without adaptive computation time [Schmidhuber, 2012; Graves, 2016] and timestep embedding. Positional embeddings are only added to the first layer.

Universal Transformers are particularly relevant for reasoning and algorithmic tasks. For example, if we assume a task consisting of executing a sequence of operations, a regular transformer will learn successive operations in successive layers with separate weights. In consequence, if only some particular orderings of the operations are seen during training, each layer will only learn a subset of the operations, and thus it will be impossible for them to recombine operations in an arbitrary order. Moreover, if the same operation has to be reused multiple times, the network has to re-learn it, which is harmful for systematic generalization and reduces the data efficiency of the model [Csordás et al., 2021]. Universal Transformers have the potential to overcome this limitation: sharing the weights between each layer makes it possible to reuse existing knowledge from different compositions. On the downside, the Universal Transformer's capacity can be limited because of the weight sharing.

4.2 Improving Transformers on Systematic Generalization

In this section, we present methods that greatly improve transformers on systematic generalization tasks, while they could be considered as details in standard tasks. For each method, we provide experimental evidence on a few representative datasets. In Sec. 4.3, we apply these findings to all datasets.

	ℓ (length cutoff)	22	24	25	26	27	28	30	32	33	36	40
Reference	`+EOS`	0.00	0.05	0.04	0.00	0.09	0.00	0.09	0.35	0.00	*0.00*	0.00
	`+EOS+Oracle`	*0.53*	*0.51*	*0.69*	*0.76*	*0.74*	*0.57*	*0.78*	*0.66*	*0.77*	***1.00***	*0.97*
	`-EOS+Oracle`	***0.58***	***0.54***	*0.67*	*0.82*	*0.88*	*0.85*	*0.89*	*0.82*	***1.00***	***1.00***	***1.00***
Ours (+EOS)	Trafo	0.00	0.04	0.19	0.29	0.30	0.08	0.24	0.36	0.00	0.00	0.00
	+ Relative PE	0.20	0.12	0.31	0.61	**1.00**	**1.00**	**1.00**	0.94	**1.00**	**1.00**	**1.00**
	Universal Trafo	0.02	0.05	0.14	0.21	0.26	0.00	0.06	0.35	0.00	0.00	0.00
	+ Relative PE	0.20	0.12	**0.71**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**

Table 4.1: Exact match accuracies on length splits with different cutoffs. The reported results are the median of 5 runs. Trafo denotes transformers. The numbers in rows `+EOS+Oracle` and `-EOS+Oracle` are taken from Newman et al. [2020] as reference numbers, but they cannot be compared with others as they are evaluated with oracle length. Our models use different hyperparameters compared to theirs. We refer to Sec. 4.2.1 for details.

4.2.1 Addressing the EOS Decision Problem with Relative Positional Embedding

The EOS decision problem. A thorough analysis by Newman et al. [2020] highlights that LSTMs and transformers struggle to generalize to longer output lengths than they are trained for. Specifically, it is shown that the decision on when to end the sequence (the EOS decision) often overfits to the specific positions observed in the train set. To measure whether the models are otherwise able to solve the task, they conduct a so-called *oracle evaluation*: they ignore the EOS token during evaluation and use the ground-truth sequence length to stop decoding. The performance with this evaluation mode is much better, which illustrates that the problem is in fact the EOS decision. More surprisingly, if the model is trained without EOS token as part of the output vocabulary (thus, it can only be evaluated in oracle mode), the performance is further improved. It is concluded that teaching the model when to end the sequence has undesirable side effects on the model's length generalization ability.

We show that the main cause of this EOS decision problem in the case of transformers is the absolute positional embedding. Generally speaking, the meaning of a word is rarely dependent on the word's absolute position in a document but depends on its neighbors. Motivated by this assumption, various relative positional embedding methods [Shaw et al., 2018a; Dai et al., 2019] have been proposed. Unfortunately, they have not been considered for systematic generalization in prior work (however, see Sec. 4.4), even though they are particularly relevant for that.

We test transformers with relative positional embedding in the form used in Transformer XL [Dai et al., 2019]. Since it is designed for auto-regressive models, we directly apply it in the decoder of our model, while for the encoder, we use a symmetrical variant of it (see Appendix C.3). The interface between encoder and decoder uses standard attention without any positional embedding.

Our experimental setting is similar to Newman et al. [2020]. The length split in the SCAN dataset restricts the length of the train samples to 22 tokens (the test set consists of samples with an output of more than 22 tokens). This removes some compositions from the train set entirely, which introduces additional difficulty to the task. 80% of the test set consists of these missing compositions. In order to mitigate the issue of unknown composition and focus purely on the length problem, Newman et al. [2020] resplit SCAN by introducing different length cutoffs and report the performance of each split. We test our models similarly. However, our preliminary experiments showed that the performance of the original model is additionally limited by being too shallow: it uses only 2 layers for both the encoder and decoder. We increase the number of layers to 3. To compensate for the increased number of parameters, we decrease the size of the feedforward layers from 1024 to 256. In total, this reduces the number of parameters by 30%. We train our models with Adam optimizer, a learning rate of 10^{-4}, batch size of 128 for 50k steps.

The results are shown in Tab. 4.1. To show that our changes of hyperparameters are not the main reason for the improved performance, we report the performance of our modified model without relative positional embedding (row Trafo). We also include the results of Newman et al. [2020] for reference. We report the performance of universal transformer models trained with identical hyperparameters. All our models are trained *to predict* the EOS token and are evaluated *without* oracle (+EOS configuration). It can be seen that both our standard and universal transformers with absolute positional embedding have near-zero accuracy for all length cutoffs, whereas models with relative positional embedding excel: they even outperform the models trained without EOS prediction and evaluated with the ground-truth length.

Although Tab. 4.1 highlights the advantages of relative positional embedding and shows that they can largely mitigate the EOS-overfitting issue, this does not mean that the problem of generalizing to longer sequences is fully solved. The suboptimal performance on short length cutoffs (22-25) indicates that the model finds it hard to zero-shot generalize to unseen compositions of specific rules. To improve these results further, research on models which assume analogies between rules and compositions is necessary, such that they can recombine known constituents without any training example.

Further benefits of relative positional embedding. In addition to the benefit highlighted in the previous paragraph, we found that models with relative positional embedding are easier to train in general. They converge faster (Fig. C.3 in the appendix) and are less sensitive to batch size (Tab. C.6 in the appendix). As another empirical finding, we note that relative transformers without shared layers sometimes catastrophically fail before reaching their final accuracy: the accuracy drops to 0, and it never recovers. We observed this with PCFG productivity split and the "Math: place_value" task. Reducing the number of parameters (either using Universal Transformers or reducing the state size) usually stabilizes the network.

4.2.2 Model Selection Should Be Done Carefully

The danger of early stopping. Another crucial aspect that greatly influences the generalization performance of transformers is model selection, in particular early stopping. In fact, on these datasets, it is a common practice to use only the IID split to tune hyperparameters or select models with early stopping (e.g. Kim and Linzen [2020]). However, since any reasonable models achieve nearly 100% accuracy on the IID validation set, there is no good reason to believe this to be a good practice for selecting models for generalization splits. To test this hypothesis, we train models on COGS dataset without early stopping, but with a fixed number of 50k training steps. The best model achieved a test accuracy of 81%, while the original performance by Kim and Linzen [2020] is 35%. Motivated by this huge performance gap, we had no other choice but to conduct an analysis on the generalization split to demonstrate the danger of early stopping and discrepancies between the performance on the IID and generalization split. The corresponding results are shown in Fig. 4.1 (further effect of embedding scaling is discussed in next Sec. 4.2.3) and Tab. 4.2. Following Kim and Linzen [2020], we measure the model's performance every 500 steps, and mark the point where early stopping with patience of 5 would pick the best performing model. It can be seen that, in some cases, the model chosen by early stopping is not even reaching half of the final generalization accuracy.

To confirm this observation in the exact setting of Kim and Linzen [2020], we also disabled early stopping in the original codebase [3], and observed that

[3] https://github.com/najoungkim/COGS

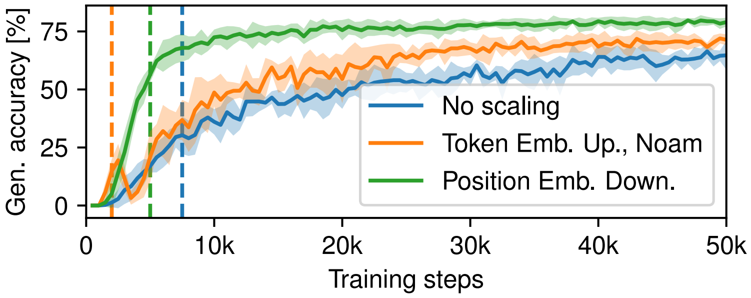


Figure 4.1: Generalization accuracy on COGS as a function of training steps for standard transformers with different embedding scaling schemes. The vertical lines show the median of the early stopping points for the five runs. The early stopping parameters are from Kim and Linzen [2020]. "Token Emb. Up., Noam" corresponds to the baseline configuration [Kim and Linzen, 2020]. See Sec. 4.2.3 for details on scaling.

		IID Validation	Gen. Test
COGS	TEU	**1.00** ± **0.00**	0.78 ± 0.03
	No scaling	**1.00** ± **0.00**	0.62 ± 0.06
	PED	**1.00** ± **0.00**	**0.80** ± **0.00**
PCFG	TEU	0.92 ± 0.07	0.47 ± 0.27
	No scaling	**0.97** ± **0.01**	0.63 ± 0.02
	PED	0.96 ± 0.01	**0.65** ± **0.03**

Table 4.2: Final IID validation and generalizations accuracy for COGS (50k steps) and PCFG Productivity set (300k steps) with different scaling (Sec. 4.2.3). Token Embedding Upscaling (TEU) is unstable on PCFG with our hyperparameters. Position Embedding Downscaling (PED) performs the best on both datasets.

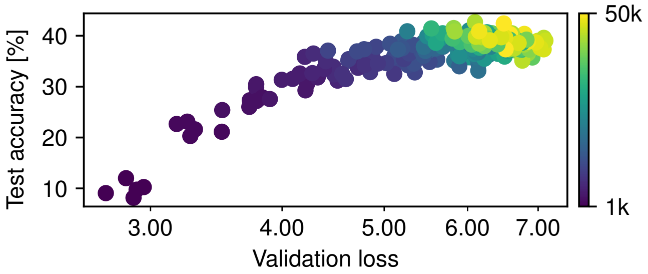


Figure 4.2: Relationship between validation loss and test accuracy (same distribution) on CFQ MCD 1 split for a relative transformer. The color shows the training step. Five runs are shown. The loss has a logarithmic scale. High accuracy corresponds to higher loss, which is unexpected. For detailed analysis, see Fig. C.2.

the accuracy improved to 65% without any other tricks. We discuss further performance improvements on COGS dataset in Sec. 4.3.4.

The lack of validation set for the generalization split. A general problem raised in the previous paragraph is the lack of a validation set to evaluate models for generalization. Most of the datasets come without a validation set for the generalization split (SCAN, COGS, and PCFG). Although CFQ comes with such a set, the authors argue that only the IID split should be used for hyperparameter search, and it is not clear what should be used for model development.

In order to test novel ideas, a way to gradually measure progress is necessary, such that the effect of changes can be evaluated. If the test set is used to develop the model, it implicitly risks overfitting to this test set. On the other hand, measuring the performance on the IID split does not necessarily provide any valuable information on the generalization performance on the systematically different test set (see Tab. 4.2). The IID accuracy of all the considered datasets is 100% (except on PCFG where it's also almost 100%); thus, no further improvement, nor the potential difference between the generalization performance of models can be measured. We further demonstrate this in Tab. 4.3. We only show datasets for which an IID validation set is available in the same split as the one reported in Tab. 4.4. With the exception of standard transformer on PCFG and the "place_value" module of the Mathematics dataset, all other validation accuracies are 100%, while their generalization accuracy vary wildly.

It would be beneficial if future datasets would have a validation and test set for both the IID *and* the generalization split. For the generalization split, the

	Transformer	Uni. Trafo	Rel. Trafo	Rel. Uni. Trafo
SCAN (lc = 26)	**1.0** ± **0.0** (0.30)	**1.0** ± **0.0** (0.21)	**1.0** ± **0.0** (0.72)	**1.0** ± **0.0** (1.00)
COGS	**1.0** ± **0.0** (0.80)	**1.0** ± **0.0** (0.78)	**1.0** ± **0.0** (0.81)	**1.0** ± **0.0** (0.77)
M add_or_sub	**1.0** ± **0.0** (0.89)	**1.0** ± **0.0** (0.94)	**1.0** ± **0.0** (0.91)	**1.0** ± **0.0** (0.97)
M place_value	0.8 ± 0.5 (0.12)	**1.0** ± **0.0** (0.20)	-	**1.0** ± **0.0** (0.75)

Table 4.3: IID validation accuracy for datasets where IID test set is available. CFQ and PCFG are not shown because they require the model to be trained on a separate, IID split. The other settings correspond to Tab. 4.4 in the main text. Generalization split test accuracies are shown in parenthesis for easy comparison. "M" denotes the Mathematics Dataset [Saxton et al., 2019]. SCAN with length cutoff of 26 shown.

test set could be designed to be more difficult than the validation set. In this way, the validation set can be used to measure progress during development, but overfitting to it would prevent the model from generalizing well to the test set. Such a division can be easily done on the splits to test productivity. For other types of generalization, we could use multiple datasets sharing the same generalization problem. Some of them could be dedicated for development and others for testing.

Intriguing relationship between generalization accuracy and loss. Finally, we also note the importance of using accuracy (instead of loss) as the model selection criterion. We find that generalization accuracy and loss do not necessarily correlate. Despite this, sometimes, model selection based on the loss is reported in practice e.g. by Kim and Linzen [2020]. Examples of this undesirable behavior are shown on Fig. 4.2 for CFQ and on Fig. C.1 in the appendix for COGS dataset. On these datasets, the loss and accuracy on the generalization split both grow during training. We conducted an analysis to understand the cause of this surprising phenomenon; we find that the total loss grows because the loss of the samples with incorrect output increases more than it improves on the correct ones. For the corresponding experimental results, we refer to Fig. C.2 in the Appendix. We conclude that even if a validation set is available for the generalization split, it would be crucial to use the *accuracy instead of the loss* for early stopping and hyperparameter tuning.

Finally, on PCFG dataset, we observed epoch-wise double descent phenomenon [Nakkiran et al., 2019], as shown in Fig. 4.3. This can lead to equally

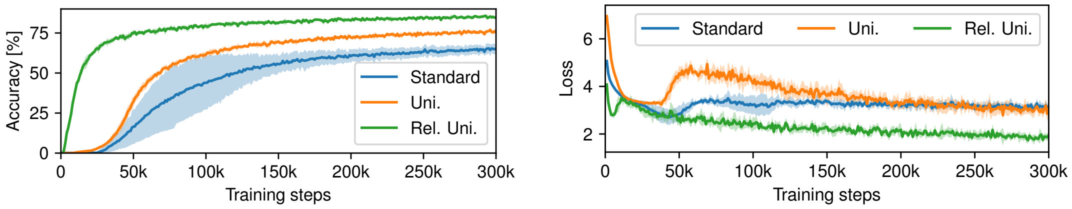


Figure 4.3: Test loss and accuracy on PCFG during training. The loss exhibits an epoch-wise double descent phenomenon [Nakkiran et al., 2019], while the accuracy increases monotonically. Standard transformer with PED (Sec. 4.2.3), universal transformer with absolute, and relative positional embeddings are shown.

problematic results if the loss is used for model selection or tuning.

4.2.3 Large Impacts of Embedding Scaling

The last surprising detail that greatly influences the generalization performance of transformers is the choice of embedding scaling scheme. This is especially important for transformers with absolute positional embedding, where the word and positional embedding have to be combined. We experimented with the following scaling schemes:

1. Token Embedding Upscaling (TEU). This is the standard scaling used by Vaswani et al. [2017]. It uses Glorot initialization [Glorot and Bengio, 2010] for word embeddings. However, the range of sinusoidal positional embedding is always in $[-1, 1]$. Since the positional embedding is directly added to the word embeddings, this discrepancy can make the model untrainable. Thus, the authors upscale the word embeddings by $\sqrt{d_{\text{model}}}$ where d_{model} is the embedding size. OpenNMT[4], the framework used for the baseline models for PCFG and COGS datasets respectively by Hupkes et al. [2020] and Kim and Linzen [2020], also uses this scaling scheme.

2. No scaling. It initializes the word embedding with $\mathcal{N}(0, 1)$ (normal distribution with mean 0 and standard deviation of 1). Positional embeddings are added without scaling.

3. Position Embedding Downscaling (PED), which uses Kaiming initialization [He et al., 2015], and scales the positional embeddings by $\frac{1}{\sqrt{d_{\text{model}}}}$.

[4] `https://opennmt.net/`

The PED differs from TEU used by Vaswani et al. [2017] in two ways: instead of scaling the embedding up, PED scales the positional embedding down and uses Kaiming instead of Glorot initialization. The magnitude of the embeddings should not depend on the number of words in the vocabulary but on the embedding dimension.

Tab. 4.2 shows the results. Although the "no scaling" variant is better than TEU on the PCFG test set, it is worse on the COGS test set. PED performs consistently the best on both datasets. Importantly, the gap between the best and worst configurations is large on the test sets. The choice of scaling thus also contributes in the large improvements we report over the existing baselines.

4.3 Results Across Different Datasets

In this section, we apply the methods we illustrated in the previous section across different datasets. Tab. 4.4 provides an overview of all improvements we obtain on all considered datasets. Unless reported otherwise, all results are the mean and standard deviation of 5 different random seeds. If multiple embedding scaling schemes are available, we pick the best performing one for a fair comparison. Transformer variants with relative positional embedding outperform the absolute variants on almost all tested datasets. Except for COGS and CFQ MCD 1, the universal variants outperform the standard ones. In the following, we discuss and highlight the improvements we obtained for each individual dataset.

4.3.1 SCAN

We focus on the **length split** of the dataset. We show that it is possible to mitigate the effect of overfitting to the absolute position of the EOS token by using relative positional embedding. We have already discussed the details in Sec. 4.2.1 and Tab. 4.1.

4.3.2 CFQ

On the **output length split** of CFQ, our universal transformer with absolute positional embedding achieves significantly better performance than the one reported by Keysers et al. [2020]: 77% versus $\sim 66\%$[5]. Here, we were unable to identify the exact reason for this large improvement. The only architectural difference between the models is that ours does not make use of any timestep

[5] As Keysers et al. [2020] only report charts, the exact value is unknown.

	Trafo	Uni. Trafo	Rel. Trafo	Rel. Uni. Trafo	Prior Work
SCAN (cutoff=26)	0.30 ± 0.02	0.21 ± 0.01	0.72 ± 0.21	**1.00** ± **0.00**	0.00[1]
CFQ Output length	0.57 ± 0.00	0.77 ± 0.02	0.64 ± 0.06	**0.81** ± **0.01**	∼ 0.66[2]
CFQ MCD 1	**0.40** ± **0.01**	0.39 ± 0.03	0.39 ± 0.01	0.39 ± 0.04	0.37 ± 0.02[3]
CFQ MCD 2	0.10 ± 0.01	0.09 ± 0.02	0.09 ± 0.01	**0.10** ± **0.02**	0.08 ± 0.02[3]
CFQ MCD 3	0.11 ± 0.00	0.11 ± 0.01	0.11 ± 0.01	**0.11** ± **0.03**	0.11 ± 0.00[3]
CFQ MCD mean	0.20 ± 0.14	0.20 ± 0.14	0.20 ± 0.14	**0.20** ± **0.14**	0.19 ± 0.01[2]
PCFG Prod. split	0.65 ± 0.03	0.78 ± 0.01	-	**0.85** ± **0.01**	0.50 ± 0.02[4]
PCFG Sys. split	0.87 ± 0.01	0.93 ± 0.01	0.89 ± 0.02	**0.96** ± **0.01**	0.72 ± 0.00[4]
COGS	0.80 ± 0.00	0.78 ± 0.03	**0.81** ± **0.01**	0.77 ± 0.01	0.35 ± 0.06[5]
Math: add_or_sub	0.89 ± 0.01	0.94 ± 0.01	0.91 ± 0.03	**0.97** ± **0.01**	∼ 0.91[6]*
Math: place_value	0.12 ± 0.07	0.20 ± 0.02	-	**0.75** ± **0.10**	∼ 0.69[6]*

Table 4.4: Test accuracy of different transformer (Trafo) variants on the considered datasets. See Sec. 4.3 for details. The last column shows previously reported accuracies. References: [1] Newman et al. [2020], [2] Keysers et al. [2020], [3] `https://github.com/google-research/google-research/tree/master/cfq`, [4] Hupkes et al. [2020], [5] Kim and Linzen [2020], [6] Saxton et al. [2019]. Results marked with ∗ cannot be directly compared due to different training setups. ∼ denotes approximative numbers read from charts reported in previous works.

(i.e. layer ID) embedding. Also, the positional embedding is only injected to the first layer in case of absolute positional embeddings (Sec. 4.1.2). The relative positional embedding variant performs even better, achieving 81%. This confirms the importance of using relative positional embedding as a default choice for length generalization tasks, as we also demonstrated on SCAN in Sec. 4.2.1.

On the **MCD splits**, our results slightly outperform the baseline by Keysers et al. [2020], as shown in Tab. 4.4. Relative universal transformers perform marginally better than all other variants, except for MCD 1 split, where the standard transformer wins with a slight margin. We use hyperparameters from Keysers et al. [2020]. We report performance after 35k training steps.

4.3.3 PCFG

The performance of different models on the PCFG dataset is shown on Tab. 4.4. First, simply by increasing the number of training epochs from 25, used by Hupkes et al. [2020], to $\sim$237 (300k steps), our model achieves 65% on the **productivity split** compared to the 50% reported by Hupkes et al. [2020] and 87% compared to 72% on the **systematicity split**. Furthermore, we found that universal transformers with relative positional embeddings further improve performance to a large extent, achieving 85% final performance on the **productivity** and 96% on the **systematicity split**. We experienced instabilities while training transformers with relative positional embeddings on the productivity split; therefore, the corresponding numbers are omitted in Tab. 4.4 and Fig. C.3 in the Appendix.

4.3.4 COGS

On COGS, our best model achieves the generalization accuracy of 81% which greatly outperforms the 35% accuracy reported by Kim and Linzen [2020]. This result obtained by simple tricks is competitive compared to the state-of-the-art performance of 83% reported by Akyürek and Andreas [2021][6]. As we discussed in Sec. 4.2.2, just by removing early stopping in the setting of Kim and Linzen [2020], the performance improves to 65%. Moreover, the baseline with early stopping is very sensitive to the random seed and even sensitive to the GPU type it is run on. Changing the seed in the official repository from 1 to 2 causes a dramatic performance drop with a 2.5% final accuracy. By changing the scaling of embeddings (Sec. 4.2.3), disabling label smoothing, fixing the learning rate

[6] Akyürek and Andreas [2021] was published on arXiv on June 7 and later at ACL 2021. We were unaware of this work at the time of submission to EMNLP 2021 (May 17, 2021).

to 10^{-4}, we achieved 81% generalization accuracy, which is stable over multiple random seeds.

Tab. 4.4 compares different model variants. Standard transformers with absolute and relative positional encoding perform similarly, with the relative positional variant having a slight advantage. Here universal transformers perform slightly worse.

4.3.5 Mathematics Dataset

We also test our approaches on subsets of Mathematics Dataset [Saxton et al., 2019]. Since training models on the whole dataset is too resource-demanding, we only conduct experiments on two subsets: "place_value" and "add_or_sub".

The results are shown in Tab. 4.4. While we cannot directly compare our numbers with those reported by Saxton et al. [2019] (a single model is jointly trained on the whole dataset there), our results show that relative positional embedding is advantageous for the generalization ability on both subsets.

4.4 Related Work

The study of generalization ability of neural networks at different stages of training has been a general topic of interest [Nakkiran et al., 2019; Roelofs, 2019]. Our analysis has shown that this question is particularly relevant to the problem of systematic generalization, as demonstrated by large performance gaps in our experiments, which has not been discussed in prior work.

Prior work proposed several sophisticated initialization methods for transformers [Zhang et al., 2019; Zhu et al., 2021], e.g. with a purpose of removing the layer normalization components [Huang et al., 2020]. While our work only revisited basic scaling methods, we demonstrated their particular importance for systematic generalization.

In recent work,[7] Ontañón et al. [2021] have also focused on improving the compositional generalization abilities of transformers. In addition to relative positional encodings and universal transformers, novel architectural changes such as "copy decoder" as well as dataset-specific "intermediate representations" [Herzig et al., 2021] have been studied. However, other aspects we found crucial, such as early stopping, scaling of the positional embeddings, and the validation set issues have not been considered. In consequence, our models achieve

[7] Our work was submitted to EMNLP 2021 on May 17, 2021 and has been under the anonymity period until Aug. 25. Ontañón et al. [2021] appeared on arXiv on Aug. 9, 2021.

substantially higher performance than the best results reported by Ontañón et al. [2021] across all standard datasets: PCFG, COGS, and CFQ (without intermediate representations).

Finally, our study focused on the basic transformer architectures. However, the *details* discussed above in the context of algorithmic tasks should also be relevant for other transformer variants and fast weight programmers [Schmidhuber, 1992a; Schlag et al., 2021; Irie et al., 2021], as well as other architectures specifically designed for algorithmic reasoning [Graves et al., 2016; Kaiser and Sutskever, 2016; Csordás and Schmidhuber, 2019; Freivalds et al., 2019].

4.5 Conclusion

In this work, we show that the performance of transformer architectures on many recently proposed datasets for systematic generalization can be greatly improved by revisiting basic model and training configurations. Model variants with relative positional embedding often outperform those with absolute positional embedding. They also mitigate the EOS decision problem, an important problem previously found by Newman et al. [2020] when considering the length generalization of neural networks. This allows us to focus on the problem of compositions in the future, which is the remaining problem for the length generalization.

We also demonstrated that reconsidering early stopping and embedding scaling can greatly improve baseline transformers, in particular on the COGS and PCFG datasets. These results shed light on the discrepancy between the model performance on the IID validation set and the test accuracy on the systematically different generalization split. As consequence, currently common practice of validating models on the IID dataset is problematic. We conclude that the community should discuss proper ways to develop models for systematic generalization. In particular, we hope that our work clearly demonstrated the necessity of a validation set for systematic generalization in order to establish strong baselines and to avoid a false sense of progress.

Chapter 5

Achieving Length Generalization with Transformers[1]

Despite the significant improvements we saw in Sec. 4, our improved shared layer transformer with relative positional encodings still fails to achieve length generalization on simple algorithmic tasks. In this work, we ask the question: which type of architectural inductive bias encourages the training process to select "good" solutions that have good productivity?

As a reminder, popular transformers [Vaswani et al., 2017] also often fail to generalize on algorithmic tasks (e.g. Liska et al. [2018]; Dubois et al. [2020]; Chaabouni et al. [2021]; Csordás et al. [2021]; Ontañón et al. [2021]), even on tasks with intuitive solutions that can be simply expressed in terms of transformer attention patterns. Given an input sequence of length N and a transformer encoder of depth T, solving an algorithmic task often consists of routing the relevant information to the right node/operation at the right time in the T-by-N grid represented by transformer columns (illustrated in Fig. 5.2/Left). Effectively, the task is to learn to draw an *adaptive control flow* on the canvas of transformer columns. In fact, recent work by Weiss et al. [2021] introduced a programming language called RASP, which is specifically designed to express solutions to sequence processing problems, and which has a direct equivalent to the operations in transformer encoders. However, it is shown that transformers learn solutions expressed in RASP only through intermediate supervision of attention patterns, and sometimes even such supervision fails. Generally speaking, transformers fail to find easily interpretable and/or symbolic solutions to algorithmic tasks. We conversely hypothesize that attention-based NNs that are able to find

[1] For the full paper please see our work "The Neural Data Router: Adaptive Control Flow in Transformers Improves Systematic Generalization" [Csordás et al., 2022a]

intuitive solutions (achieving interpretable attention patterns) could improve systematic generalization.

Here we point out that regular transformers lack some basic ingredients for learning such "intuitive" solutions to algorithmic problems. As a remedy, we propose simple architectural modifications to help them learn data routing. As a first step towards validating our model, we focus on the popular length generalization task of compositional table lookup (CTL [Liska et al., 2018; Hupkes et al., 2019; Dubois et al., 2020]), as well as two more complex tasks: a simple arithmetic task and a variant of ListOps [Nangia and Bowman, 2018] designed to test the compositional generalization ability of NNs. Our novel Neural Data Router (NDR) achieves 100% generalization accuracy (never reported before) on the CTL task and obtains nearly perfect accuracy on both the proposed simple arithmetic and ListOps tasks. We show that the attention and gating patterns of NDR tend to be interpretable as plausible control flows.

5.1 Improving Transformers for Learning Adaptive Control Flow

We argue that the following components are needed to build transformers capable of learning adaptive control flow. **First**, as discussed in Sec. 1.3.2, composing known operations in an arbitrary order requires that all operations are available at every computational step. This can be easily achieved by sharing the weights of the layers, as is done in Universal Transformers [Dehghani et al., 2019]. **Second**, the network should be sufficiently deep, at least as deep as the deepest data dependency in the computational graph built from elementary operations (e.g., in the case of a parse tree, this is the depth of the tree). Otherwise, multiple operations must be fused into a single layer and hinder natural and elegant compositions (see Sec. 1.3.3). **Third**, inputs in some columns should remain unchanged until it is their turn to be processed. The regular transformer lacks a mechanism to skip the whole transformation step by simply copying the input to the next step/layer. We propose a special gating function, *copy gate*, to implement such a mechanism (Sec. 5.1.1). **Finally**, many algorithmic tasks require combining several local computations in the right order. This typically implies that attention should not focus on all possible matches at a given time, but only on the closest match. We propose and investigate a new type of attention with a corresponding inductive bias called *geometric attention* (Sec. 5.1.2). Using both the geometric attention and the copy gate, our model implements a "neural data routing mechanism", which can adaptively serialize the input problem. We refer to the resulting new transformer as Neural Data Router (NDR). In the experimental section (Sec. 5.2), we evaluate this model on three algorithmic tasks that require length generalization and demonstrate its effectiveness.

5.1.1 Copy Gate: Learning to Skip Operations (Vertical Flow)

Each layer of the regular transformer consists of one self-attention and one feed-forward block. The input of each of these blocks is directly connected to the corresponding output via a residual connection [Srivastava et al., 2015b,a; He et al., 2016]. However, such a connection does not allow for easily skipping the transformation of the entire layer and simply passing the unchanged input to the next layer. Here, we propose adding an explicit gate, which we call *copy gate*, to facilitate such a behavior.

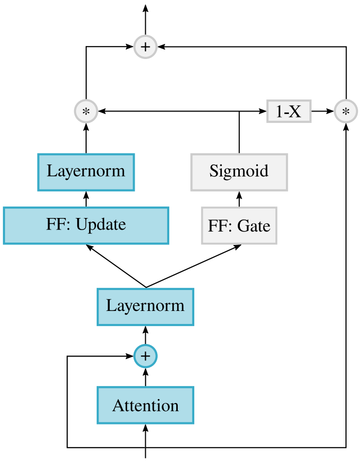


Figure 5.1: Structure of the transformer/NDR layer with a copy gate. The blue part corresponds to the standard transformer, except for the missing residual connection around the feedforward block ("FF: Update"). The gray part is the copy gate. The feedforward part corresponding to the gate is usually significantly smaller than the one used for the update.

We consider a T-layer (post-layernorm) transformer encoder and an input sequence of length N. Since each layer corresponds to one *computational step*, we often refer to a layer as a step t. We denote the transformer state of the column i in layer t as $\boldsymbol{h}^{(i,t)} = \mathbf{H}_{t,i} \in \mathbb{R}^d$ where d is the state size, and $\mathbf{H}_t \in \mathbb{R}^{N \times d}$ denotes the states of all N columns in layer t. In the copy gate-augmented transformer

(Fig. 5.1), each column i in layer $(t+1)$ processes the input $\mathbf{H}_t$ similarly to regular transformers:

$$\boldsymbol{a}^{(i,t+1)} = \mathrm{LayerNorm}(\mathrm{MultiHeadAttention}(\boldsymbol{h}^{(i,t)}, \mathbf{H}_t, \mathbf{H}_t) + \boldsymbol{h}^{(i,t)}) \tag{5.1}$$

$$\boldsymbol{u}^{(i,t+1)} = \mathrm{LayerNorm}(\mathrm{FFN}^{\mathrm{data}}(\boldsymbol{a}^{(i,t+1)})) \tag{5.2}$$

using the standard multi-head attention operation [Vaswani et al., 2017] with a query obtained from $\boldsymbol{h}^{(i,t)}$ and keys/values from $\mathbf{H}_t$, but the output is gated (using $\boldsymbol{g}^{(i,t+1)} \in \mathbb{R}^d$) as:

$$\boldsymbol{g}^{(i,t+1)} = \sigma(\mathrm{FFN}^{\mathrm{gate}}(\boldsymbol{a}^{(i,t+1)})) \tag{5.3}$$

$$\boldsymbol{h}^{(i,t+1)} = \boldsymbol{g}^{(i,t+1)} \odot \boldsymbol{u}^{(i,t+1)} + (1 - \boldsymbol{g}^{(i,t+1)}) \odot \boldsymbol{h}^{(i,t)} \tag{5.4}$$

We use the basic two-layer feedforward block [Vaswani et al., 2017] for both $\mathrm{FFN}^{\mathrm{data}}$ and $\mathrm{FFN}^{\mathrm{gate}}$ which transforms input $\boldsymbol{x} \in \mathbb{R}^d$ to:

$$\mathrm{FFN}(\boldsymbol{x}) = \boldsymbol{W}_2 \max(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1, 0) + \boldsymbol{b}_2 \tag{5.5}$$

but with separate parameters and different dimensionalities: for $\mathrm{FFN}^{\mathrm{data}}$ $\boldsymbol{W}_1^{\mathrm{data}} \in \mathbb{R}^{d_{\mathsf{FF}} \times d}$, $\boldsymbol{W}_2^{\mathrm{data}} \in \mathbb{R}^{d \times d_{\mathsf{FF}}}$, while for $\mathrm{FFN}^{\mathrm{gate}}$ $\boldsymbol{W}_1^{\mathrm{gate}}, \boldsymbol{W}_2^{\mathrm{gate}} \in \mathbb{R}^{d \times d}$, with biases $\boldsymbol{b}_1^{\mathrm{data}} \in \mathbb{R}^{d_{\mathsf{FF}}}$ and $\boldsymbol{b}_2^{\mathrm{data}}, \boldsymbol{b}_1^{\mathrm{gate}}, \boldsymbol{b}_2^{\mathrm{gate}} \in \mathbb{R}^d$.

When the gate is closed i.e. $\boldsymbol{g}^{(i,t+1)} = 0$ in Eq. 5.4, the entire transformation is skipped and the input is copied over to the next layer $\boldsymbol{h}^{(i,t+1)} = \boldsymbol{h}^{(i,t)}$. Crucially, we parameterize the gate (Eq. 5.3) as a function of the output of self-attention (Eq. 5.1), such that the decision to copy or transform the input for each column depends on the states of all columns. This is a crucial difference compared to previously proposed gatings in transformers, which are motivated solely by training stability [Parisotto et al., 2020] or by a common practice of convolution-based models [Chaabouni et al., 2021]. None of the previous approaches can implement the behavior of our copy gate (see Sec. 5.5 on related work).

The gate bias $\boldsymbol{b}_2^{\mathrm{gate}}$ is initialized to -3 [Hochreiter and Schmidhuber, 1997]. This ensures that no update is performed initially to create a better gradient flow between layers. It also encourages the model to skip layers unless they have an important contribution in the corresponding step.

5.1.2 Geometric Attention: Learning to Attend to the Closest Match (Horizontal Flow)

We propose *geometric attention* designed to attend to the closest matching element. Like in regular self-attention, given an input sequence $[\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(N)}]$

with $\boldsymbol{x}^{(i)} \in \mathbb{R}^{d_{\text{in}}}$, each input is projected to key $\boldsymbol{k}^{(i)} \in \mathbb{R}^{d_{\text{key}}}$, value $\boldsymbol{v}^{(i)} \in \mathbb{R}^{d_{\text{value}}}$, query $\boldsymbol{q}^{(i)} \in \mathbb{R}^{d_{\text{key}}}$ vectors, and the dot product is computed for each key/query combination. In our geometric attention, the dot product is followed by a sigmoid function to obtain a score between 0 and 1:

$$\boldsymbol{P}_{i,j} = \sigma(\boldsymbol{k}^{(j)\top}\boldsymbol{q}^{(i)}) \tag{5.6}$$

which will be treated as a probability of the key at (source) position j matching the query at (target) position i. These probabilities are finally converted to the attention scores $\boldsymbol{A}_{i,j}$ as follows:

$$\boldsymbol{A}_{i,j} = \boldsymbol{P}_{i,j} \prod_{k \in \mathbb{S}_{i,j}} (1 - \boldsymbol{P}_{i,k}) \tag{5.7}$$

where $\mathbb{S}_{i,j}$ denotes the set of all (source) indices that are closer to i than j is to i, and when two indices have the same distance to i, we consider the one which is to the right of i (i.e., greater than i) to be closer, i.e.,

$$\mathbb{S}_{i,j} = \begin{cases} k \in \{1, ..., N\} \setminus \{i, j\} : |i - k| < |i - j|, & \text{if } i < j \\ k \in \{1, ..., N\} \setminus \{i, j\} : |i - k| \leq |i - j|, & \text{if } j < i \end{cases} \tag{5.8}$$

In addition, we explicitly zero out the diagonal by setting $\boldsymbol{A}_{i,i} = 0$ for all $i = 1, ..., N$. The ordering of source indices is illustrated in Fig. 5.2/Right. The resulting scores $\boldsymbol{A}_{i,j}$ are the attention scores used to compute the weighted averages of the value vectors.

By using the terms $(1 - \boldsymbol{P}_{i,k})$ in Eq. 5.7, when there is a match, it downscales any other more distant matches. Two recent works [Brooks et al., 2021; Banino et al., 2021] use such a parameterized geometric distribution in the form of Eq. 5.7 (see Sec. 5.5 on related work).

The resulting attention function has a complexity of $O(N^2)$, similar to the regular self-attention used in transformers [Vaswani et al., 2017]. Eq. 5.7 can be implemented in a numerically stable way in log space. The products can then be calculated using cumulative sums, subtracting the elements for the correct indices in each position.

Directional encoding. In practice, we augment Eq. 5.6 with an additional *directional encoding*. In fact, the only positional information available in the geometric attention presented above is the ordering used to define the product in Eqs. 5.7-5.8. In practice, we found it crucial to augment the score computation

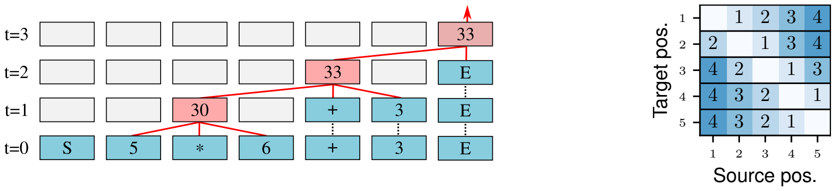


Figure 5.2: Left: an ideal sequence of computations in a transformer for an arithmetic expression. Right: ordering (numbers in the grid) of source positions used in geometric attention (Eq. 5.8; $N = 5$).

of Eq. 5.6 with additional *directional information*, encoded as a scalar $\boldsymbol{D}_{i,j} \in \mathbb{R}$ for each target/source position pair (i, j):

$$\boldsymbol{D}_{i,j} = \begin{cases} \boldsymbol{W}_{\text{LR}}\boldsymbol{h}^{(i)} + b_{\text{LR}}, & \text{if } i \leq j \\ \boldsymbol{W}_{\text{RL}}\boldsymbol{h}^{(i)} + b_{\text{RL}}, & \text{if } i > j \end{cases} \tag{5.9}$$

where $\boldsymbol{h}^{(i)} \in \mathbb{R}^d$ denotes the input/state at position i and $\boldsymbol{W}_{\text{LR}}, \boldsymbol{W}_{\text{RL}} \in \mathbb{R}^{1\times d}$, $b_{\text{LR}}, b_{\text{RL}} \in \mathbb{R}$ are trainable parameters. This directional information is integrated into the score computation of Eq. 5.6 as follows (akin to how Dai et al. [2019] introduce the relative positional encoding [Schmidhuber, 1992d] as an extra term in the computation of attention scores):

$$\boldsymbol{P}_{i,j} = \sigma\big(\alpha\big(\boldsymbol{W}_q\boldsymbol{h}^{(i)} + \boldsymbol{b}_q\big)^\top \boldsymbol{W}_{k,E}\boldsymbol{h}^{(j)} + \beta\boldsymbol{D}_{i,j} + \gamma\big) \tag{5.10}$$

where the matrix $\boldsymbol{W}_q \in \mathbb{R}^{d_{\text{head}}\times d}$ maps the states to queries, $\boldsymbol{b}_q \in \mathbb{R}^{d_{\text{head}}}$ is a bias for queries, $\boldsymbol{W}_{k,E} \in \mathbb{R}^{d_{\text{head}}\times d}$ maps states to keys (we note that d_{head} is typically the size of the key, query and value vectors for each head, $d_{\text{head}} = \frac{d}{n_{\text{heads}}}$), and $\alpha, \beta, \gamma \in \mathbb{R}$ are learned scaling coefficients and bias, initialized to $\alpha = \frac{1}{\sqrt{d_{\text{head}}}}, \beta = 1, \gamma = 0$. Using this additional directional information, each query (position i) can potentially learn to restrict its attention to either the left or right side.

5.2 Experiments

We evaluate the proposed methods on three tasks: the compositional table lookup [Liska et al., 2018; Hupkes et al., 2019], a custom variant of ListOps [Nangia and Bowman, 2018], and a simple arithmetic task, which we propose. In all cases, the task is designed to test the compositional generalization ability of NNs: the model has to learn to apply operations seen during training in a longer/deeper compositional way (productivity). Further experimental details for each task can be found in the Appendix D.3.

5.2.1 Compositional Table Lookup

Task. The compositional table lookup task [Liska et al., 2018; Hupkes et al., 2019; Dubois et al., 2020] is constructed based on a set of symbols and unary functions defined over these symbols. Each example in the task is defined by one input symbol and a list of functions to be applied sequentially, that is, the first function is applied to the input symbol and the resulting output becomes the input to the second function, and so forth. There are eight possible symbols. Each symbol is traditionally represented by a 3-bit bitstring [Liska et al., 2018]. However, in practice, they are simply processed as one token [Dubois et al., 2020]. The functions are bijective and randomly generated. Each function is represented by a letter. An example input is '`101 d a b`', which corresponds to the expression $b(a(d(101)))$; the model has to predict the correct output symbol. We note that there exists a sequence-to-sequence variant of this task [Dubois et al., 2020] where the model has to predict all intermediate steps (thus trained with intermediate supervision). We directly predict the final output. An ideal model should be able to solve this task independently of the presentation order, that is, it should not matter whether the task is encoded as '`101 d a b`' or '`b a d 101`'. We thus study both forward (former) and backward (latter) variants of the task. To evaluate systematic generalization, the train/valid/test sets reflect different numbers of compositions: samples with 1-5/6-8/9-10 operations, respectively. To the best of our knowledge, no previous work has reported perfect accuracy on this task using an NN. We refer the reader to Sec. 5.5 for further details on the previous work.

Results. We consider five different baselines: an LSTM [Hochreiter and Schmidhuber, 1997], bidirectional LSTM [Graves et al., 2005], DNC [Graves et al., 2016; Csordás and Schmidhuber, 2019], Universal Transformers [Vaswani et al., 2017; Dehghani et al., 2019], and its relative position variants [Csordás et al., 2021]. For transformers, the prediction is based on the last column in the final layer (we conduct an ablation study on this choice in Appendix D.1). The hyperparameters used for each model can be found in Tab. D.3 in the Appendix. The main results on this task are shown in Tab. 5.1. The LSTM and DNC perform well in the forward variant, achieving perfect generalization for longer sequences, but fail on the backward variant. This is not surprising since in the forward case, the input symbols are presented in the "right" processing order to the LSTM. As expected, the bidirectional LSTM performs well in both presentation orders, since one of its processing directions is always aligned with the order of computation. However, for an arbitrary task, the order of processing is not given. For example, for ListOps (Sec. 5.2.3), the processing should start from the deepest point in the parse tree, which is probably somewhere in the middle of the sequence. Experiments on other tasks (Sec. 5.2.2 and 5.2.3) that require arbitrary processing orders show that bidirectional LSTMs do not generalize well in such tasks. This is not satisfactory since our goal is to create a generic architecture that can solve arbitrary problems with an arbitrary underlying input processing order. While the transformer seems to be a good candidate for learning problem-dependent processing orders, the baseline transformer variants fail to generalize in this task in both directions.

By introducing the copy gate (Sec. 5.1.1), the relative transformer can solve the forward task, but not the backward one. Our analysis showed that the network learns to attend to the last operation based on relative position information. Since the result is read from the last column, this position changes with the sequence length. The model thus fails to generalize to such arbitrary offsets. To address this issue, we introduce a simple mechanism to let the model choose between absolute and relative positional encodings at each position (see Appendix D.2). The resulting model effectively manages to use the absolute position for the prediction and performs well in both directions. However, such a combination of absolute/relative positional encoding might be an overly specific bias. A more generic solution, geometric attention (Sec. 5.1.2), also achieved perfect generalization and was found to be easier to train. We present the corresponding visualization of our model in Sec. 5.3.

Are all those layers needed? In Sec. 5.1, we hypothesized that decomposition of the problem into its elementary operations is a necessary property of a model which generalizes. This motivated us to configure our models to have at least as many layers as the depth of the computation involved, plus a few additional layers for writing the output and for gathering an overview of the problem at the beginning. We assumed that in such a model with a sufficient number of layers, each layer learns the underlying "elementary" operation. Therefore, the resulting models are deeper than those typically used in the literature for similar tasks [Keysers et al., 2020; Tay et al., 2021]. Here we provide an ablation study to demonstrate that such depths are effectively necessary for generalization. We

	IID		Longer	
Model	Forward	Backward	Forward	Backward
LSTM	**1.00 ± 0.00**	0.59 ± 0.03	**1.00 ± 0.00**	0.22 ± 0.03
Bidirectional LSTM	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**
DNC	**1.00 ± 0.00**	0.57 ± 0.06	**1.00 ± 0.00**	0.18 ± 0.02
Transformer	**1.00 ± 0.00**	0.82 ± 0.39	0.13 ± 0.01	0.12 ± 0.01
+ rel	**1.00 ± 0.00**	**1.00 ± 0.00**	0.23 ± 0.05	0.13 ± 0.01
+ rel + gate	**1.00 ± 0.00**	**1.00 ± 0.00**	**0.99 ± 0.01**	0.19 ± 0.04
+ abs/rel + gate	**1.00 ± 0.00**	**1.00 ± 0.00**	**0.98 ± 0.02**	**0.98 ± 0.03**
+ geom.	0.96 ± 0.04	0.93 ± 0.06	0.16 ± 0.02	0.15 ± 0.02
+ geom. + gate (NDR)	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**

Table 5.1: Accuracy on **compositional table lookup** dataset.

measure the IID and generalization performance with various numbers of layers on the compositional table lookup dataset. Since our test set on the CTL task consists of up to 10 function applications, it should require about 12 layers according to our hypothesis. Tab. 5.2 shows the results. We clearly observe that, while the shallow models can also solve the IID split, only the deep models generalize to the longer problems (here the 12-layer model generalizes almost perfectly, but the 10-layer one does not). This supports our hypothesis about how the shared-layer transformers solve the problem, and demonstrates a direct consequence of the fusing effect discussed in Sec. 1.3.3.

	IID		Test	
n_{layers}	Forward	Backward	Forward	Backward
14	1.00 ± 0.00	1.00 ± 0.00	**1.00 ± 0.00**	**1.00 ± 0.00**
12	1.00 ± 0.00	1.00 ± 0.00	**1.00 ± 0.00**	**0.99 ± 0.02**
10	1.00 ± 0.00	1.00 ± 0.00	0.75 ± 0.04	0.62 ± 0.05
8	1.00 ± 0.00	1.00 ± 0.00	0.23 ± 0.02	0.24 ± 0.03
6	1.00 ± 0.00	0.96 ± 0.03	0.22 ± 0.05	0.15 ± 0.01
4	0.96 ± 0.04	0.68 ± 0.11	0.14 ± 0.01	0.13 ± 0.01

Table 5.2: The performance of NDR on the **compositional table lookup** dataset, with different number of layers.

5.2.2 Simple Arithmetic

In order to validate the success of the proposed model on a task that involves more complex data flows and operations, we propose the *simple arithmetic* task.

Task. The task is to execute an arithmetic expression consisting of nested modulo 10 additions and multiplications. This requires the model to process tree-structured data flows, which is presumably more difficult than the sequential processing required for the CTL task. Each operation is surrounded by brackets so that the boundaries of the operations are easy to determine. For example '`((4*7)+2)`' should evaluate to '`0`' (30 modulo 10). Expressions are generated randomly. The tree depth is up to 5 for the training set, 6 for the validation set, and 7-8 for the test set. The depth is measured as the number of operations, ignoring the leaves, so the example above has a depth of 2. The sequence length is limited to at most 50 tokens.

Results. Tab. 5.3 shows the results. All considered models perform well on the IID validation data, but none except the NDR performs well on the generalization test set, which achieves a near-perfect accuracy of 98%. We also note that the NDR learns very quickly: while all other models require about 200 K steps to converge, the NDR achieves near-perfect accuracy after 50 K steps of training.

	IID (1..5)	Test (7..8)	
	200 K	200 K	50 K
LSTM	**0.99** ± **0.00**	0.74 ± 0.02	0.72 ± 0.01
Bidirectional LSTM	**0.98** ± **0.01**	0.82 ± 0.06	0.80 ± 0.04
Transformer	**0.98** ± **0.01**	0.47 ± 0.01	0.29 ± 0.01
+ rel	**1.00** ± **0.00**	0.77 ± 0.04	0.40 ± 0.05
+ abs/rel + gate	**1.00** ± **0.01**	0.80 ± 0.16	0.73 ± 0.15
+ geom. att. + gate (NDR)	**1.00** ± **0.00**	**0.98** ± **0.01**	**0.98** ± **0.01**

Table 5.3: Performance of different models on the **simple arithmetic** dataset. All models are trained for 200 K iterations, except for the NDR, which we stop training at 100 K. We also report the performance after 50 K iterations, where it can be seen that NDR converges significantly faster than the others.

5.2.3 ListOps

We also evaluate our model on a variant of the ListOps task [Nangia and Bowman, 2018] which is a popular task commonly used to evaluate the parsing abilities of NNs [Havrylov et al., 2019; Shen et al., 2019; Xiong et al., 2021; Tay et al., 2021; Irie et al., 2021]. Some special architectures, such as the one presented by Chowdhury and Caragea [2021] can almost perfectly generalize to longer sequences on this task. However, as far as we know, no transformer variant has been reported to be fully successful.

Task. The task consists of executing nested list operations written in prefix notation. All operations have a list of arguments that can be either a digit (from 0 to 9) or recursively another operation with its own list of arguments. The operations are min, max, median, and sum. The sum is modulo 10, and the median is followed by the floor function such that the output of any operation lies between 0 and 9. For example: `[MED 4 8 5 [MAX 8 4 9 ] ]` should return 6. There are two well-known variants: the original variant of Nangia and Bowman [2018] and the "Long Range Arena" variant by Tay et al. [2021] which have different maximum numbers of arguments in each function and maximum sequence lengths. In both variants, there is no strict control of the depth of data samples: there is simply a certain pre-defined probability that each argument in the list is expanded into another list (which may increase the tree depth). This is not suitable for evaluating systematic generalization in terms of compositionality (over the depth of the problem). We propose instead to generate clean train, valid, and test splits with disjoint depths: up to depth 5 for training, depth 6 for validation, and depths 7 and 8 for test. Importantly, we make sure that a depth-K sample effectively requires computation until depth-K (otherwise min, max, and med operations could potentially find the output without executing all of its arguments). By dissociating the splits by depth, we can clearly identify models that fail to generalize compositionally. Apart from the depth specifications, all train/valid/test sets share the same settings as follows: the maximum sequence length is 50 (tokens), the probability of recursively sampling another function inside a list is 30% at each position, and the maximum number of arguments for a function is 5. The train set consists of 1M, the validation and test sets of 1K sequences.

Results. Tab. 5.4 shows the results. Like on the other tasks, the baseline LSTM and transformers do not generalize well on the test set consisting of deeper problems, while they achieve near-perfect accuracy on IID data. On the contrary, our model achieves a near-perfect generalization.

	IID (1..5)	Test (7..8)
LSTM	**0.99** ± **0.00**	0.71 ± 0.03
Bidirectional LSTM	**1.00** ± **0.00**	0.57 ± 0.04
Transformer	0.98 ± 0.00	0.74 ± 0.03
+ rel	**0.98** ± **0.01**	0.79 ± 0.04
+ abs/rel + gate	**1.00** ± **0.01**	0.90 ± 0.06
+ geom. att. + gate (NDR)	**1.00** ± **0.00**	**0.99** ± **0.01**

Table 5.4: Performance of different models on balanced **ListOps** dataset. All models are trained for 200 K iterations, except all `+gate` variants which converge after 100 K steps. The numbers in the parentheses indicate the problem depths (1-5 for the IID, and 7-8 for the test set).

5.3 Analysis

In this section, we provide some visualizations of attention and gating patterns of the NDR and the corresponding analyses. For more visualizations, we refer the readers to Appendix D.4.

Compositional Table Lookup. Fig. 5.3 shows the gating and attention patterns of the NDR model for an example of the backward presentation task. As shown in Fig. 5.3/Bottom, the gates of different columns open sequentially one after another when the input is available for them. Fig. 5.3/Top shows the corresponding attention maps. Each column attends to the neighboring one, waiting for its computation to be finished. The behavior of the last column is different: It always attends to the second position of the sequence, which corresponds to the last operation to be performed.

ListOps. We can also identify how the NDR processes the data in ListOps. Different attention heads play different roles. We highlight the core observations in Fig. 5.4. The input for this example is: `[SM [MED [MIN 1 7 4 [MAX 2 4 0 8 9 ] ] 7 ] 5 [MED 8 5 8 ] 0 7 ]`. First of all, we find that there is a head (*head 13* in Fig. 5.4, *first row*) which seems to be responsible for connecting operators and their arguments: the operands/arguments of an operation attend

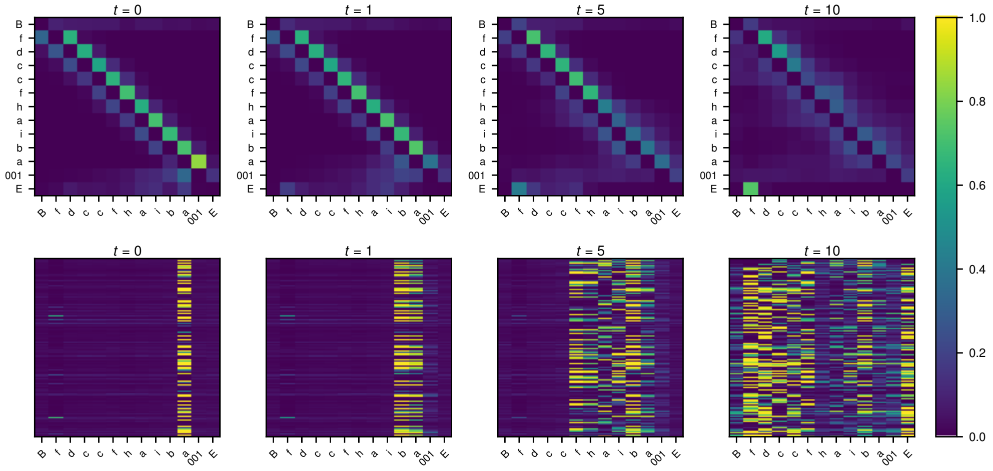


Figure 5.3: Example visualization of NDR. For other models, see Appendix D.4. Top: Attention map for different steps. The x/y-axis corresponds to source/target positions, respectively. Each position focuses on the column to the right, except the last one where the result is read from, which focuses on the last operation. The focus becomes clear only once the result is available. Bottom: gate activations for different steps/layers. The gates remain closed until the data dependencies are satisfied.

to the operator. In step 0 ($t = 0$ in the figure), we can recognize that the operations at the deepest level, namely `MAX` and the second `MED` have all the arguments ready (as is shown by vertical lines on the columns corresponding to `MAX` and `MED`). The model indeed identifies that these two operations are ready to be executed and that they can be processed in parallel (these arguments-to-operation attention patterns remain for a few steps). We note that at this stage, the last argument of `MIN` is not yet ready (`[MIN 1 7 4 [MAX 2 4 0 8 9 ] ]`). We can see that only arguments which are already ready (`1 7 4`) attend to the operator (see the column of `MIN`). In step 1 ($t = 1$, *2nd row*), we can see that *head 5* copies the expected result of `MAX`, `9` to the column of the operator (we note that this only requires one step as `9` is always the result of `MAX` when it is one of the arguments of `MAX`). Similarly in step 2, *head 7* (*2nd row*) seems to copy the result of the second `MED`, `8` to the operator column. In step 3 ($t = 3$, *1st row*), we recognize that the result of `MAX` is marked as an argument for `MIN` in *head 13* which is responsible for communication between operators and their arguments. This is shown by the new attention that appears at $t = 3$ in *head 13* from the source position `MAX` to the target position `MIN` (a pattern that is not

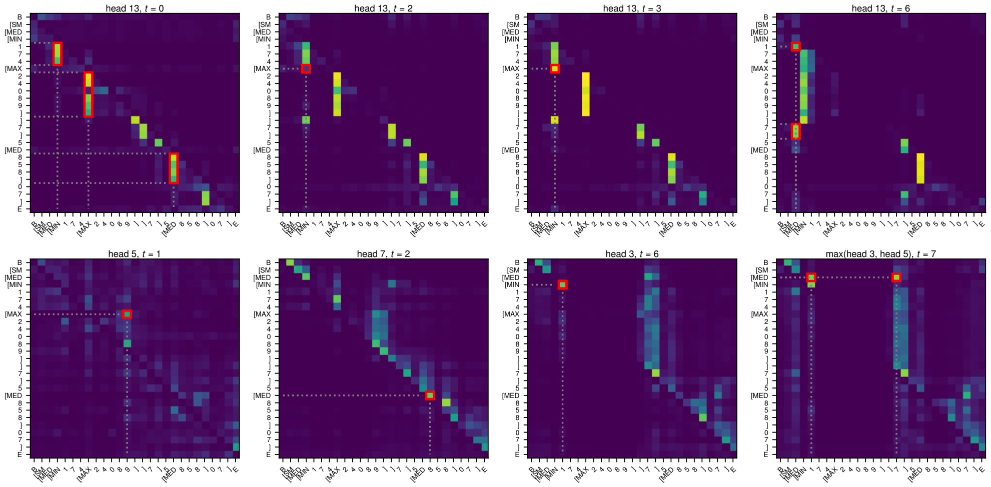


Figure 5.4: Example visualization of NDR on ListOps. The top row shows *head 13* in different steps, which controls which arguments are used in which step. The bottom row shows different heads in different key steps. Please refer to Sec. 5.3 for the step-by-step description. More visualizations are provided in the appendix: Fig. D.8 shows the max of attention over all heads for all steps, Fig. D.9 shows all steps of *head 13*, and Fig. D.10 shows the corresponding gates.

visible at $t = 2$). In *head 3*, $t = 6$ (*2nd row*), the expected result of `MIN`, which is `1`, is copied into the operator, similarly to the patterns observed above for `MAX` and `MED`. In *head 13*, $t = 6$ (*1st row*), all arguments for the first `MED` are now also recognized (the result of `MIN` which is `1`, and `7`). Finally in $t = 7$ (*2nd row*), two heads, *head 3* and *head 5* seem to copy/gather two inputs needed to compute the corresponding median, `1` and `7`, and store them in the column of the operator `MED`. A complete visualization of further steps can be found in the Appendix D.4.2. We noticed that some of the heads do not seem to play a key role; we focused on interpreting those that seem to participate in the main computation. For ListOps, we also partially find the attention patterns described above in the baseline transformer with relative positional encoding, at least on some inspected examples, which also explains its rather high accuracy.

5.4 Discussion

Learning adaptive serialization. The NDR architecture can be understood as performing adaptive serialization of the problem. A key requirement for reusable computation is to decompose the problem into reusable building blocks, typically applied in sequential steps. The granularity of the decomposition determines the degree of reusability: fusing operations in a single step makes the processing faster (fewer steps), but also more specialized. Learning the most granular solutions is thus preferable for generalization. At the same time, not all processing should happen serially: branches of the computational graph that do not have common data dependencies can be processed independently in parallel, which we empirically observe in our NDR in the ListOps example (Sec. 5.3). This enables the architecture to get away with a number of computational steps that reflect the depth of the computational graph rather than the length of the input.

Bottom up approach for improving model architectures. Transformers have seen tremendous success in various application domains [Devlin et al., 2019; Brown et al., 2020; Dosovitskiy et al., 2021]. Impressive results have been reported when they are scaled up with a large amount of data [Brown et al., 2020; OpenAI, 2022; Enryu, 2023]. On the other hand, simple tasks like those highlighted in the present work demonstrate that the transformer architecture still struggles with basic reasoning. Particularly in algorithmic tasks, it is often the case that a suboptimal choice of architecture/optimization method makes the model fall back to simple memorization. We argue that it is crucial to look at isolated problems that test specific generalization capability. This calls for a bottom-up approach: building on toy tasks that focus on individual aspects of generalization and using them for improving models.

5.5 Related Work

Gating inside transformers. Several prior works have proposed using some sort of gating within transformer architectures [Parisotto et al., 2020; Chaabouni et al., 2021]. Our proposed *copy gate* is different from those as it satisfies two important properties. First, our copy gate allows the model to skip the *entire* transformer layer (i.e., both the self-attention and the feedforward blocks) when the gate is closed. Second, the gate function is conditioned on the attention output such that the decision to open or close depends on information from all columns.

Although multiple gating variants have been proposed by Parisotto et al. [2020] to stabilize transformers for reinforcement learning, none of them can produce this behavior. Empirically, we also tried out a few other gating variants which do not satisfy the two properties above; we found them not to improve over regular transformers in our preliminary experiments on compositional table lookup. Recent work by Chaabouni et al. [2021] also makes use of "gating" in transformers through a gated linear unit (GLU) activation function commonly used in convolutional NNs [Dauphin et al., 2017]. Transformer models with such an activation function have been reported to outperform RNN baselines on a systematic generalization task [Dessì and Baroni, 2019]. Unlike our copy gate or Parisotto et al. [2020]'s gating, such a gating activation does not have the "residual" term (i.e. a closed gate zeros out the input), which allows the model to skip a transformation. In a more general context, the benefits of GLU activation in transformers vary between tasks [Irie et al., 2019; Shazeer, 2020]. In language modeling, no improvement is typically obtained by using the standard highway gate instead of the residual connection in transformers [Irie, 2020], while it yields improvements when combined with convolutional layers [Kim and Rush, 2016].

Parameterized geometric distributions. Two recent works [Brooks et al., 2021; Banino et al., 2021] have used a form of parametrized geometric distribution (PGD; in the form of Eq. 5.7). Brooks et al. [2021] have used such a distribution to parameterize the movement of a pointer on a sequence of instructions. Banino et al. [2021] have used it to implement adaptive computation time [Schmidhuber, 2012; Graves, 2016]. We use the PGD to obtain a generic attention mechanism as a replacement for the standard self-attention used in transformers [Vaswani et al., 2017].

Compositional table lookup. CTL task was proposed for evaluating the compositional ability of NNs [Liska et al., 2018]. Previous work evaluated RNNs, RNNs with attention, and transformers on this task with limited success [Hupkes et al., 2019; Dubois et al., 2020]. Dubois et al. [2020] have proposed a special attention mechanism to augment the recurrent architecture. Although they obtained good performance for the forward presentation order, the proposed model failed in the backward one. In contrast, two of our approaches (Sec. 5.2.1) achieve 100% generalization accuracy for both orders.

Positional encodings. Many previous works have focused on improving positional encoding [Schmidhuber, 1992d; Vaswani et al., 2017] for self-attention.

Most notably, relative positional encoding [Schmidhuber, 1992d; Shaw et al., 2018b; Dai et al., 2019] was found to be useful for improving the systematic generalization of transformers (see Chapter 4). Here, we also present two new approaches related to positional encoding. One is the gated combination of absolute and relative positional encoding (Sec. 5.2.1; details in Appendix D.2). We show that absolute positional encoding can complement relative positional encoding. The former enables the model to always attend to a specific position, as is needed for the CTL task in the last step, while the gating allows it to use relative positional encoding for other positions/steps. Second, we introduce directional encoding to augment geometric attention. Unlike positional encoding, which can overfit to a range of positions seen during training, direction information is found to be robust and to be a crucial augmentation of the geometric attention.

5.6 Conclusion

We proposed a new view on the internal operations of transformer encoders as a dynamic dataflow architecture between transformer columns. This overcomes two shortcomings of traditional transformers: the problem of routing and retaining data in an unaltered fashion, which we solve by an additional copy gate, and the problem of learning length-independent attention patterns, which we solve by geometric attention. Our new model, the Neural Data Router (NDR), generalizes to compositions longer than those seen during training on the popular compositional lookup table task in both forward and backward directions. NDR also achieves near-perfect performance on simple arithmetic and ListOps tasks in settings that test systematic generalization in terms of computational depth. In general, the gates and the attention maps collectively make the architecture more interpretable than the baselines.

Chapter 6

Inspecting Systematicity of Neural Networks[1]

In Sec. 5 we improved the length generalization of transformers, which corresponds to better productivity. However, another important aspect of generalization is systematicity (see Sec. 1.1.1). It turns out that systematicity is significantly more challenging compared to productivity.

The focus of this chapter is on *systematicity*: the capability to generalize to unseen compositions of known functions/words. That is crucial for learning to process natural language or to reason on algorithmic problems without an excessive amount of training examples. Some of the existing benchmarks (such as COGS [Kim and Linzen, 2020] and PCFG [Hupkes et al., 2020]) are almost solvable by plain NNs with careful tuning [Csordás et al., 2021], while others, such as CFQ [Keysers et al., 2020], are much harder. A recent analysis of CFQ by Bogin et al. [2022] suggests that the difficult examples have a common characteristic: they contain some local structures (describable by parse trees) which are not present in the training examples. These findings provide hints for constructing both challenging and intuitive (simple to define and analyze) diagnostic tasks for testing systematicity. We propose CTL++, a new diagnostic dataset building upon CTL. CTL++ is basically as simple as the original CTL in terms of task definition, but adds the core challenge of compositional generalization absent in CTL. Such simplicity allows for insightful analyses: one low-level reason for the failure to generalize compositionally appears to be the failure to learn functions whose outputs are symbol representations compatible with inputs of other learned neural functions. We will visualize this.

[1] For the full paper please see our work "The Neural Data Router: Adaptive Control Flow in Transformers Improves Systematic Generalization" [Csordás et al., 2022b]

Well-designed diagnostic datasets have historically contributed to studies of systematic generalization in NNs. Our CTL++ strives to continue this tradition.

6.1 Original CTL

Our new task (Sec. 6.2) is based on the CTL task [Liska et al., 2018; Hupkes et al., 2019; Dubois et al., 2020] whose examples consist of compositions of bijective unary functions defined over a set of symbols. Each example in the original CTL is defined by one input symbol and a list of functions to be applied sequentially, i.e., the first function is applied to the input symbol and the resulting output becomes the input to the second function, and so forth. The functions are bijective and randomly generated. The original CTL uses eight different symbols. We represent each symbol by a natural number, and each function by a letter. For example, '`d a b 3`' corresponds to the expression $d(a(b(3)))$. The model has to predict the corresponding output symbol (this can be viewed as a sequence classification task). When the train/test distributions are independent and identically distributed (IID), even the basic transformer achieves perfect test accuracy [Csordás et al., 2022a]. The task becomes more interesting when test examples are longer than training examples. In such a *productivity* split, which is the common setting of the original CTL [Dubois et al., 2020; Csordás et al., 2022a], standard transformers fail, while NDR and bi-directional LSTM work perfectly.

6.2 Extensions for Systematicity: CTL++

To introduce a *systematicity* split to the CTL framework, we divide the set of functions into disjoint *groups* and restrict the sampling process such that some patterns of compositions between group elements are never sampled for training, only for testing. Based on this simple principle, we derive three variations of CTL++. They differ from each other in terms of compositional patterns used for testing (excluded from training) as described below. We'll also visualize the difference using *sampling graphs* in which the nodes represent the groups, and the edges specify possible compositional patterns. The colors of the edges reflect when the edges are used: black for both training and testing, blue for training, and red only for testing. Note that variants 'A' and 'R' are easy to define by simple grammar, but the 'S' variant has restrictions on the transition symbols, making it very long to describe. Thus, we choose to use the sampling graph

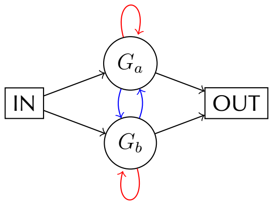


Figure 6.1: Sampling graph for variant 'A.'

representation for all variants.

Variation 'A' (as in 'Alternating'). Here functions are divided in groups G_a and G_b. During training, successively composed functions are sampled from different groups in an alternating way—i.e., successive functions cannot be from the same group. During testing, however, only functions from the same group can be composed. The sampling graph is shown in Fig. 6.1. Importantly, the single function applications are part of the training set, to allow the model to learn common input/output symbol representations for the interface between different groups.

Variation 'R' (as in 'Repeating'). This variant is the complement of variation 'A' above. To get a training example, either G_a or G_b is sampled, and all functions in that example are sampled from that same group for the whole sequence. In test examples, functions are sampled in an alternating way. There is thus no exchange of information between the groups, except for the shared input embeddings and the output classification weight matrix. The sampling graph is like in Fig. 6.1 for 'A' except that blue edges should become red and vice versa (see Fig. E.1 in the appendix).

Variation 'S' (as in 'Staged'). In this variant, functions are divided into five disjoint groups: G_{a1}, G_{a2}, G_{b1}, G_{b2} and G_o. As indicated by the indices, each group belongs to one of the two *paths* ('a' or 'b') and one of the two *stages* ('1' or '2'), except for G_o which only belongs to stage '2' shared between paths 'a' and 'b' during training. The corresponding sampling graph is shown in Fig. 6.2. To get a training example, we sample an integer K which defines the sequence length as $2K + 1$, and iterate the following process for $k \in [0, .., K]$ and $i = 2k$: we first sample a path $p \in \{a, b\}$ and then a function f_i from G_{p1} and a function f_{i+1} from $G_{p2} \cup G_o$. Each example always contains an even number of

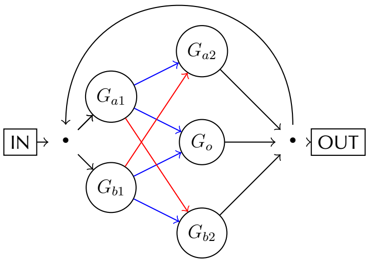


Figure 6.2: Sampling graph for Variant 'S'

functions, and no isolated single function application is part of training, unlike in the previous two variants. For testing, we sample a path $p \in \{a, b\}$ and a function f_i from G_{p1}, but then sample a function f_{i+1} from $G_{\{a,b\} \setminus \{p\}2}$, which results in a compositional pattern never seen during training.

The unique feature of this variant is the use of two stages: as can be seen in Fig. 6.2, during training, given a path $p \in \{a, b\}$, outputs of any functions belonging to G_{p1} are only observed by the functions belonging to G_{p2}, i.e., the stage '2' group belonging to the same path p, or G_o. Hence, if $G_o = \emptyset$, the model has no incentive to learn common representations for the interface between G_{a1} and G_{b1}: to solve the training examples, it suffices to learn output representations of G_{a1} which are 'compatible' with the input representations of G_{a2}; similarly for G_{b1} and G_{b2}. There is no reason for outputs of G_{a1} to be compatible with the inputs of G_{b2} (analogously for G_{b1} and G_{a2}) which is required at test time. The size of G_o is our first parameter for controlling task difficulty (the y-axis of Fig. 6.4 which we will present later).

We introduce further restrictions: for each function $f \in G_o$, we define a set of symbols S_a^f for G_{a1} (and S_b^f for G_{b1}), and we only allow for sampling f if the output symbol of function from G_{a1} (or G_{b1}) belongs to S_a^f (or S_b^f). This allows for defining another control parameter: the number of overlapping symbols between S_a^f and S_b^f (same for all f; the x-axis of Fig. 6.4). Note that we ensure that the union of shared symbols defined for functions in G_o cover all possible symbols. This might not be the case in a more realistic scenario, but as we'll see, the standard models already struggle in this setting. By controlling these two parameters, we precisely control the degree of overlap offered by G_o in terms of both the number of functions and symbols. Ideal models should be "sample efficient" in terms of this overlap, since we cannot expect the training set to contain all combinations of such overlaps in a practical scenario with semantically rich domains such as natural language.

6.3 Results

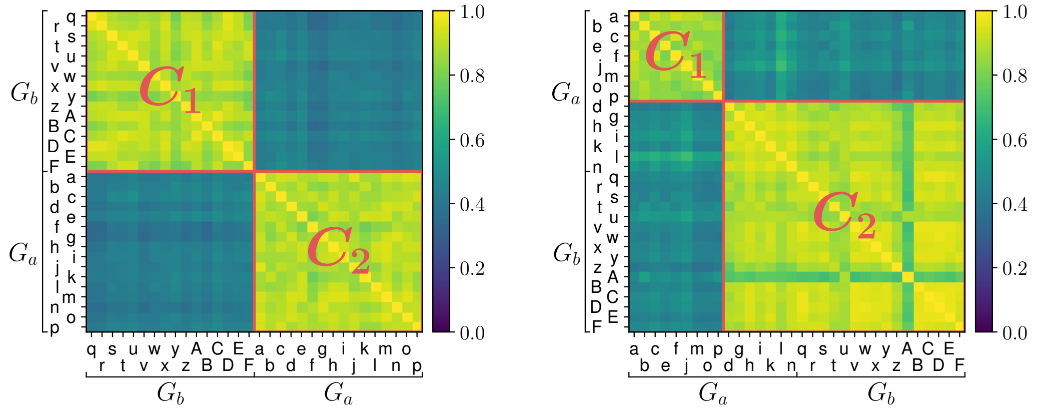
We evaluate standard CTL-tested models on the new CTL+ + task, including: the transformer [Vaswani et al., 2017] with shared layers [Dehghani et al., 2019], the neural data router (NDR) [Csordás et al., 2022a], and the bi-directional LSTM [Graves et al., 2005]. Recall that both NDR and bi-directional LSTM are reported to perfectly solve the original CTL's length generalization split [Csordás et al., 2022a], unlike the transformer. Further experimental details can be found in Appendix E.1.

Model	Dataset	Accuracy	
		IID	OOD
Bi-LSTM	A	1.00 ± 0.00	0.95 ± 0.03
	R	1.00 ± 0.00	1.00 ± 0.00
Transformer	A	1.00 ± 0.00	0.21 ± 0.09
	R	1.00 ± 0.00	0.75 ± 0.25
NDR	A	1.00 ± 0.00	0.34 ± 0.26
	R	1.00 ± 0.01	0.75 ± 0.27

Table 6.1: Results on the task variants 'A' and 'R.' Mean and standard deviation are computed using 25 seeds.

6.3.1 Results on Variants 'A' and 'R'

Tab. 6.1 shows the performance overview for 'A' and 'R.' The OOD (out-of-distribution) column indicates the train/test data sampling processes described above. As a reference, we also report the IID cases where the training example sampling graph is also used for testing. Our initial expectation was that the pressure from shared input/output embeddings is sufficient for these models to learn common symbol representations for all functions. However, we observe that only the bi-LSTM solves these tasks consistently across seeds. Interestingly, the NDR, which perfectly performs on the length generalization split of CTL and beyond [Csordás et al., 2022a], performs poorly on both the 'A' and 'R' variants of CTL+ +. Tested with 25 seeds, only 32% of the seeds (out of 25) achieve over 95% accuracy for NDR on 'R' (20% for the standard transformer). The success



(a) Example for symbol '6'. Perfect clustering w.r.t. the function groups is observed.

(b) Example for symbol '3'. Partial clustering w.r.t. the function groups is observed.

Figure 6.3: Cosine similarity of output representations of different functions representing the same symbol for the NDR with a seed that fails on 'R.'

rate is 0% for the 'A' variant[2]. These simple tasks thus turn out to be good first diagnostic tasks for testing systematicity.

Analysis. The small input/output space of this task allows for an exhaustive analysis of the learned symbol representations. Specifically, given an *output* symbol s', for each function $f \in G_a \cup G_b$, we can find a unique input symbol s such that $s' = f(s)$ (because all functions defined for this task are bijective). Hence, for a fixed symbol s', for all functions f, we can extract the learned vector representation of this symbol s' at the output of f as the vector of the layer beneath the final classification layer when we feed 'f s' to the network. Then we can compare the extracted representations (of a fixed symbol for different functions) by computing their cosine similarities.

Here we compare representations learned with successful/failed seeds for NDR in variation 'R.' Fig. 6.3a and 6.3b show the results for two different (output) symbols '6' and '3' from the same *failed* seed. In both cases we observe two clusters (C_1 and C_2): two separate/different representations are learned for

[2] 'A' turns out to be harder than 'R.' We speculate that in both 'A' and 'R', given that the training set contains single function applications with shared input/ouput embeddings, the learned symbol representation of all functions should be compatible with each other to some extent, but with some "deviation" from perfect compatibility in case of failure. Such "deviations" might accumulate in case of 'A' where we sample all functions in each sequence from a single group at test time.

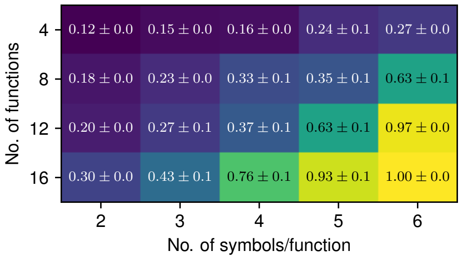


Figure 6.4: Test accuracy of NDR on the 'S' variant. The total number of symbols is 8, and the number of functions is 32. The y-axis shows the number of overlapping functions, while the x-axis shows the number of symbols shared between two groups for each function in G_o during training (Sec. 6.2). Results for the transformer and LSTM are reported in the appendix (Figs. E.3 and E.4).

the same symbol (by abuse of notation, we also refer to the corresponding representations as C_1 and C_2). In the case of symbol '6' in Fig. 6.3a, we observe perfect/strict clustering in line with the group of the applied function; C_1 and C_2 are representations of symbol '6' learned by functions belonging to G_a and G_b respectively. This is problematic since functions in G_a never see symbol '6' represented as C_2 during training (analogously for functions in G_b with representation C_1). As a consequence, during testing, when a function $f_a \in G_a$ is applied after a function $f_b \in G_b$, f_b may output symbol '6' represented as C_2, and pass it to f_a, but in principle, f_a can not "understand/interpret" C_2 as representing symbol '6.' This naturally prevents cross-group generalization. In the case of symbol '3' shown in Fig. 6.3b, some of the functions yield the same symbol representations as certain functions from the other group (see the cluster C_2 at the lower right: a good trend), but we still have a small cluster (C_1 at the upper left) consistent only among elements of G_a. Hence, cross-group generalization can still fail because the functions in G_b never see symbol '3' represented as C_1 during training but only during testing. In contrast, for *successful* seeds, we do not observe any of these clusters for any symbols (see Fig. E.5 in the appendix). A single representation shared across all functions is learned for each symbol. Further quantitative analysis can be found in Appendix E.2.

6.3.2 Results of Staged Variant 'S'

As described in Sec. 6.2, variant 'S' is designed to evaluate models at different task difficulty levels determined by the number of overlapping functions and symbols during training. Fig. 6.4 shows the corresponding performance overview for NDR. The overall picture is similar for bi-LSTM and transformer (see Figs. E.3 and E.4 in the appendix). We observe that to achieve 100% accuracy, half of the possible functions should overlap (16/32), as well as most of the possible symbols seen for each function (6/8). This implies an unrealistically large amount of data for real world scenarios, where the "functions" might correspond to more complex operations with multiple input arguments (as in the CFQ case). This calls for developing approaches that achieve higher accuracy in the upper left part of Fig. 6.4.

6.4 Limitations

Achieving 100% on this dataset may be a necessary condition for NNs capable of systematic generalization, but certainly not a sufficient one. In practice, there may be many reasons which prevent NNs from generalizing systematically in other tasks or, more generally, on real-world data. Compare the original CTL dataset for evaluating productivity: Csordás et al. [2022a] show that some models that achieve 100% on CTL still fail in other tasks such as ListOps. This is why we refer to CTL++ as a simple *diagnostic* dataset for testing systematicity of NNs. Nevertheless, it allows for uncovering certain important failure modes of NNs.

6.5 Conclusion

Motivated by the historically crucial role of diagnostic datasets for research on systematic generalization of NNs, we propose a new dataset called CTL++. Unlike the classic CTL dataset, typically used for testing productivity, CTL++ is designed for testing systematicity. We propose three variants, 'A,' 'R,' and 'S.' Despite their simplicity, even the CTL-solving transformer variant fails on 'A' and 'R.' Using 'S,' we show that existing approaches require impractically large amounts of examples to achieve perfect compositional generalization. The small task size allows for conducting exhaustive visualizations of (in)compatibility of learned symbol representations in outputs of functions with inputs of subsequent functions. Of course, the ultimate goal is to go beyond just solving CTL++. Nevertheless, we hope CTL++ will become one of the standard diagnostic datasets for testing systematicity of NNs.

Chapter 7

Accelerating Transformer MLP Layers: a Path Towards Scalable NDRs[1]

We saw in Sec 5 that NDRs show good length generalization properties. But in practice, if they are used for language modeling applications, they would be very slow. The reason is the shared layers: they lose a significant number of parameters, proportional to the number of layers in the network. If a single layer is scaled up to compensate for this loss, it becomes prohibitively slow due to the $O(n^3)$ complexity of matrix multiplication. Furthermore, memory usage would also increase. Thus, as a first step towards scaling up NDRs, we focus on using Mixture of Experts (MoEs) for accelerating standard transformers. This avoids the extra difficulty that comes from sharing the layers and focuses only on the efficiency of the MoEs.

Another motivation for improving MoEs comes from the recent impressive results achieved by large language models (LLMs; [Radford et al., 2019; Brown et al., 2020; Rae et al., 2021]). The vast resource requirement remains their obvious limitation. In fact, most existing LLMs, such as GPT-3 [Brown et al., 2020], cannot be trained, fine-tuned or even evaluated without access to enormous compute. Many recent works strive to develop LLMs that, at least, enable inference with limited resources (e.g., on consumer hardware), e.g., by building "smaller" yet capable LMs [Touvron et al., 2023; Taori et al., 2023; Chiang et al., 2023] or developing post-training quantization methods [Zafrir et al., 2019; Dettmers et al., 2022]. Although these methods are gaining popularity, a

[1] For the full paper please see our work "Approximating Two-Layer Feedforward Networks for Efficient Transformers" [Csordás et al., 2023]

principled solution for resource-efficient neural networks (NNs) remains elusive.

One promising approach explored by several recent works on extremely large LMs is the sparse mixture of experts (MoE; [Shazeer et al., 2017; Lewis et al., 2021; Lepikhin et al., 2021; Fedus et al., 2022; Clark et al., 2022; Chi et al., 2022]). Unlike their *dense* counterparts, MoEs only compute a subset of their activations (i.e, only a few *experts*) at each step, offering reduced computation and memory costs. However, MoEs are not yet generally adopted as a generic/to-go approach, perhaps because of certain common beliefs on MoEs: (1) They are hard to train (involving complex engineering tricks to prevent collapsing), (2) they are not competitive against their dense counterparts with the *same number of parameters* (in fact, prior work focuses on FLOP-equal comparison, "unfairly" comparing MoEs against dense baselines with many fewer trainable parameters), and finally, (3) they are reserved for extremely-large models (they are rarely/never considered to further improve the efficiency of "small" models). Indeed, even prior work on MoE-based Transformer LMs only deploys MoEs in a few feedforward blocks; while ideally, *all* such blocks should benefit from replacement by MoEs. Here, we challenge these common beliefs and propose novel perspectives on MoEs.

We present MoEs within a unified framework of methods that approximate two-layer feedforward networks, which includes product-key memories (PKMs [Lample et al., 2019]) and top-k sparsification. This principled view not only allows us to conceptually group and compare MoEs with PKMs, it also provides insights on design choices for improving these methods. Our resulting MoE Transformer variant outperforms our improved PKMs, and performs as well as or even outperforms the dense baseline, while using a fraction of its compute for both training and inference. Importantly, unlike prior work, we compare our MoEs with dense baselines with the same number of total trainable parameters, which is crucial for proper evaluation in language modeling. We conduct experiments on the standard WikiText-103 (at two different model scales) and Enwik8 datasets. We demonstrate that MoEs are not limited to extremely-large LMs, but useful as a generic approach for resource-efficient NNs at any scale, and in line with the recent trend of improving "smaller" models [Touvron et al., 2023; Taori et al., 2023; Chiang et al., 2023]. Finally, we release a CUDA kernel for our MoE layers that allows faster wall clock time and large memory reduction compared to the dense model.[2] The kernel, along with the source code for all experiments can be found on `https://github.com/robertcsordas/moe`.

[2] Since we are not CUDA experts, our implementation still has much room for further optimization.

7.1 Background

Transformers [Vaswani et al., 2017] have two main building blocks: the self-attention layer [Schmidhuber, 1991, 1992c, 1993; Bahdanau et al., 2015; Parikh et al., 2016; Cheng et al., 2016], and the two-layer feedforward, i.e, multi-layer perceptron (MLP) block. Acceleration and memory reduction of the self-attention are rather well explored (see e.g., linear attention [Katharopoulos et al., 2020; Choromanski et al., 2021; Schmidhuber, 1991; Schlag et al., 2021]), and very efficient implementations [Dao et al., 2022] are also available. In contrast, resource-efficient MLP blocks are still underexplored. This is our main focus, and it is of particular relevance today, as the proportion of the total parameter counts, compute, and memory requirements due to MLP blocks in Transformers is increasing in ever-growing LLMs.

Let $d_{\text{model}}, d_{\text{ff}}$ denote positive integers. Each Transformer MLP block consists of one up-projection layer with a weight matrix $\boldsymbol{W}_1 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ where typically $d_{\text{ff}} = 4d_{\text{model}}$, and one down-projection layer with parameters $\boldsymbol{W}_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ that projects it back to the original size. Non-linearity (typically ReLU [Fukushima, 1969]) is applied between these two layers. That is, an input $\boldsymbol{x} \in \mathbb{R}^{d_{\text{model}}}$ is transformed to an output $\boldsymbol{y} \in \mathbb{R}^{d_{\text{model}}}$ as

$$\boldsymbol{u} = \mathrm{ReLU}\left(\boldsymbol{W}_1 \boldsymbol{x}\right) \tag{7.1}$$

$$\boldsymbol{y} = \boldsymbol{W}_2 \boldsymbol{u} \tag{7.2}$$

where $\boldsymbol{u} \in \mathbb{R}^{d_{\text{ff}}}$, and we omit biases (as well as batch and time dimensions) for simplicity.

Alternatively, this layer can be viewed as a key-value memory accessed by attention (Vaswani et al. [2017][3],Geva et al. [2021]), where keys and values are rows and columns of weight matrices $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$:

$$\boldsymbol{W}_1 = \begin{bmatrix} — & \boldsymbol{k}_1^\intercal & — \\ — & \boldsymbol{k}_2^\intercal & — \\ & \vdots & \\ — & \boldsymbol{k}_{d_{\text{ff}}}^\intercal & — \end{bmatrix} \tag{7.3}$$

$$\boldsymbol{W}_2 = \begin{bmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_{d_{\text{ff}}} \\ | & | & & | \end{bmatrix} \tag{7.4}$$

[3] See the appendix "Two feedforward Layers = Attention over Parameter" in their paper version "arXiv:1706.03762v3."

where $\boldsymbol{k}_i \in \mathbb{R}^{d_{\text{model}}}, \boldsymbol{v}_i \in \mathbb{R}^{d_{\text{model}}}$ for $i \in \{1, ..., d_{\text{ff}}\}$. Then, the output is computed as "attention":

$$\boldsymbol{y} = \sum_{i=1}^{d_{\text{ff}}} \boldsymbol{v}_i \operatorname{ReLU}(\boldsymbol{k}_i^\intercal \boldsymbol{x}) = \sum_{i=1}^{d_{\text{ff}}} \alpha_i \boldsymbol{v}_i \tag{7.5}$$

where $\alpha_i = \operatorname{ReLU}(\boldsymbol{k}_i^\intercal \boldsymbol{x}) \in \mathbb{R}_{\geq 0}$ are the "attention weights." Note that $\alpha_i = \boldsymbol{u}[i]$ where $\boldsymbol{u}[i] \in \mathbb{R}$ denotes the i-th component of $\boldsymbol{u} \in \mathbb{R}^{d_{\text{ff}}}$ in Eq. 7.1. Unlike standard self-attention, the MLP block uses a ReLU activation function (instead of softmax) without scaling.

It has been observed that, in practice, only a few of the factors $\boldsymbol{k}_i^\intercal \boldsymbol{x}$ are positive [Li et al., 2023; Shen et al., 2023], making the first layer's output, i.e., $\boldsymbol{u}$, sparse. Concretely, Shen et al. [2023] report that in a Transformer with $d_{\text{model}} = 256$ and $d_{\text{ff}} = 1024$, 10% of the channels account for 90% of the total activation mass. We confirm this trend in our own preliminary study. Fig. 7.1 shows the average number of non-zero units in $\boldsymbol{u}$ of size $d_{\text{ff}} = 2053$ in our 47M parameter dense model trained on WikiText-103 (we refer to App. F.1.2 for more details). The number is below 200 for all layers. This suggests that the MLP block can be *approximated* without a significant performance loss.

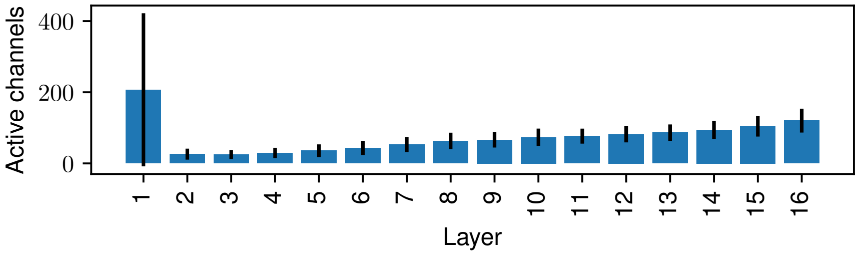


Figure 7.1: Number of active channels in $\boldsymbol{u}$ in our dense 47M parameter model on WikiText-103. Standard deviation over all tokens of the test and validation set.

7.2 Approximating 2-layer MLPs

Here we present a unified view on methods to approximate 2-layer MLPs (Sec. 7.1) that includes many existing methods such as MoEs (Sec. 7.2.3) and PKMs (Sec. 7.2.2).

Preliminaries. Let $\hat{\boldsymbol{y}} \in \mathbb{R}^{d_{\text{model}}}$ denote an approximation of $\boldsymbol{y} \in \mathbb{R}^{d_{\text{model}}}$ in Eq. 7.5. Let $\boldsymbol{y}_i \in \mathbb{R}^{d_{\text{model}}}$ denote $\boldsymbol{y}_i = \alpha_i \boldsymbol{v}_i$ for $i \in \{1, ..., d_{\text{ff}}\}$. The core idea is to approximate the sum in Eq. 7.5, i.e., $\boldsymbol{y} = \sum_{i=1}^{d_{\text{ff}}} \boldsymbol{y}_i$ by only keeping a subset $\mathcal{S} \subset \{1, ..., d_{\text{ff}}\}$ of the key-value pairs, i.e., $\hat{\boldsymbol{y}} = \sum_{i \in \mathcal{S}} \boldsymbol{y}_i$. The intuition of this approximation is as follows. We assume that a good approximation $\hat{\boldsymbol{y}}$ of $\boldsymbol{y}$ is the one that minimizes their Euclidean distance $e = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||_2^2 \in \mathbb{R}$, which can now be expressed as $e = ||\sum_{i \in \bar{\mathcal{S}}} \alpha_i \boldsymbol{v}_i||_2^2$ where $\bar{\mathcal{S}}$ denotes the complement of $\mathcal{S}$, i.e., $\bar{\mathcal{S}} = \{1, ..., d_{\text{ff}}\} \setminus \mathcal{S}$. Since we have $e = ||\sum_{i \in \bar{\mathcal{S}}} \alpha_i \boldsymbol{v}_i||_2^2 \leq \sum_{i \in \bar{\mathcal{S}}} \alpha_i ||\boldsymbol{v}_i||_2^2$ (triangle inequality; where the equality is achieved when $\boldsymbol{v}_i$ are orthogonal), this upper-bound $\sum_{i \in \bar{\mathcal{S}}} \alpha_i ||\boldsymbol{v}_i||_2^2$ can be minimized if each term $c_i = \alpha_i ||\boldsymbol{v}_i||_2^2 \in \mathbb{R}$ are small. If we further assume that all value vectors $\boldsymbol{v}_i$ have the same norm, the crucial factor for approximation quality is reduced to the attention weights α_i. In this context, we also call α_i the *contribution* of key-value pair i.

Let K be a positive integer. The general idea of all the methods discussed in this work is to keep K pairs ($\boldsymbol{k}_i$, $\boldsymbol{v}_i$) whose contribution α_i is the highest and ignore other pairs with low contribution. The goal is to find the best mechanism to select such K pairs. Here, we discuss three variants: Top-K activation (Sec. 7.2.1), Product-Key Memories (PKMs, Sec. 7.2.2), and Mixture of Experts (MoEs, Sec. 7.2.3).

7.2.1 Top-K Activation Function

The most straightforward implementation of the approximation described above is the top-K activation function:

$$\mathcal{E}_{\boldsymbol{x}} = \arg\operatorname{topk}(\boldsymbol{u}, K) \subset \{1, ..., d_{\text{ff}}\} \tag{7.6}$$

$$\hat{\boldsymbol{y}} = \sum_{i \in \mathcal{E}_{\boldsymbol{x}}} \alpha_i \boldsymbol{v}_i \tag{7.7}$$

Unfortunately, this only saves less than half of the entire computation: while this allows us to reduce computation of Eq. 7.2, no computation can be saved in Eq. 7.1 because full computation of $\boldsymbol{u} = \operatorname{ReLU}(\boldsymbol{W}_1 \boldsymbol{x})$ is required for Eq. 7.6. Going beyond this requires us also to introduce some approximation to Eq. 7.6 as in PKMs (Sec. 7.2.2) and MoEs (Sec. 7.2.3).

7.2.2 Product-Key Memories (PKMs)

Product-Key memories [Lample et al., 2019] consist of replacing $\boldsymbol{W}_1 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ in Eq. 7.1 by two matrices $\boldsymbol{W}_a, \boldsymbol{W}_b \in \mathbb{R}^{\sqrt{d_{\text{ff}}} \times \frac{d_{\text{model}}}{2}}$. It slices the input vector

$\boldsymbol{x} \in \mathbb{R}^{d_{\text{model}}}$ into two halves, $\boldsymbol{x}_a, \boldsymbol{x}_b \in \mathbb{R}^{\frac{d_{\text{model}}}{2}}$, so that $\boldsymbol{x} = \boldsymbol{x}_a | \boldsymbol{x}_b$, where | denotes concatenation. The matrix multiplication is then performed on these smaller vectors: $\boldsymbol{u}_a = \boldsymbol{W}_a \boldsymbol{x}_a$ and $\boldsymbol{u}_b = \boldsymbol{W}_b \boldsymbol{x}_b$. Then $\boldsymbol{u} \in \mathbb{R}^{d_{\text{ff}}}$ is calculated by combining the elements of $\boldsymbol{u}_a \in \mathbb{R}^{\sqrt{d_{\text{ff}}}}$ and $\boldsymbol{u}_b \in \mathbb{R}^{\sqrt{d_{\text{ff}}}}$ in all possible ways (i.e., Cartesian products), similarly to the outer product, but using addition instead of multiplication, i.e., for all $i \in \{1, ..., d_{\text{ff}}\}$,

$$\boldsymbol{u}[i] = \boldsymbol{u}_b[\lfloor i / \sqrt{d_{\text{ff}}} \rfloor] + \boldsymbol{u}_a[i \text{ mod } \sqrt{d_{\text{ff}}}] \tag{7.8}$$

In addition to applying Top-K at the output as in Sec 7.2.1, here Top-K can also be used to accelerate the operation from above. By applying Top-K to $\boldsymbol{u}_a$ and $\boldsymbol{u}_b$ before combining them to compute $\boldsymbol{u}$, only the $K^2 << d_{\text{ff}}$ components of $\boldsymbol{u}[i]$ have to be calculated, and they are guaranteed to contain the K biggest components of the full $\boldsymbol{u}$.

In the original formulation [Lample et al., 2019], PKMs use a softmax activation function, taking inspiration from self-attention [Vaswani et al., 2017]. Instead, we will show how a non-competing activation function, such as ReLU is a better choice (see Sec. 7.5.2).

7.2.3 Mixture of Experts (MoE)

Let N_E, G denote positive integers. MoEs partition d_{ff} pairs of ($\boldsymbol{k}_i$, $\boldsymbol{v}_i$) (see their definition in Sec. 7.1) into N_E groups of size G each, such that $G \cdot N_E = d_{\text{ff}}$. This means that the weight matrices $\boldsymbol{W}_1 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ (Eqs. 7.1-7.2) are partitioned into matrices $\boldsymbol{W}_1^e \in \mathbb{R}^{\frac{d_{\text{ff}}}{N_E} \times d_{\text{model}}}$ and $\boldsymbol{W}_2^e \in \mathbb{R}^{d_{\text{model}} \times \frac{d_{\text{ff}}}{N_E}}$ for $e \in \{1, ..., N_E\}$,

$$\boldsymbol{W}_1^e = \begin{bmatrix} — & \boldsymbol{k}_{eG+1}^{\intercal} & — \\ — & \boldsymbol{k}_{eG+2}^{\intercal} & — \\ & \vdots & \\ — & \boldsymbol{k}_{(e+1)G}^{\intercal} & — \end{bmatrix} \tag{7.9}$$

$$\boldsymbol{W}_2^e = \begin{bmatrix} | & | & & | \\ \boldsymbol{v}_{eG+1} & \boldsymbol{v}_{eG+2} & \dots & \boldsymbol{v}_{(e+1)G} \\ | & | & & | \end{bmatrix} \tag{7.10}$$

The output is computed as:

$$\hat{\boldsymbol{y}} = \sum_{e \in \mathcal{E}_{\boldsymbol{x}}} \boldsymbol{W}_2^e \boldsymbol{s}[e] \, \text{ReLU}(\boldsymbol{W}_1^e \boldsymbol{x}) \tag{7.11}$$

where $\boldsymbol{s}[e] \in \mathbb{R}$ is the e-th element of vector $\boldsymbol{s} \in \mathbb{R}^{N_E}$ computed by an expert scoring function $\mathrm{sel} : \mathbb{R}^{d_{\text{model}}} \to \mathbb{R}^{N_E}$ (typically $\boldsymbol{s} = \mathrm{sel}(\boldsymbol{x}) = \mathrm{softmax}(\boldsymbol{W}_3\boldsymbol{x})$ with $\boldsymbol{W}_3 \in \mathbb{R}^{N_E \times d_{\text{model}}}$), and $\mathcal{E}_{\boldsymbol{x}}$ denotes a subset of indices $\{1, ..., N_E\}$ resulting from the Top-K operation on $\boldsymbol{s}$, i.e., $\mathcal{E}_{\boldsymbol{x}} = \arg\mathrm{topk}(\boldsymbol{s}, K)$. Note that in some variants, additional *re-normalization* is applied *after* Top-K, so that $\sum_{e \in \mathcal{E}_{\boldsymbol{x}}} \boldsymbol{s}[e] = 1, \boldsymbol{s}[e] \geq 0$; we define such an operation as $\mathrm{norm\,topk}$, see its exact definition in App. F.1.1 [4]. The efficiency of MoEs comes from the fact that $N_E \ll d_{\text{ff}}$, therefore calculating $\boldsymbol{s}$ is cheap. Furthermore, G and K are chosen so that $G * K \ll d_{\text{ff}}$, so the calculation performed by experts is less expensive than dense MLP.

Given the notation above, it is straightforward to see that MoEs can also be viewed as approximating 2-layer MLPs with a trainable component (i.e., the selection function sel to produce $\boldsymbol{s}$). Similarly to Eqs. 7.5 and 7.7, Eq. 7.11 can be expressed as:

$$\hat{\boldsymbol{y}} = \sum_{e \in \mathcal{E}_{\boldsymbol{x}}} \sum_{i=1}^{G} \alpha_{eG+i} \boldsymbol{s}[e] \boldsymbol{v}_{eG+i} \tag{7.12}$$

where, compared to Eqs. 7.5 and 7.7, the "contribution scores" of key-value pair i (defined in Sec. 7.2/Preliminaries) have an additional factor $\boldsymbol{s}[e]$ of an expert group e to which the key-value pair belongs.

The key challenge of MoEs is to learn an expert selection mechanism/function sel that assigns high scores to only a few experts (so that we can ignore others without sacrificing performance), while avoiding a well-known issue, called expert *collapsing*, where only a few experts are used and the rest are never selected. To avoid this, some regularization is typically applied to the selection score $\mathrm{sel}(\boldsymbol{x})$, encouraging a more uniform routing of experts across the whole batch of tokens. We provide a comprehensive review of MoE variants and their details in Sec. 7.3 and our improved version in Sec. 7.4.

7.3 Existing MoE Variants

Several variations of MoEs have been proposed with many different details. Here, we briefly review the most popular and representative ones (e.g., we do not cover those that make use of reinforcement learning for expert routing) before describing our improved version in Sec. 7.4. We'll review their *expert selection function* and *regularization method*, and highlight their key characteristics.

[4] In the case of the $\mathrm{softmax}(\cdot)$ activation function, this is equivalent to applying Top-K to the logits *before* softmax.

Sparsely Gated Mixtures of Experts. Shazeer et al. [2017] have revisited MoEs [Jacobs et al., 1991; Ivakhnenko and Lapa, 1965a] with the Top-K operation, allowing a reduction in its resource demands. Their method is basically the one described in Sec. 7.2.3 (with re-normalization after Top-K) except that they use a noisy gating function:

$$\mathrm{sel}(\boldsymbol{x}) = \mathrm{softmax}(\boldsymbol{W}_3\boldsymbol{x} + \mathcal{N}(0,1) \cdot \zeta(\boldsymbol{W}_4\boldsymbol{x}))$$

where $\boldsymbol{W}_4 \in \mathbb{R}^{N_E \times d_{\text{model}}}$, the Gaussian noise term $\mathcal{N}(0,1)$ is element-wise and independent for each channel, and $\zeta(x) = \log(1+e^x)$. They use the following auxiliary regularization term for load balancing,

$$L = \mathrm{CV}\left(\sum_{\boldsymbol{x}\in\mathcal{B}} \mathrm{norm\,topk}(\mathrm{sel}(\boldsymbol{x}))\right) \tag{7.13}$$

where $\mathrm{CV}(x) = \frac{\mu_x}{\sigma_x}$ is the coefficient of variation and $\mathcal{B}$ is the set of all tokens in the batch.

Key characteristics: The scores are normalized after the top-K operation (with $K > 1$), which is equivalent to applying top-K *before* the softmax.

Switch Transformer. Fedus et al. [2022] integrate the MoE above into the Transformer to obtain their Switch Transformer. In terms of MoE details, one of Fedus et al. [2022]'s key claims is that top-1 routing is enough. Their selection function is simply: $\mathrm{sel}(\boldsymbol{x}) = \mathrm{softmax}(\boldsymbol{W}_3\boldsymbol{x})$, but they propose a hard load-balancing between experts that run on different hardware accelerators: At most $\mu\frac{|\mathcal{B}|}{N_E}$ tokens are allowed to be routed to an expert, where $\mu \in \mathbb{R}_{>0}$ is the capacity factor (typically between 1 and 1.5), defining how many times more tokens can be processed by one expert compared to the ideal case of uniform routing. Each expert is forbidden to process more than this number of tokens. For regularization, the fraction of the tokens $\boldsymbol{f} \in \mathbb{R}^{N_E}$ processed by each expert, and the average selection probability $\boldsymbol{p} \in \mathbb{R}^{N_E}$ for each expert are calculated ($K = 1$; top-1 is used) as:

$$\boldsymbol{f}_i = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{x}\in\mathcal{B}} \mathbb{1}\{i \in \arg\mathrm{topk}(\mathrm{sel}(\boldsymbol{x}), K)\} \tag{7.14}$$

$$\boldsymbol{p} = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{x}\in\mathcal{B}} \mathrm{sel}(\boldsymbol{x}) \tag{7.15}$$

$$L = N_E \boldsymbol{f} \cdot \boldsymbol{p} \tag{7.16}$$

where $\mathbb{1}$ denotes the indicator function (which is equal to 1 if the argument is true, and 0 otherwise), and $\cdot$ denotes dot product. Intuitively, this serves as an adaptive regularization that penalizes experts that are often used with high "weights." Additionally, they use dropout with a high drop rate (40%) in experts (but only 10% in the normal layers).

Furthermore, Fedus et al. [2022] also proposes to initialize experts with $\sqrt{\frac{0.1}{G}}$. As we'll see in Sec. 7.4, we use a modified version of this scheme.

Note that applying Top-K after softmax encourages collapse: if the score of the selected expert is increased, the scores of all other experts are automatically decreased. This is not the case for Shazeer et al. [2017]: In their method, only the selected experts compete with each other, so if their presence is beneficial, their score can be increased.

Key characteristics: Note that Top-1 is applied *after* the softmax without renormalization.

BASE layers and S-BASE. Inspired by the routing strategy and the hard capacity factor of the Switch Transformer, Lewis et al. [2021] propose BASE layers. They use top-1 routing and a sigmoid activation σ in the selection function:

$$\mathrm{sel}(\boldsymbol{x}) = \sigma(\boldsymbol{W}_3\boldsymbol{x}) \tag{7.17}$$

Now instead of using $\arg\mathrm{topk}$, they solve the following linear assignment problem to find the index $e_{\boldsymbol{x}} \in \{1, ..., N_E\}$ of the expert to which each input $\boldsymbol{x} \in \mathcal{B}$ is routed,

$$\begin{aligned} &\underset{e_{\boldsymbol{x}} \in \{1,...,N_E\}, \boldsymbol{x} \in \mathcal{B}}{\mathrm{maximize}} \sum_{\boldsymbol{x} \in \mathcal{B}} \mathrm{sel}(\boldsymbol{x})[e_{\boldsymbol{x}}] \\ &\text{s.t. } \forall i \in \{1, ..., N_E\}, \sum_{\boldsymbol{x} \in \mathcal{B}} \mathbb{1}\{e_{\boldsymbol{x}} == i\} = \frac{|\mathcal{B}|}{N_E} \end{aligned} \tag{7.18}$$

This guarantees uniform assignment of experts, which is efficient for multi-accelerator training. The output is computed using Eq. 7.11 with $\mathcal{E}_{\boldsymbol{x}} = \{e_{\boldsymbol{x}}\}$ (a set with a single element; "top-1"). However, in inference time, such a balance is not possible because not all tokens of the sequence are available at each step; $\mathcal{E}_{\boldsymbol{x}} = \{\arg\max(\mathrm{sel}(\boldsymbol{x}))\}$ is used instead. Lewis et al. [2021] show that, while during training, the routing is enforced to be completely uniform, during the test time, the distribution looks exponential (in fact, this is similar to the Switch Transformer but more balanced for BASE).

The algorithm for solving the linear assignment problem (Eq. 7.18) is difficult to implement efficiently on modern accelerators. Clark et al. [2022] have proposed to use the Sinkhorn algorithm [Sinkhorn, 1964; Sinkhorn and

Knopp, 1967] instead (resulting in a model called Sinkhorn-BASE or S-BASE), to approximate the solution to this problem (note that similar routing is independently discussed by Kool et al. [2021]). They report that this works well, while being simpler to implement. Thus, our reimplementation of BASE is S-BASE using the Sinkhorn algorithm.

Key characteristics: During training, Sinkhorn iterations are used on scores to obtain a balanced assignment. Singmoid activation is always applied to compute the weighting score.

Overall, all above load-balancing methods are rather complex. We propose a simpler but effective approach for MoEs in Sec. 7.4.

7.4 Improving Mixture of Experts

Here we present our improved MoE variant, which we call σ-MoE. We conduct thorough ablation studies on our design choices in Sec. 7.5.

σ-MoE Expert Selection Function. Our MoE makes use of the top-K operation (unlike BASE). The activation we use on the selection function is sigmoid (as in Eq. 7.17 of BASE) instead of softmax used in Switch Transformer and Sparsely Gated Mixtures of Experts. This choice is motivated by the view of MoEs as approximate 2-layer MLPs (Sec. 7.2). In fact, softmax introduces competition between experts. No such competition between channels is used in the regular 2-layer MLP (that is, there is no constraint on α_i in Eq. 7.5). This suggests that, in principle, no competition is needed between terms in the sum of Eq. 7.12 in the MoE either, to induce sparsity. It is also well known to practitioners that softmax as regular activation negatively affects the trainability of standard MLPs. Therefore, we opt for sigmoid instead of softmax; we experimentally confirm that this is indeed a good choice.

Additionally, looking at MoEs in this framework gives us hints on combining them with Top-K activation (Sec. 7.2.1) for further acceleration. We can calculate $\boldsymbol{u}^e = \boldsymbol{s}[e]\,\mathrm{ReLU}(\boldsymbol{W}_1^e x)$ (Eq. 7.11) for the selected experts and perform an additional Top-K to keep the highest units among them and set the rest to zero. We leave this for future work.

σ-MoE Initialization. Another design choice guided by the MLP-approximation view of MoEs (Sec. 7.2) is the initialization scheme for experts. Typically, experts are assumed to be independent, and the standard deviation of the initialization [Glorot and Bengio, 2010; He et al., 2015] of $\boldsymbol{W}_2^e$

is calculated based on G instead of d_{ff}. Our experiments in Sec. 7.5.3 show that this is suboptimal.

In contrast, we initialize all weight matrices identically to the pre-layernorm dense baselines, not taking in account the smaller size of the individual experts, i.e., $\boldsymbol{W}_1^e \sim \mathcal{N}(0, \sqrt{\frac{2}{d_{\text{model}} \cdot n_{\text{layers}}}})$ and $\boldsymbol{W}_2^e \sim \mathcal{N}(0, \sqrt{\frac{2}{d_{\text{ff}} \cdot n_{\text{layers}}}})$ where n_{layers} denotes the number of layers, using d_{model} and d_{ff} instead of G.

We also take special care when initializing $\boldsymbol{W}_3$ of the selection function. We initialize it to a normal distribution with the same standard deviation as $\boldsymbol{W}_1^e$, but we also ensure that the rows of $\boldsymbol{W}_3$ have the same norm. This can be easily achieved in practice by initializing the weights to $\boldsymbol{W}_3' \sim \mathcal{N}(0, 1)$, rescaling its rows to norm 1, and then rescaling the whole matrix again to have the desired standard deviation. Note that each scalar score in $\boldsymbol{s}$ is the dot product of a row of $\boldsymbol{W}_3$ and $\boldsymbol{x}$. This initialization method ensures that only the angle between $\boldsymbol{x}$ and the rows of $\boldsymbol{W}_3$ initially affects the score $\boldsymbol{s}$, rather than an additional random factor resulting from initialization.

σ-MoE Regularization. As already noted in Sec. 7.3, existing regularization methods for load-balancing are complex (e.g., Switch Transformers need to deal separately with the actual selection distribution and the scores, Sparsely Gated Mixture of Experts needs noise in the selection function). On the contrary, we propose to simply maximize the entropy of the selection distribution $\boldsymbol{p} \in \mathbb{R}^{N_E}$ calculated across the entire batch. Let $\mathcal{B}$ be the set of all tokens in the batch (counting through both batch and time dimensions). We introduce the following regularization term L:

$$\boldsymbol{p} = \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x} \in \mathcal{B}} \operatorname{softmax}(\boldsymbol{W}_3 \boldsymbol{x}) \tag{7.19}$$

$$L = \sum_{e=1}^{N_E} \boldsymbol{p}[e] \log \boldsymbol{p}[e] \tag{7.20}$$

Furthermore, we propose to randomly drop complete experts during training; we refer to this as *expert dropout*. Unlike the standard dropout on the activation level, we do not apply rescaling, i.e.,

$$\operatorname{sel}(\boldsymbol{x}) = \begin{cases} \sigma(\boldsymbol{W}_s \boldsymbol{x}) \odot \boldsymbol{m} & \text{if training} \\ \sigma(\boldsymbol{W}_s \boldsymbol{x}) & \text{otherwise} \end{cases} \tag{7.21}$$

where $\boldsymbol{m} \in \{0, 1\}^{N_E}$, $\boldsymbol{m} \sim \text{Bernoulli}(1 - \delta)$, where δ is the dropout rate, and $\odot$ is the element-wise product. This prevents the dropped experts from being selected, while not affecting the other ones. We experimentally show that our regularization method (Eq. 7.20) and expert dropout (Eq. 7.21) are both effective despite their simplicity.

7.5 Experiments

Our experimental setup is based on Dai et al. [2019]'s Transformer XL with some modifications: we use pre-layer norm and reduce the number of training steps to 100k to reduce the computational budget. Furthermore, to match the parameter counts between the baseline and MoEs, we slightly modify the hyperparameters of the baselines [Dai et al., 2019]. In fact, our MoE CUDA kernel can only work with dimensions divisible by 4. We round the original sizes up to the next suitable number, e.g., we change d_{model} of our 47M-parameter WikiText-103 model from the original 410 to 412. Furthermore, since MoEs require additional parameters for the expert selection function, we compensate for these by increasing the d_{ff} of the baseline model to match the number of parameters. Our modified baseline model on Enwik8 still has 41M parameters and performs similarly to the original Transformer XL (see Tab. 7.1). For WikiText-103, we use subword units [Sennrich et al., 2016] using SentencePiece tokenizer [Kudo and Richardson, 2018] instead of the word-level vocabulary, to avoid extra tricks required to reduce the parameter count and compute requirement resulting from the huge vocabulary size. On WikiTest-103, we consider two different model sizes: a 47M-parameter one (denoted by "WT-S" for "small"), and a 262M-parameter one ("WT-B" for "big"). We refer to Enwik8 as "E8" in certain tables. For more details, see Appendix F.2.

For all the methods considered, we use them in *every* MLP block of the model, which is not a common practice in the literature. Typically, MoE (or other approximation methods) is used only once every n^{th} layer or even only in one layer. This is not satisfactory since our goal is to find a generally applicable method that can accelerate all layers throughout the model. Moreover, this amplifies the difference between different methods, helping to better illustrate the effects of each of the design choices.

7.5.1 Top-K

We first evaluate the Top-K method (Sec. 7.2.1). This standalone evaluation is important as Top-K is the basis of both the PKM and the MoE approximations. Tab. 7.1 shows the results. We observe that not only Top-K in the MLP blocks preserves the performance of Transformers, it even improves performance. We hypothesize that these improvements are due to the reduction in feature interference as described by Elhage et al. [2022]. However, we obviously cannot arbitrarily reduce K; there should be a trade-off between the denoising effect and the capacity of the network. Here, the optimal value we find is $K = 128$, independently of model size and dataset.

Dataset	#params	d_{ff}	K	bpc/perplexity
Enwik8	41M	2053	-	1.08
	41M	2053	128	1.07
	41M	2053	256	1.08
	41M	2053	512	1.08
WikiText 103	47M	2053	-	11.81
	47M	2053	64	11.86
	47M	2053	128	11.74
	47M	2053	256	11.74
	47M	2053	512	11.68
WikiText 103	262M	4110	-	9.46
	262M	4110	128	9.26
	262M	4110	256	9.34
	262M	4110	512	9.36

Table 7.1: Effects of the top-k activation function on the perplexity (WikiText-103) and bits/character (Enwik8).

7.5.2 Product-Key Memory (PKM)

Our view of Sec. 7.2 suggests using a non-competitive activation such as ReLU instead of the softmax used in the original PKM [Lample et al., 2019]. Our experiments confirm the benefits of this choice (Tab. 7.2): the performance of the ReLU variants is much closer to the dense baseline (see also related findings by Shen et al. [2023]). But even the best PKM models underperform the dense baselines, indicating the fundamental limitation of PKMs. Note that, as stated above, we conduct a careful comparison between the approximation method (here, PKM) and the dense baseline using the same number of parameters. For more results and details on PKM, we refer to App. F.1.3.

Variant	Nonlin	WT-S	WT-B	E8
Dense Baseline	ReLU	11.81	9.46	1.08
PKM	Softmax	13.96	11.10	1.16
	ReLU	12.77	9.98	1.11

Table 7.2: Performance of the parameter-matched PKM models. We provide more results in Appendix/Tab. F.1.

7.5.3 Mixture of Experts (MoE)

Here we evaluate our σ-MoE models (Sec. 7.4).

Main results. Tab. 7.3 shows the main results; our σ-MoE models match the performance of their parameter-equal dense baselines, while achieving significant memory and compute reduction. These models use $K = 4$ for $N_E = 16$ or $N_E = 32$, which is a "moderate" level of sparsity but already offering significant compute reduction as shown in the column "% FLOPs"; concrete compute and memory reduction is further shown in Fig. 7.2, (as well as Figs. F.5 and F.6 in the appendix). Naturally, there is a limit on the minimum sparsity level to preserve good performance of MoEs, which is determined by several factors. First, we empirically find that experts with a group size of $G < 128$ generally degrade performance. Second, our benchmarks with the Top-K operation (Tab. 7.1) and our ablations (Tab. F.4 in the Appendix) show that the minimum number of simultaneously active channels $G \cdot K$ need to be above a certain critical threshold (usually around 256-512). Finally, we match the number of parameters of the baseline model; this is the last constraint. Under these constraints, we find that the performance of dense baselines can be matched using 25% of the required FLOPs and memory for activations for our small models, and 12.5% sparsity for the big one (note that the FLOPs here do not take into account the linear projection used to select the experts, which is negligible within the range of N_E used here).

Increasing N_E and Impact of Sparsity. The above results demonstrate that our σ-MoEs can be configured to match the desired performance with fewer resources. Here, we conduct an extra experiment where we naively increase N_E (while keeping $K = 4$) from 16 to 128. This increases the number of parameters to 238M, while keeping the speed and memory requirements comparable to the original model (column "WT-S*" in Tab. 7.4). This model achieves a test perplexity of 10.37, which is worse than 9.46 of the 262M dense model (see Tab. 7.1). Indeed, even when the parameter count is matched, there

Dataset	Model	#params	% FLOPs	bpc/ppl
Enwik8	Dense	41M	100.0%	1.08
	σ-MoE	41M	25.0%	1.08
WikiText-103	Dense	47M	100.0%	11.81
	σ-MoE	47M	25.0%	11.71
WikiText-103	Dense	262M	100.0%	9.46
	σ-MoE	262M	12.5%	9.44

Table 7.3: Performance of parameter-batched σ-MoEs on perplexity (WikiText-103) and bits/character (Enwik8).

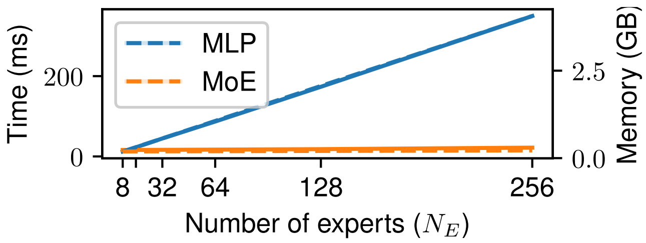


Figure 7.2: Execution time and memory usage of a forward-backward pass of a single MLP and MoE layer. $|\mathcal{B}| = 32768$, corresponding to a batch size 64 and sequence length 512, $d_{\text{model}} = 512$, $K = 4$, and $d_{\text{ff}} = G \cdot N_E$. Full/dashed lines show the execution time/memory consumption, respectively. As they are both linear with similar slopes, they are almost indistinguishable. Even our sub-optimal CUDA kernel is faster starting from 16 experts. Measured on an RTX 3090 with PyTorch 2.01 and CUDA 11.

are other bottlenecks that are crucial, e.g., here d_{model} is much smaller (412 vs. 1024). We construct another dense baseline by setting every hyperparameter like in the 47M model, except d_{ff}, which is set to 16480 to match the number of parameters of the $N_E = 128$ MoE. This baseline achieves a perplexity of 10.03: thus, the gap between the scaled-up MoE and its dense counterpart still remains significant (10.37 vs 10.03), unlike with the MoE with moderate sparsity. This indicates the importance of controlling MoE sparsity to preserve its performance against the dense baseline.

Comparison to Existing MoEs. We also compare our σ-MoE to other MoE variants (Sec. 7.3), namely Switch Transformer [Fedus et al., 2022], S-BASE [Clark et al., 2022][5] and the basic softmax variant. Tab. 7.4 shows the results.

[5] Unlike the original ones, our implementation does not enforce capacity factor-based hard

As Switch Transformer and S-BASE select only one single expert ($K = 1$), we increase the expert size by a factor of 4 (instead of $G = 128$ in our models, we use $G = 512$), and we decrease N_E by the same factor for fair comparison in terms of the parameter count. Neither of them uses our proposed expert dropout. For Switch Transformer, we test a variant with standard dropout in the experts (see App. F.2 for details), and a version without. We also extend S-BASE to $K = 4$, which is similar to ours, except for the balancing method. Even considering all these cases, our σ-MoE outperforms Switch Transformer and S-BASE.

Ablation Studies. Finally, we conduct ablation studies of individual design choices (Sec. 7.4). Tab. 7.4 shows the results. Standard dropout instead of expert dropout leads to performance degradation for most of the cases, except the model with $N_E = 128$ experts. The softmax-based selection functions (with and without re-re-normalization) consistently perform worse than our sigmoid one. The same is true for standard initialization ; ours is better. Interestingly, removing all regularization methods degrades performance but does not entail catastrophic collapse even with $N_E = 128$. We also examine the best (G, K) combinations, given a constant number ($G \cdot K$) of active pairs $\boldsymbol{k}_i$, $\boldsymbol{v}_i$; we find that high $K = 4$ works best within this range. Further analysis of our σ-MoE can be found in App. F.1.4.

Analyzing expert utilization. A typical failure mode of MoEs is the expert collapse, where only a few experts are used and the others are completely ignored or underused. We asked the question to what extent our method is affected by this issue. Thus, we computed the total proportion of the expert selection weights ($\mathrm{sel}(\boldsymbol{x})$) that the individual experts are assigned to on the entire validation set of WikiText 103. A representative layer is shown in Fig. 7.3. We use our WT-S* model from Tab. F.4, with 128 experts. Models with big performance gap in Tab. F.4 (Switch Transformer and ablation of σ-MoE with a softmax and renormalization, "softmax (renom.)") can be easily distinguished based on the selection statistics, showing that the expert collapse issue is at the heart of the problem. The other methods perform comparably, and the fine differences between them are not explained by expert collapse. In fact, for the least used experts, σ-MoE shows more collapse, yet it performs better than the other methods. Moreover, our perplexity-regularized models with expert dropout, especially with sigmoid activation function, are capable of matching the balancing effect of S-BASE without using the Sinkhorn activation function. In general, we do not consider uniform expert activation to be optimal: we expect expert specialization, and thus the frequency of their usage should

balancing.

Dataset	WT-S	WT-S*	WT-B	E8
# params. (in M)	47	238	262	41
Switch Transformer	12.27	11.24	9.68	1.08
no dropout	11.88	11.10	9.77	1.10
S-BASE ($K=4$, $G=128$)	13.01	10.96	10.50	1.17
$K=1, G=512$	12.32	11.31	9.77	1.32
σ-MoE ($K=4$, $G=128$)	11.59	10.37	9.44	1.08
standard dropout	12.01	10.27	9.53	1.08
softmax (renorm.)	11.89	11.27	9.58	1.09
softmax (no renorm.)	12.05	10.54	9.62	1.09
standard init	11.80	10.59	9.67	1.08
no regularization	11.83	10.41	9.51	1.08
$K=8, G=64$	11.63	10.30	9.58	1.08
$K=2, G=256$	11.84	10.44	9.56	1.09
$K=1, G=512$	11.90	10.83	9.58	1.09

Table 7.4: Ablation studies. WT-S* is obtained by naively scaling N_E in WT-S. More details in Sec. 7.5.3 & Tab. F.4.

depend on the occurrence of the task they are performing.

7.6 Limitations

Our experiments show that if we naively increase the number of experts, the performance gap between MoE models and their dense counterparts increases. This indicates the need for careful control of sparsity and hyperparameters, which remains a challenge for MoEs.

Our CUDA kernel is suboptimal and I/O limited. However, even in its current form, it already yields significant performance boosts and memory reduction. We expect that an expert CUDA programmer could improve the speed of our kernel by at least a factor of 2.

We do not consider load balancing between hardware accelerators as is done in Switch Transformers and S-BASE. Our goal is to make a larger model fit a single accelerator, or multiple accelerators, in the standard data-parallel training. Our preliminary experiments suggest that such balancing entails a performance hit.

We could not reproduce the 277M Enwik8 model of Dai et al. [2019], because we were unable to fit the baseline model on any of our machines. We

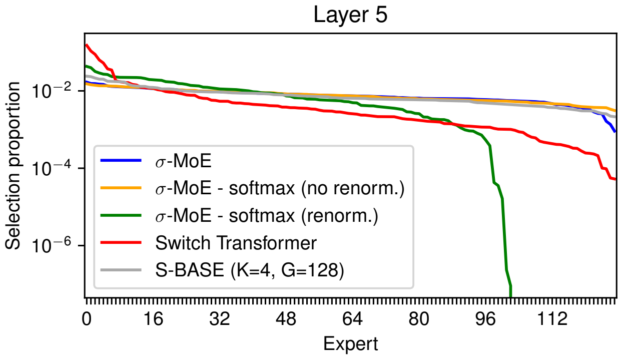


Figure 7.3: The total proportion of selection weights assigned to a given expert (indicated on the x-axis) on the validation set of Wikitext-103 with our WT-S* model from Tab. F.4. Experts are sorted by their popularity. A representative layer of different models are shown. The models with a big performance gap can be distinguished easily (Switch Transformer and σ-MoE with a softmax and renormalization, “softmax (renom.)”). Their performance gap can be at least partially attributed to expert collapse. However, it seems to be difficult to distinguish the fine performance difference between the rest of the models based solely on the expert collapse phenomenon. Similar plots for all layers of the network are shown in Fig. F.4 in the Appendix.

tried to use rotary positional encodings with PyTorch 2.0's memory-efficient attention to reducing its memory consumption; however, this resulted in a significant performance degradation (even for the smaller models).

Our study focuses on end-to-end trainable MoEs. Other MoE methods [Irie et al., 2018; Li et al., 2022a] that pre-train LMs on disjoint data, to recombine them later into a single model, are out-of-scope.

Our study only considers standard Transformers; however, similar acceleration methods are of utmost importance for shared-layer Transformers, such as Universal Transformers [Dehghani et al., 2019] and NDRs [Csordás et al., 2022a]. In fact, layer sharing dramatically reduces the number of parameters. Compensating for this by naively increasing d_{model} or d_{ff} results in prohibitively high memory overhead and slow execution. In contrast, MoEs allow for an increase in the number of parameters without such dramatic drawbacks. We leave shared-layer MoEs for future work.

7.7 Conclusion

Our novel view unifies methods that approximate 2-layer MLPs, such as Top-K, Mixture of Experts (MoE) and product-key memory (PKM) methods. While Top-K by itself provides limited performance improvements and speedups, further speedup requires PKM or MoE. A non-competitive activation function inspired by our unified view improves both PKM and MoE. Further novel enhancements of MoEs yield our σ-MoE which outperforms existing MoEs. A σ-MoE with moderate sparsity performs as well as parameter-equal dense baselines while being much more resource efficient. Our new insights improve the training of language models with limited hardware resources, making language modeling research more accessible.

Chapter 8

Conclusion and Future Work

In this dissertation, we have argued about the importance of learning algorithmic solutions instead of pure pattern matching and memorization. We discussed the associated difficulties and the importance of inductive biases. We started by improving one of the most promising architectures of the time [Csordás and Schmidhuber, 2019], which led to better performance in reasoning tasks. Later, we analyzed whether neural networks learn reusable modules [Csordás et al., 2021], and found that they tend to resist reusing knowledge, which is crucial for compositional behavior. Motivated by this and by the intuition that the structure of the transformers seems to be well suited to represent computational graphs, we managed to improve them significantly on a series of problems requiring generalization [Csordás et al., 2021]. We did this by adhering to the intuition introduced in Sec. 1.3, introducing relative positional encodings and shared layers. However, we found that these improvements are not enough to achieve length generalization in algorithmic tasks. To fix this, we proposed an extension of the transformer architecture, which, to our knowledge, is the first fully neural architecture capable of length generalization on CTL, Simple arithmetics and ListOps datasets [Csordás et al., 2022a]. We continued by analyzing the systematicity of the transformers and proposed a dataset to diagnose their behavior [Csordás et al., 2022b]. We found that transformers struggle even with basic generalization. Finally, motivated by the need to scale up shared layer transformers and NDR, we focused on improving the speed and memory requirements of transformers using MoEs. We showed that with a few simple modifications, they can save a significant amount of computation and improve the execution speed [Csordás et al., 2022b]. We believe that our work provides a significant contribution to systematic generalization research and opens up interesting further research directions.

8.1 Future Directions

Accelerating shared layer transformers and NDRs. The obvious way to go forward is to apply the acceleration methods discussed in Sec. 7 to the shared layer transformers and NDRs. Unfortunately, they pose additional nontrivial challenges: e.g. the norm of the update change over layers, which causes the MoE selection mechanism to reuse the same module in each layer. Additionally, the total number of attention heads is also reduced by the factor of the number of layers. This significantly degrades performance. The obvious solution is to increase the total number of attention heads, which, however, results in a significant slowdown and increase in memory usage. Alternatively, a conditional, MoE-like attention could be developed. We found this to be a highly nontrivial challenge.

Avoiding autoregressive masking. We hypothesize that autoregressive transformers would perform poorly in generalization tasks even with all the improvements introduced by NDR: the autoregressive mask does not permit building a computation graph in the layers of the transformers as discussed in Sect. 5. The information can flow just from left to right, meaning that only a single column of a transformer is available to store all changes in the state introduced by any new input token. Because of this, further research is needed in parallel decoding techniques, or a novel memory and processing structure, which can enable processing multiple columns at once. Alternatively, the advantages of combining NDRs with block-recurrent transformers [Hutchins et al., 2022] should be considered.

Evaluating the resulting models on real-world datasets. We believe that the above directions provide the minimal set of modifications required for transformers to prefer learning more algorithmically. After implementing these changes, the models should be thoroughly evaluated in reasoning tasks. It is reasonable to believe that pre-training on a large amount of language data helps in improving systematicity. However, if models are pre-trained, special care must be taken to avoid dataset leaks.

Improving systematicity of neural networks. Systematicity proved to be more difficult compared to productivity. It is not clear how this can be improved. Perhaps discretizing the representations on certain points in the network could improve the interoperability of the learned operations.

Appendix A

Further Details on Improving Differentiable Neural Computers

A.1 Implementation Details

Here we present the equations for our full model (DNC-MDS). The other models can be easily implemented according to the details in Sec. 2.2. We also highlight the differences to the DNC by Graves et al. [2016]. We try to keep our notation close to that of Graves et al. [2016].

Memory in step t is represented by matrix $\boldsymbol{M_t} \in \mathbb{R}^{N \times W}$, where N is the number of cells, and W is the length of the memory word. The network receives an input $\boldsymbol{x}_t \in \mathbb{R}^X$ and produces output $\boldsymbol{y}_t \in \mathbb{R}^Y$. The network controller receives the input vector $\boldsymbol{x}_t$ concatenated with all R (number of read heads) read vectors $\boldsymbol{r}_{t-1}^1, ..., \boldsymbol{r}_{t-1}^R$ from the previous step, and produces the output vector $\boldsymbol{h}_t \in \mathbb{R}^S$. The controller can be an LSTM or feedforward network and may have one or multiple layers. The controller's output is projected to the interface vector $\boldsymbol{\xi}_t$ by the matrix $\boldsymbol{W_\xi} \in \mathbb{R}^{2(W \cdot R)+4W+7R+3 \times S}$ by $\boldsymbol{\xi}_t = \boldsymbol{W_\xi h}_t$. An intermediate output vector $\boldsymbol{v}_t \in \mathbb{R}^Y$ is also generated: $\boldsymbol{v}_t = \boldsymbol{W}_y \boldsymbol{h}_t$. The output interface vector is split into many sub-vectors controlling various parts of the network:

$$\begin{aligned}\boldsymbol{\xi}_t = \boldsymbol{k}_t^{r,1}..\boldsymbol{k}_t^{r,R}|\hat{\beta}_t^{r,1}..\hat{\beta}_t^{r,R}|\boldsymbol{k}_t^w|\hat{\beta}_t^w|\hat{\boldsymbol{e}}_t|\boldsymbol{v}_t|\hat{f}_t^1..\hat{f}_t^R|\hat{g}_t^a|\hat{g}_t^w|\hat{\boldsymbol{\pi}}_t^1..\hat{\boldsymbol{\pi}}_t^R|\\ \hat{\boldsymbol{m}}_t^w|\hat{\boldsymbol{m}}_t^{r,1}..\hat{\boldsymbol{m}}_t^{r,R}|\hat{s}_t^{f,1}..\hat{s}_t^{f,R}|\hat{s}_t^{b,1}..\hat{s}_t^{b,R}\end{aligned} \quad \text{(A.1)}$$

Notation: $1 \leq i \leq R$ is the read head index; $\boldsymbol{k}_t^{r,i} \in \mathbb{R}^W$ are the keys used for read content-based address generation; $\beta_t^{r,i} = oneplus(\hat{\beta}_t^{r,i})$ are the read key strengths ($oneplus(x) = 1 + log(1 + e^x)$); $\boldsymbol{k}_t^w \in \mathbb{R}^W$ is the query used for content-based address generation for writes; $\beta_t^w = oneplus(\hat{\beta}_t^w)$ is the write key strength; $\boldsymbol{e}_t = \sigma(\hat{e}_t), \boldsymbol{e}_t \in \mathbb{R}^W$ is the erase vector which acts as an in-cell gate for memory writes; $\boldsymbol{v}_t \in \mathbb{R}^W$ is the write vector which is the actual data being written; $f_t^i = \sigma(\hat{f}_t^i)$ are the free gates controlling whether to de-allocate the cells read in the previous step; $g_t^a = \sigma(\hat{g}_t^a)$ is the allocation gate; $g_t^w = \sigma(\hat{g}_t^w)$ is the write gate; $\boldsymbol{\pi}_t^i = \mathrm{softmax}(\hat{\boldsymbol{\pi}}_t^i), \boldsymbol{\pi}_t^i \in \mathbb{R}^3$ are the read modes (controlling whether to use temporal links or content-based look-up distribution as read address); $s_t^{f,i} = oneplus(\hat{s}_t^{f,i})$ are the forward sharpness enhancement coefficients; $s_t^{b,i} = oneplus(\hat{s}_t^{b,i})$ are the backward sharpness enhancement coefficients.

Special care must be taken with the range of the masks $\boldsymbol{m}_t^w$ and $\boldsymbol{m}_t^{r,i}$. They must be limited to $(\delta, 1)$, where δ is a small real number. A δ close to 0 might harm gradient propagation by blocking gradients of masked parts of the key and memory vector.

$$\boldsymbol{m}_t^w = \sigma(\hat{\boldsymbol{m}}_t^w) * (1-\delta) + \delta \qquad \boldsymbol{m}_t^{r,i} = \sigma(\hat{\boldsymbol{m}}_t^{r,i}) * (1-\delta) + \delta \tag{A.2}$$

We suggest initializing biases for $\hat{\boldsymbol{m}}_t^w$ and $\hat{\boldsymbol{m}}_t^{r,i}$ to 1 to avoid low initial gradient propagation.

Content-based lookup is used to generate an address distribution based on matching a key against memory content:

$$C(\boldsymbol{M}, \boldsymbol{k}, \beta, \boldsymbol{m})[i] = \mathrm{softmax}(D\left(\boldsymbol{k} \odot \boldsymbol{m}, \boldsymbol{M} \odot \mathbf{1}\boldsymbol{m}^T)\beta\right) \tag{A.3}$$

Compare this with $C(\boldsymbol{M}, \boldsymbol{k}, \beta, \boldsymbol{m})[i] = \mathrm{softmax}(D\left(\boldsymbol{k}, \boldsymbol{M}\right)\beta)$ of Graves et al. [2016].

Where D is the row-wise cosine similarity to numerical stabilization:

$$D(\boldsymbol{u}, \boldsymbol{M})[i] = \frac{\boldsymbol{u} \cdot \boldsymbol{M}[i, \cdot]}{|\boldsymbol{u}||\boldsymbol{M}[i, \cdot]| + \epsilon} \tag{A.4}$$

The memory is first written to, then read from. To write the memory, allocation, and content-based lookup distributions are needed. The allocation is calculated based on usage vectors $\boldsymbol{u}_t$. These are updated with the help of memory retention vector $\boldsymbol{\psi}_t$:

$$\boldsymbol{\psi}_t = \prod_{i=1}^{R} \left(\mathbf{1} - f_t^i \boldsymbol{w}_{t-1}^{r,i}\right) \tag{A.5}$$

$$\boldsymbol{u}_t = \left(\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^w - \boldsymbol{u}_{t-1} \odot \boldsymbol{w}_{t-1}^w\right) \odot \boldsymbol{\psi}_t. \tag{A.6}$$

Operation $\odot$ is the elementwise multiplication. The free list $\boldsymbol{\phi}_t = \arg\mathrm{sort}(\boldsymbol{u}_t)$ is the list of indices of the memory locations sorted in ascending order of their usage $\boldsymbol{u}_t$. So $\boldsymbol{\phi}_t[1]$ is the index of the least used location. Then allocation address distribution $\boldsymbol{a_t}$ is

$$\boldsymbol{a}_t[\boldsymbol{\phi}_t[j]] = (1 - \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]]) \prod_{i=1}^{j-1} \boldsymbol{u_t}[\boldsymbol{\phi}_t[i]]$$

The write address distribution $w_t^w \in [0,1]^N$ is:

$$\boldsymbol{c}_t^w = C\left(\boldsymbol{M}_{t-1}, \boldsymbol{k}_t^w, \beta_t^w, \boldsymbol{m}_t^w\right) \tag{A.7}$$

$$\boldsymbol{w}_t = g_t^w \left[g_t^a \boldsymbol{a}_t + (1 - g_t^a)\boldsymbol{c}_t^w\right]$$

Memory is updated by ($\mathbf{1} \in \mathbb{R}^N$ is a vector of ones, $\boldsymbol{E} \in \mathbb{R}^{N \times W}$ is a matrix of ones):

$$\boldsymbol{M}_t = \boldsymbol{M}_{t-1} \odot \boldsymbol{\psi}_t \mathbf{1}^T \odot \left(\boldsymbol{E} - \boldsymbol{w}_t^w \boldsymbol{e}_{\boldsymbol{t}}^\intercal\right) + \boldsymbol{w}_t^w \boldsymbol{v}_t^\intercal \tag{A.8}$$

Compare this to the $\boldsymbol{M}_t = \boldsymbol{M}_{t-1} \odot \left(\boldsymbol{E} - \boldsymbol{w}_t^w \boldsymbol{e}_{\boldsymbol{t}}^\intercal\right) + \boldsymbol{w}_t^w \boldsymbol{v}_t^\intercal$ of Graves et al. [2016].

To track the temporal distance of memory allocations, a temporal link matrix $\boldsymbol{L}_t \in [0,1]^{N \times N}$ is maintained. It is a continuous adjacency matrix. A helper quantity called precedence weighting is defined: $\boldsymbol{p}_0 = \mathbf{0}$ and

$$\boldsymbol{p}_t = \left(1 - \sum_i \boldsymbol{w}_t^w[i]\right) \boldsymbol{p}_{t-1} + \boldsymbol{w}_t^w$$

$$\boldsymbol{L}_0[i,j] = 0 \;\; \forall i,j \qquad \boldsymbol{L}_t[i,i] = 0 \;\; \forall i$$

$$\boldsymbol{L}_t[i,j] = \left(1 - \boldsymbol{w}_t^w[i] - \boldsymbol{w}_t^w[j]\right) \boldsymbol{L}_{t-1}[i,j] + \boldsymbol{w}_t^w[i]\boldsymbol{p}_{t-1}[j] \tag{A.9}$$

Forward and backward address distributions are given by $\boldsymbol{f}_t^i$ and $\boldsymbol{b}_t^i$:

$$\boldsymbol{f}_t^i = S\left(\boldsymbol{L}_t \boldsymbol{w}_{t-1}^{r,i}, s_t^{f,i}\right) \qquad \boldsymbol{b}_t^i = S\left(\boldsymbol{L}_t^\intercal \boldsymbol{w}_{t-1}^{r,i}, s_t^{b,i}\right) \qquad S(\boldsymbol{d}, s)_i = \frac{\left(\frac{\boldsymbol{d}_i + \epsilon}{\max\left(\boldsymbol{d} + \epsilon\right)}\right)^s}{\sum_j \left(\frac{\boldsymbol{d}_j + \epsilon}{\max\left(\boldsymbol{d} + \epsilon\right)}\right)^s} \tag{A.10}$$

Compare this to the $\boldsymbol{f}_t^i = \boldsymbol{L}_t \boldsymbol{w}_{t-1}^{r,i}$ and $\boldsymbol{b}_t^i = \boldsymbol{L}_t^\intercal \boldsymbol{w}_{t-1}^{r,i}$ of Graves et al. [2016].

The read address distribution is given by:

$$\boldsymbol{c}_t^{r,i} = C\left(\boldsymbol{M}_t, \boldsymbol{k}_t^{r,i}, \beta_t^{r,i}, \boldsymbol{m}_t^{r,i}\right) \tag{A.11}$$

$$\boldsymbol{w}_t^{r,i} = \boldsymbol{\pi}_t^i[1]\boldsymbol{b}_t^i + \boldsymbol{\pi}_t^i[2]\boldsymbol{c}_t^{r,i} + \boldsymbol{\pi}_t^i[3]\boldsymbol{f}_t^i \tag{A.12}$$

Finally, memory is read, and the output is calculated:

$$\boldsymbol{y}_t = \boldsymbol{v}_t + \boldsymbol{W}_r \left[\boldsymbol{r}_t^1; ...; \boldsymbol{r}_t^R\right] \qquad \boldsymbol{r}_t^i = \boldsymbol{M}_t^\intercal \boldsymbol{w}_t^{r,i}$$

A.2 Hyperparameters for the Experiments

Copy Task. We use an LSTM controller with hidden size 32, memory of 16 words of length 16, 1 read head. W is 8, with the 9th bit indicating the start of the repeat phase. L is randomly chosen from range $[1, 8]$, N from range $[2, 14]$. Batch size is 16.

Associative Recall Task. We use a single layer LSTM controller (size 128), memory of 64 cells of length 32, 1 read head. $W_b = 3$, $B \in [2, 16]$, $W_b = 8$, batch size of 16.

Key-Value Retrieval Task. We use a single layer LSTM controller of size 32, 16 memory cells of length 32, 1 read head. $W = 8$, $L \in [2, 16]$.

bAbI. Our network has a single layer LSTM controller (hidden size of $S = 256$), 4 read heads, word length of 64, and 256 memory cells. Embedding size is $E = 256$, batch size is 2.

Appendix B

Additional Details for Inspecting the Implicit Modularity of Neural Networks

B.1 Derivations

B.1.1 From Gumbel-Softmax to Gumbel-Sigmoid

In what follows, we use the notation by Jang et al. [2017]: k is the number of categories, class probabilities are $\boldsymbol{\pi}_i$, $i \in \{1..k\}$, $\boldsymbol{y} \in \mathbb{R}^k$ is a sample vector from the Gumbel-Softmax distribution (also called Concrete distribution by Maddison et al. [2017]). Individual components of $\boldsymbol{y}$ are denoted by $\boldsymbol{y}_i$. We will refer to $\boldsymbol{l}_i = \log \boldsymbol{\pi}_i$ as logits. We show how to sample from the Gumbel-Sigmoid distribution, the special case of $k = 2$, $\boldsymbol{l}_2 = 0$ of the Gumbel-Softmax distribution.

First, we show that the sigmoid is equivalent to a first element $\boldsymbol{y}_1 \in \mathbb{R}$ of the output vector of the two element softmax with $\boldsymbol{l}_1 = x$, $x \in \mathbb{R}$ and $\boldsymbol{l}_2 = 0$:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} = \frac{e^x}{e^x+e^0} = \frac{e^{\boldsymbol{l}_1}}{e^{\boldsymbol{l}_1}+e^{\boldsymbol{l}_2}} = \boldsymbol{y}_1. \tag{B.1}$$

According to Jang et al. [2017], the sample vector $\boldsymbol{y} \in \mathbb{R}^k$ from the Gumbel-Softmax distribution can be drawn as follows:

$$\boldsymbol{y}_i = \frac{e^{\frac{1}{\tau}(\boldsymbol{l}_i+\boldsymbol{g}_i)}}{\sum_{j=1}^{k} e^{\frac{1}{\tau}(\boldsymbol{l}_j+\boldsymbol{g}_j)}}, \tag{B.2}$$

where $\boldsymbol{g}_i \sim \mathrm{Gumbel}(0, 1)$ are independent samples from the Gumbel distribution. We are interested in the special case of a sigmoid, which we showed to be equivalent to the $\boldsymbol{y}_1$ in $k = 2$, $l_2 = 0$ case:

$$\boldsymbol{y}_1 = \frac{e^{\frac{1}{\tau}(\boldsymbol{l}_1+\boldsymbol{g}_1)}}{e^{\frac{1}{\tau}(\boldsymbol{l}_1+\boldsymbol{g}_1)} + e^{\frac{1}{\tau}\boldsymbol{g}_2}}. \tag{B.3}$$

This can be rearranged as:

$$\boldsymbol{y}_1 = \frac{1}{1 + e^{-\frac{1}{\tau}(\boldsymbol{l}_1+\boldsymbol{g}_1-\boldsymbol{g}_2)}} = \sigma\left(\frac{1}{\tau}\left(\boldsymbol{l}_1 + \boldsymbol{g}_1 - \boldsymbol{g}_2\right)\right). \tag{B.4}$$

Writing out the inverse transformation sampling formula for $\boldsymbol{g}_i \sim \mathrm{Gumbel}(0, 1)$; $\boldsymbol{g}_i = -\log(-\log \boldsymbol{U}_i)$, were $\boldsymbol{U}_i \sim \mathrm{U}(0, 1)$ are independent samples from the uniform distribution, we get:

$$\begin{aligned}\boldsymbol{y}_1 &= \sigma\left(\frac{1}{\tau}\left(\boldsymbol{l}_1 - \log(-\log \boldsymbol{U}_1) + \log(-\log \boldsymbol{U}_2)\right)\right)\\ &= \sigma\left(\frac{1}{\tau}\left(\boldsymbol{l}_1 - \log\frac{\log \boldsymbol{U}_1}{\log \boldsymbol{U}_2}\right)\right).\end{aligned} \tag{B.5}$$

Finally, by renaming $s = \boldsymbol{y}_1$ and $l = \boldsymbol{l}_1$ (we have just a single logit), we obtain the sampling formula for Gumbel-Sigmoid:

$$s = \sigma\left(\frac{1}{\tau}\left(l - \log\frac{\log \boldsymbol{U}_1}{\log \boldsymbol{U}_2}\right)\right). \tag{B.6}$$

B.1.2 Straight-Through Estimator

Samples from the Gumbel-Softmax distribution can directly be converted to a sample from the categorical distribution as:

$$\boldsymbol{c}_i = \mathbb{1}_{i=\arg\max_j y_j} \tag{B.7}$$

Applying the straight-through estimator [Hinton, 2012; Bengio et al., 2013b] can provide 'hard' samples while permitting gradient flow ($[\cdot]_{\text{stop}}$ is an operator for blocking backward gradient flow):

$$\boldsymbol{c}_i = [\mathbb{1}_{i=\arg\max_j \boldsymbol{y}_j} - \boldsymbol{y}_i]_{\text{stop}} + \boldsymbol{y}_i \tag{B.8}$$

Since in the Gumbel-Sigmoid we have $k = 2$ categories and $\sum_i \boldsymbol{y}_i = 1$, the $\arg\max$ can be replaced by testing whether $\boldsymbol{y}_i > 0.5$:

$$\boldsymbol{c}_i = [\mathbb{1}_{\boldsymbol{y}_i>0.5} - \boldsymbol{y}_i]_{\text{stop}} + \boldsymbol{y}_i \tag{B.9}$$

Since we defined the sample s to be $\boldsymbol{y}_1$ (i.e. we are interested in the $i = 1$ case) the index i can be omitted. The variable has a Bernoulli distribution (see Sec. B.1.3), so we use a substitution $b = \boldsymbol{c}_1$ to get:

$$b = [\mathbb{1}_{s>0.5} - s]_{\text{stop}} + s \tag{B.10}$$

B.1.3 The Expected Value of the Samples

Let us analyze the distribution governing the samples b. Each sample is binary and independent since it is generated using independent samples from a uniform distribution. Since the sampling process is stationary they must therefore be Bernoulli distributed. Next, we show that their mean is $\mu = \sigma(l)$.

We are interested in:

$$\mu = P(b = 1) = P\left([\mathbb{1}_{s>0.5} - s]_{\text{stop}} + s = 1\right). \tag{B.11}$$

The straight-through estimator does not change the numerical value of b, so we can ignore it:

$$\mu = P(b = 1) = P\left(\mathbb{1}_{s>0.5} = 1\right) = P\left(s > 0.5\right), \tag{B.12}$$

where $s = \sigma\left(\frac{1}{\tau}\left(l - \log\frac{\log U_1}{\log U_2}\right)\right)$. Let us simplify the condition $s > 0.5$:

$$\sigma\left(\frac{1}{\tau}\left(l - \log\frac{\log U_1}{\log U_2}\right)\right) > 0.5 \tag{B.13}$$

σ is monotonically increasing and $\sigma(0) = 0.5$, so:

$$\frac{1}{\tau}\left(l - \log\frac{\log U_1}{\log U_2}\right) > 0. \tag{B.14}$$

By multiplying both sides with $\tau > 0$ and re-ordering we obtain:

$$l > \log\frac{\log U_1}{\log U_2}. \tag{B.15}$$

Since e^x is monotonically increasing, we can exponentiate both sides to obtain:

$$e^l > \frac{\log U_1}{\log U_2}. \tag{B.16}$$

The samples U_i are uniform random samples from the range $U_i \in (0, 1)$. Hence, it follows that $\log U_i < 0$. Multiplying both sides by $\log U_2$, we get:

$$e^l \log U_2 < \log U_1. \tag{B.17}$$

Exponentiating once again leads to:

$$U_2^{e^l} < U_1. \tag{B.18}$$

Let us return to the original problem, which is now takes a much simpler form:

$$\mu = P(b = 1) = P(U_2^{e^l} < U_1). \tag{B.19}$$

Using the definition of the mean, we get:

$$\mu = \mathbb{E}_{U_1 \sim \mathrm{U}(0,1), U_2 \sim \mathrm{U}(0,1)} \left[\mathbb{1}_{U_2^{e^l} < U_1} \right]. \tag{B.20}$$

Using the definition of expectation:

$$\mu = \int_{-\infty}^{\infty} P(U_2) \int_{-\infty}^{\infty} P(U_1) \mathbb{1}_{U_2^{e^l} < U_1} dU_1 dU_2. \tag{B.21}$$

Since $U_i \sim \mathrm{U}(0,1)$ are samples form uniform distribution with range $(0,1)$, $P(U_i) = 1$ in interval $U_i \in (0,1)$ and 0 otherwise. This enables us to change the boundaries of the integrals:

$$\mu = \int_0^1 \int_0^1 \mathbb{1}_{U_2^{e^l} < U_1} dU_1 dU_2. \tag{B.22}$$

Since the value of $\mathbb{1}_{U_2^{e^l} < U_1}$ is 1 when $U_1 > U_2^{e^l}$ and 0 otherwise, we can tighten the bounds of integration and eliminate the indicator function:

$$\mu = \int_0^1 \int_{U_2^{e^l}}^1 1 dU_1 dU_2 = \int_0^1 [U_1]_{U_2^{e^l}}^1 dU_2. \tag{B.23}$$

$$\begin{aligned}
\mu &= \int_0^1 1 - U_2^{e^l} dU_2 = 1 - \int_0^1 U_2^{e^l} dU_2 \\
&= 1 - \left[\frac{U_2^{e^l+1}}{e^l + 1} \right]_0^1 = 1 - \frac{1^{e^l+1}}{e^l+1} + \frac{0^{e^l+1}}{e^l+1} \\
&= 1 - \frac{1}{e^l+1} = \frac{e^l+1}{e^l+1} - \frac{1}{e^l+1} = \frac{e^l}{e^l+1} = \frac{1}{e^{-l}+1} \\
&= \sigma(l).
\end{aligned} \tag{B.24}$$

B.1.4 Choosing the Temperature

Notice that $\mu = \sigma(l)$ does not depend on the temperature τ (Appendix B.1.3). The binarized sample, b, will have the same output regardless of the value of τ, thus s will have the same gradients. The logit l, however, has a gradient scaled by $\frac{1}{\tau}$, but which can be mitigated by the normalization in the Adam optimizer. Thus, we can choose τ freely. We set $\tau = 1$ for all experiments of our paper.

B.2 Additional Discussion

B.2.1 Stability of the Masks

Multiple sources of randomness could affect the final masks discovered by our method. These include sampling the mask at each iteration, different data for each target task, and the order in which data is used for training. To verify that the masks discovered by our method are consistent we considered pairs of CIFAR10 classes as target tasks in combination with a simple CNN without dropout (Appendix B.3.8). Pairs are chosen instead of the leave-one-out scheme used in Sec. 3.4 to increase the sparsity of the masks as much as possible (potentially making them even more unstable). We trained 10 CNNs and analyzed 10 random pairs of classes for each of them. For each pair we trained two separate masks and calculate the Intersection over Union (IoU), resulting in a total of 100 data points.

We found that the mean IoU is $93.26 \pm 0.96\%$, which confirms the discovered masks' stability. Note that in case multiple redundant weight configurations are present in the network, different mask seeds will find a different subset of them, so their IoU will be less then 100% even in the optimally stable case. Using dropout would encourage such cases.

Potential Errors Introduced by the Straight-Through Estimator

The straight-through estimator introduces approximations in the gradient calculation. Fortunately, the inaccuracies do not build up through multiple estimation steps, since the masking and straight-through estimator are applied directly to the network's parameters. Indeed, on each gradient path, there is at most a single straight-through estimator present.

B.2.2 Does Masking Change the Performed Operation?

The recent work of [Zhou et al., 2019] demonstrated how it is possible to achieve non-trivial performance by training binary masks on a neural network with frozen weights that were randomly chosen. This raises the question whether the masking process in our method changes the performed operation after the weights are frozen, and thus could cause misleading observations.

To investigate this possibility, we randomly selected some of the networks and datasets used throughout the paper and trained as usual. After both the weights and the masks are learned, we performed the following experiment:

we applied masks to roughly half of the networks weights, while leaving the remainder unmasked. In one variant of this experiment, early layers near to the input are masked, while the later layers, including the output, are not. In the other variant, the opposite is true. If the network demonstrates compatibility between the masked and non-masked layers for these experiments, then this is a strong indication that it has not altered the performed operation significantly.

The outcome of these experiments are shown on Fig. B.1, where we report the performance drop for transformer on SCAN dataset, FNN on addition/multiplication dataset, LSTM, big (4 layers of 2000 units) and small (4 layers of 800 units) FNNs on the double-addition experiments and the small CNN on CIFAR 10.

For almost all configurations, we observe only a low drop in performance, indicating that the operations performed by the network remain mostly the same under the masking process. The only exception we found is the big FNN on the double-addition task, when the early layers are masked. Note, however, that its performance is well above the chance level ($P = 10^{-4}$). Since this network is severely overparametrized, we speculate how this might be the reason for this observed difference. For example, it could have learned to solve the problem by combining multiple alternative pathways, all of which contribute to the output. If the masking process removes some of those pathways from the layer near the input, but leaves them in the layers near the output, the pathways are cut in half. Thus, they might produce erroneous outputs. To further analyze this we therefore also trained a smaller version of the same network, which we observe behaves similarly to all other networks, suggesting that also in this case the masking does not alter the performed computation significantly.

Finally, we note the variant where the early layers are masked appears considerably more difficult than other way around. This might be because some of the inputs of the unmasked later layers are removed by masking the early layers. Normally, if all layers are masked as well, such weights of later layers would be removed together with the ones in the early layers, thus not affecting the result.

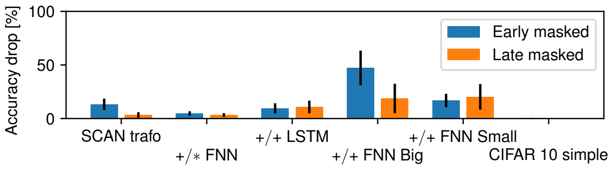


Figure B.1: Accuracy drop for masks applied to half of the weights. See Sec. B.2.2 for details.

There may be multiple reasons for these different findings compared to Zhou et al. [2019]. First, we use well-trained networks instead a randomly initialized ones. Untrained networks are believed to contain many random subnetworks which can useful for performing any task. However, the fully trained network has its subnetworks tuned to the task, likely decreasing the possibility of further subnetworks existing that implement radically different operations. Second, we are training the masks on a subset of the train set, which does not encourage changing the performed operations either: the highest performance can be achieved by selecting the correct subnetwork already performing the operation well. Finally, the experiments of Zhou et al. [2019] indicated that the performance of the best found network decreases with the task complexity, and performs best on MNIST. Some of the experiments considered here are significantly more complex.

B.2.3 Choosing Target Functionality

In principle, any operation that the network is able to perform can be used as a target functionality. This includes partitions of the dataset, or even novel tasks, if the network can generalize to them. The resulting masks will highlight which weights are responsible for performing them. For our experiments, we always chose a subset of the training set of the weights as target functionality. This ensures that no generalization is required to solve the problem and that the subset of the original weights required to solve this subproblem is highlighted. The discovered module then corresponds to functionality that the network should already have learned in the original training phase.

Interesting target functionality should be chosen such that removing the discovered set of weights or its inverse can be expected to lead to measurable performance difference on some test set. This test set should ideally be a subset of the original training set used for the weights. In this way, one does not have to directly consider the amount of sharing, and can measure (the difference in) accuracy, which we find more reliable and easy to interpret (see also Sec. 3.1 for further details) However, if such a choice is not available and the amount of sharing has to be analyzed directly, then we recommend drawing conclusions only when the observed difference compared to some reference score is sufficiently high. For example, in the Permuted MNIST experiments, the sharing of $< 20\%$ is significantly lower than the expected 100%.

B.2.4 Is Attention the Solution?

Could a form of attention [Schmidhuber, 1991, 1992c, 1993; Bahdanau et al., 2015] solve the problem discussed in Sec. 3.2.4? At least the current use of attention does not seem promising. In theory, attention-based Transformers [Vaswani et al., 2017] can reuse the same modules in parallel, but only if they are executed in the same layer. For $a * b + c * d$ the multiplier is reusable, but for $a*b*c$ it is not, since the second multiplication requires the result of the first; that is, different layers are needed. In recurrent models, such as RNNs with attention [Bahdanau et al., 2015] and Universal Transformers [Dehghani et al., 2019], attention does not permit routing between functional modules but is just used to route data to the input of a monolithic transformation block which processes the information. Emerging functional modules must in that case appear within the processing blocks. However, since attention is neither able to rewire the block's internal data flow, nor to permute elements of the attended vector, it does not help with the routing between modules emerging inside the block. Indeed, we showed empirically in Sec. 3.3 that Transformers suffer from the same generalization issues as LSTM on the SCAN dataset, and they did not generalize on the more complex Mathematics Dataset either. Attention might help though, if all the input and output interfaces of functional blocks overlap, and a single processing step executes a single function. However, as our experiments show, the separation between modules tends to be inadequate in the case of shared interfaces (Sec 3.2.1). Moreover, there is no control over executing a single function per time step (e.g., the whole $a * b * c$ block could be executed in a single step).

In order to help with data routing between emerging functional modules, attention must be able to focus on arbitrary parts of the activation vectors. This would enable information exchange between such modules, but it is unclear how this could be implemented. For example, in the double addition experiment (Sec. 3.2.2), the task requires to process disjoint subsets of the input, which is not possible with the attention mechanism. In general, attention-based solutions would require to store one "concept" in a single vector so that they can separately attended to. However, what makes a good "concept" in this case is unclear, especially since different processing stages might require a different granularity – for example, sorting tuples of numbers based on the first element of the tuple requires accessing individual elements of it but also treating the tuple as a whole. For a broader discussion on dynamic information routing and the problem of variable binding in neural networks, we refer the reader to Greff et al. [2020].

Despite these limitations, the models with attention still start from a better position than the ones without. If the computation block of the transformer can learn to conditionally read/write from the same memories in the column, and to copy data between different memories, in theory, it is possible to learn a behavior that generalizes. This does not happen in practice by default. Whether it can be enforced is an interesting research question. As we saw in Chapters 4 and 5, it is possible to some extent.

B.2.5 Explicitly Modular Networks

At first glance, explicitly modular networks [Clune et al., 2014; Andreas et al., 2016; Kirsch et al., 2018; Chang et al., 2019; Bahdanau et al., 2019b] could provide a solution for the discovered problems. In what follows, we will call hardcoded modules "blocks" to distinguish them from functional modules, which we call "modules". Routing networks, in addition to the problems described by Rosenbaum et al. [2019], are restricted to exchange information between blocks as a single fixed-size activation vector. Because all the information has to be stored in this vector (such as different variables), either different parts of the vector should be responsible for different stored variables or they have to be stored in superposition, e.g., by projecting them in a space where they are orthogonal to each other. Either way, this requires the blocks not only to perform a given operation, but also to be aware which variable they want to access. Thus, the blocks are not universal, since operating on different variables encoded in the state require different modules. For example, in the double addition experiment (Sec. 3.2.2), the task requires to process disjoint subsets of the input. This is true in general: Different subsets of the network state may require independent processing. Routing networks consist of simple modules stacked sequentially, which is obviously not a good fit for this type of data. Alternatively, RIMs [Mittal et al., 2020; Goyal et al., 2021b,a] attend to the data, making it possible to execute multiple modules in parallel. However, they are based on attention, which also has its limitations (Sec. B.2.4). These difficulties let us believe that a general inductive bias towards function-based specialization in generic neural networks would be a preferable solution compared to explicit modality and motivates this paper's topic.

B.3 Additional Results and Experimental Details

B.3.1 Sanity Checking the Mask Discovery Process

Our method frequently discovers a resistance against weight sharing. Perhaps this could raise the question whether our method is able to discover shared weights at all. We ran additional experiments to verify this.

We used the double-addition experiments from Sec. 3.2.2. Specifically, we trained the network as before, but after the weight training phase, we copy the input and output weights of one pair to the part of the weight matrix corresponding to the other. This ensures that the hidden layers cannot see any difference between the two pairs. We use the FNN variant since it can be used without any modification, while the LSTM would require changes to avoid state conflicts.

In Fig. B.2 it can be seen how our method accurately discovers that sharing is almost perfect in this case, which further justifies our approach. Compare this to the identical setup of Fig. 3.3. Note that the first and last layers are still not shared: they contain identical, but non-shared copies of the weights.

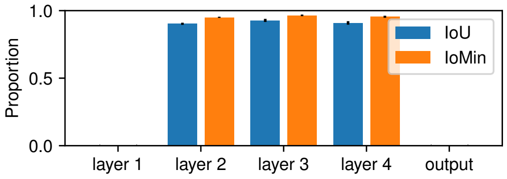
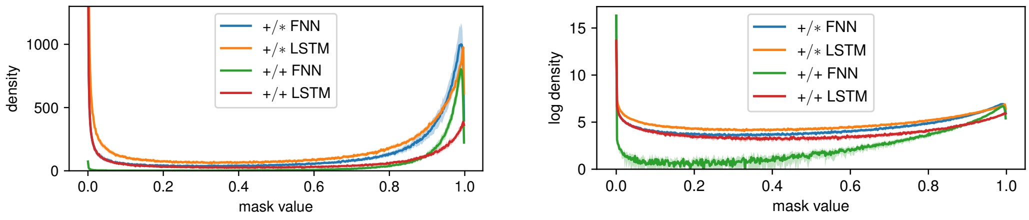


Figure B.2: Double addition task with manually edited input/output weight matrices to reuse the hidden layers. Proportion of weights shared per operation in case of FFN.

B.3.2 Common Hyperparameter Choices

Our method is implemented in PyTorch [Paszke et al., 2019], and available at `https://github.com/robertcsordas/modules`. Unless otherwise noted we use the Adam optimizer [Kingma and Ba, 2015], a batch size of 128, a learning rate of 10^{-3}, and gradient clipping of 1. To improve the quality of the masks we divide a batch into four parts that each act on a different mask sample. For non-LSTM networks, we use the ReLU activation function [Fukushima, 1969; von der Malsburg, 1973; Nair and Hinton, 2010] for the activations of intermediate layers. The Gumbel-sigmoid always has a temperature of $\tau = 1$ (the reason for this



(a) Linear scale, small values cut off

(b) Log scale, all values

Figure B.3: Histogram (normalized as a 500-bin PDF) of expected values of the mask elements ($\mu_i = \sigma(l_i)$) on different tasks. (a) Shown on a linear scale. Values < 0.0002 (bottom 10% of the first bin) are removed from the calculation because their number vastly exceeds the number of kept weights for most of the networks, making the details invisible. (b) All mask means, μ_i, (without small values removed) shown on log-scale.

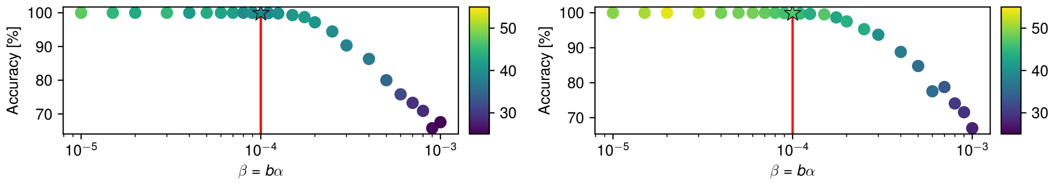
is explained in Appendix B.1.4). For most of our experiments, the regularization coefficient α is specified as $\beta = b\alpha$, where b is the batch size used for training the masks. Otherwise, we will mention α separately. All figures in this paper, unless noted otherwise, show mean and standard deviation calculated over 10 runs with different seeds.

B.3.3 Choosing the Regularization Hyperparameter

Choosing the regularization hyperparameter α is critical to obtain valid conclusions. Too low α might yield the false impression that no modules exist or that they share more weights than they really do. Too strong regularization may degrade performance on the target task, discarding essential weights.

Fortunately, there is a simple and consistent heuristic for choosing α, which follows from training a mask on the full task. We increase α as long as the performance does not start to drop. Then, we reduce α slightly until the performance is adequate, e.g. $> 95\%$ of the original performance. This method is not very sensitive to the *exact* value α and transfers well across different network sizes (Fig. B.4). We find that it is less critical but still important to tune the learning rate and the number of steps of the mask training process. We always check the chosen hyperparameters' validity by training a mask on the full, unmodified problem, where we expect to see only a slight drop in performance.

Note that underfitting NNs tend to share more weights. Indeed, in our experiments we found that choosing a sufficiently large network size is essential to



(a) Big network

(b) Medium network

Figure B.4: Sensitivity analysis for hyperparameter $\beta = b\alpha$ (b is the batch size) on addition/multiplication experiments. Note the logarithmic x-axis. The color indicates the total amount of sharing [%]. The red line and the star indicate the value chosen for our experiments. Each point is a mean of 10 independent seeds. The network is not very sensitive to the exact choice of β. (a) Big network, with 5 layers of size 2000. (b) Medium network, with 5 layers of size 800. It can be seen that the hyperparameter transfers well between network sizes.

avoid false conclusions about the reason for sharing. Fig. B.5 shows an example of how the amount of weight sharing changes as a function of network capacity.

B.3.4 Addition/Multiplication Experiments

Since preliminary experiments indicated that modulo 100 multiplication require lots of weights, we used reasonably large networks for this experiment. The FNN is 5 layers deep, each layer having 2000 units and the LSTM a hidden state size of 256 (further increase resulted in overfitting). A network was trained for 20k steps on the full task before freezing. The following mask training phase takes an additional 20k steps for each mask. Mask training uses a learning rate of 10^{-2} and $\beta = 10^{-4}$ for regularization. For the LSTM we use 3 time steps where the input is repeated for every step. The dataset is generated by sampling numbers and operations uniformly at random.

Fig. B.5 demonstrates that even if we use a large network (the small, 3 layer networks of 400, 400, 200 can solve the task), the percentage of shared weights still changes when increasing the network size.

In Sec. 3.2.1 we showed that even though there is a certain level of natural separation between the modules responsible for addition and multiplication, there is still a significant proportion of shared weights. To analyze the importance of those shared weights, we tested the network with inverted masks as in Sec. 3.2.2. Tab. B.1 shows the results. Given the high proportion of sharing, especially in the input and output layers, the results are as expected: inverted masks do not perform well, showing that the separation of the modules is limited.

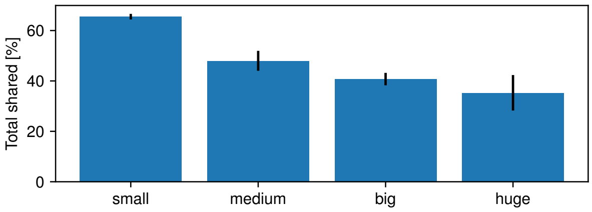


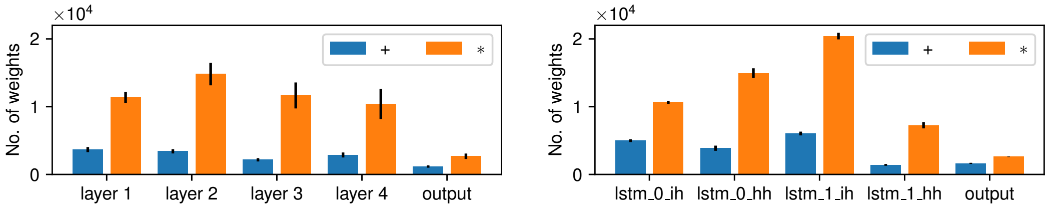
Figure B.5: Addition/multiplication task: the total proportion of shared weights for the "add" operation for different network sizes. "small" means a 4 layer network with hidden sizes of 400, 400, 200, "medium" 5 layers / hidden sizes of 800, "big" 5 layers / 2000, "huge" 5 layers / 4000.

		Full	$+$	$\neg+$	$*$	$\neg*$
FNN	$+$	100 ± 0.0	100 ± 0.0	13 ± 5.5	1 ± 0.0	20 ± 7.0
	$*$	100 ± 0.2	0 ± 0.0	69 ± 9.7	100 ± 0.0	17 ± 5.8
LSTM	$+$	100 ± 0.0	100 ± 0.0	2 ± 0.6	2 ± 0.6	1 ± 0.2
	$*$	100 ± 0.1	3 ± 1.2	6 ± 0.8	100 ± 0.0	2 ± 1.2

Table B.1: Accuracy of addition/multiplication task on addition and multiplication with FNN and LSTM. The header shows on what the applied mask was trained on. $\neg$ denotes an inverted mask

B.3.5 Double Addition Experiments

The training protocol for double-addition experiments is identical to the one in Appendix B.3.4, except that the FNN variant uses a mask regularizer of $\beta = 4 * 10^{-4}$. The LSTM uses 6 steps in total in this case (3 steps per operation). In the full input case, both tuples are presented for all 6 steps and the output is read from the last step. In the case where one tuple is presented at a time, the first tuple is shown for the first 3 steps, resulting in an output at the 3rd step, the second tuple is presented for the next 3 steps, resulting in an output at the 6th step.



(a) FNN

(b) LSTM

Figure B.6: Addition/multiplication taks: number of weights per operation for each layer in (a) feedforward network, (b) LSTM.

Additional Inverted Mask Experiments

Following the inverted-mask experiments done in Sec. 3.2.2, we investigated how well the separation holds if we consider only the hidden layers. This is achieved by using inverted masks for the hidden layers, while using the mask trained on the full task without inversion for the input and output layers. Hence, in this case the inputs and outputs contain all the connections needed for both tasks. Our findings are shown in Tab. B.2 and are consistent with Tab. 3.1. It can be seen how the performance of the inverted mask tends to work well on the opposite task, while its performance is significantly lower on the original task (note that chance is at $P = 0.01$ for these experiments), suggesting that this effect is not due to their inputs/outputs being disjoint.

Furthermore, we experimented with leaving the input and output layers unmasked, while inverting the discovered masks for the hidden layers. Surprisingly, in this case the inverted masks perform well on both tasks (around 90%), even on the task for which the mask was inverted. This suggests that the network contains an ensemble of subnetworks individually capable of solving the problem with good performance. However, based on the findings in Tab. B.2, these subnetworks in the hidden layers appear to be mostly independent of each other: the performance is non-zero on the original task *only* if both the original weights and the ones corresponding to the inverted masks are now included. It remains unclear what causes this particular behavior in this setting, which we believe is an interesting direction for future research.

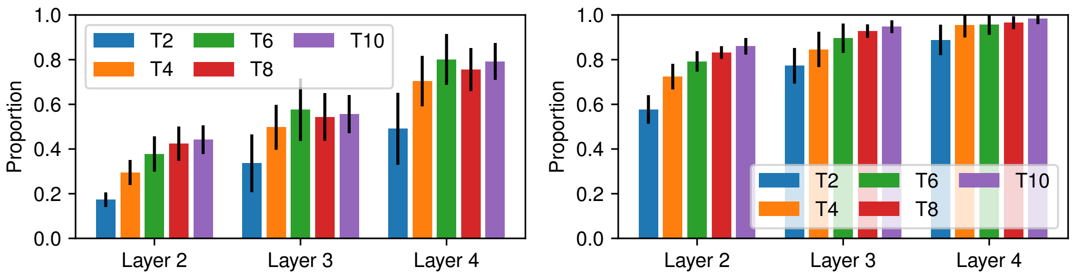
		Full	Pair 1	¬Pair 1	Pair 2	¬Pair 2
FNN	Pair 1	100 ± 0.4	100 ± 0.0	7 ± 4.0	1 ± 0.1	63 ± 15.9
	Pair 2	100 ± 0.1	1 ± 0.1	62 ± 16.9	100 ± 0.0	8 ± 5.0
LSTM	Pair 1	100 ± 0.0	100 ± 0.0	16 ± 4.1	1 ± 0.1	99 ± 1.3
	Pair 2	100 ± 0.0	1 ± 0.0	97 ± 4.9	100 ± 0.0	16 ± 5.9
LSTM (forced)	Pair 1	100 ± 0.0	100 ± 0.0	25 ± 6.1	1 ± 0.1	76 ± 10.0
	Pair 2	100 ± 0.1	1 ± 0.1	94 ± 4.2	100 ± 0.0	42 ± 14.7

Table B.2: Double-addition task: accuracy [%] of LSTMs and FNN on the two pairs. In case of LSTM (forced) only one input is presented at a time (to prevent interference). The header shows on which pair the mask was trained on. ¬ denotes an inverted mask for the hidden layers, while the regular mask (for the full task) is applied to the input and output layers. For further details, please refer to Sec. B.3.5

B.3.6 Transfer Learning Experiments

In the transfer learning setup, we train on 11 permutations of MNIST using the same network. Training the weights and masks together is more difficult than the usual setup. In order to improve the quality of the mask gradients we use 8 mask samples per batch instead of the standard 4. Each phase takes 30k steps. The learning rate is 10^{-2}. The network is 4 layers deep, with hidden sizes of 800, 800, 64. We are using a mask loss of $\alpha = 10^{-5}$.

Fig. B.7 demonstrates the number of shared weights per layer for a network that has its masks initialized so that it prefers to reuse the old weights. The mask logits corresponding to weights of the *previous task* are initialized to 2 (corresponding to $P \approx 0.88$), the logits for newly initialized weights to either 0 ($P = 0.5$, Fig. B.7a) or -1 ($P \approx 0.27$, Fig. B.7b). Compared to Fig. 3.4, the sharing is significantly increased.



(a) New weights with $P = 0.5$.

(b) New weights with $P \approx 0.27$.

Figure B.7: Proportion of weights shared per layer after every second task on permuted MNIST, for a network with masks initialized to prefer reusing the old weights. Old weights are sampled with $P \approx 0.88$ probability. Each task corresponds to a permutation. Decreasing the probability of new weights forces increased sharing.

B.3.7 Experiments on Algorithmic Tasks

SCAN Experiments

In preliminary experiments, we observed that the full-size word embeddings used for the baseline by Lake and Baroni [2018] yielded many possible redundant input-to-hidden weight configurations that have a greatly reduced probability of being sampled. This caused the input-to-hidden layer to be removed by the thresholding procedure. Therefore, we appropriately reduced the size of word embeddings to 16 (note that SCAN has only 13 input and 6 output tokens and they are not shared). When using the reduced embedding, we do not suffer from the aforementioned problems. Teacher forcing was used for each batch with 50% probability.

The Transformer network is based on PyTorch's internal implementation, with modifications needed to apply multiple masks more effectively. We use $d_{model} = 100$, inner-layer dimensionality of $d_{ff} = 200$, $h = 4$ heads, and 3 layers both in the encoder and decoder. The network is always trained with teacher forcing. An end token is applied to the end of each input sequence, and decoding starts with a start token. The sinusoidal positional embedding [Vaswani et al., 2017] is applied to the inputs of the transformer in both encoder and decoder.

The training procedure uses a batch size of 256 and a gradient clipping of 5. The mask learning rate is 10^{-2} and we use $\beta = 3{*}10^{-5}$ for the LSTM experiments and $\beta = 10^{-3}$ for the transformers. We train the networks for 25k steps without masks before freezing and for another 25k steps for each mask training phase.

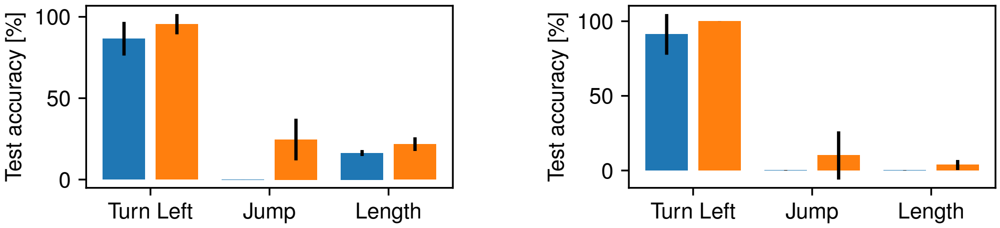
We train the network weights on the IID dataset (the "simple" split), and only the masks on the rest of the data splits. Fig. B.8 shows that this process marginally improves the performance of all splits, compared to when the network is directly trained on the corresponding train split. However, performance degrades significantly when removing weights that are unnecessary for the given training split. This allows us to conclude that the learned solution requires tasks-specific weights. Notice that the remaining weights still perform significantly better than training the network solely on the training set of the given split without masking.

The word embeddings are excluded from the masking process and remain unmodified after initial training to keep the learned word representations unchanged.

Experiments on the Mathematics Dataset

The Mathematics Dataset [Saxton et al., 2019] is a dataset intended to test the mathematical reasoning skills of NNs. It consists of 56 tasks from different areas of mathematics, on high school-level. Each task consists of a train set divided into 3 levels of difficulty (easy, medium, and hard) and an IID test set. All questions and answers are provided only in a human-readable text format (see examples in Fig. B.9).

We train the network on individual tasks, in contrast to the method of Saxton et al. [2019], where all tasks are trained together. The main reason for this is to



(a) LSTM

(b) Transformer

Figure B.8: The networks' performance when it is directly trained and tested on the splits indicated on the x-axis, without masking. This is the standard setup from Lake and Baroni [2018]. Performance when the network is trained on IID split, then masks are trained on train split indicated on the x-axis. It can be seen that although training on the IID set helps compared to the basic setup, the network still needs task-specific weights, which hurt performance when removed by the masks.

save computation and to prevent possible interference between the tasks. We chose five different tasks to analyze, based on their difficulty: the chosen tasks should have good performance without masking but should be nontrivial. Thus, we choose "arithmetic: add_or_sub", "algebra: linear_1d", "calculus: differentiate", "comparison: sort" and "polynomials: collect". Note that the performance of our network may be lower than that reported by Saxton et al. [2019], since no transfer between tasks is possible, and we train for significantly fewer iterations due to limited computational resources.

We split the official easy, medium, and hard *train* sets to obtain new train and validation sets for each difficulty level. We randomly choose 10k samples for the new validation set; the rest is used as the new train set. We filter for repetitions, making sure that no sample appears twice. This way, we get a train and validation set for each difficulty level. We ignore the official test sets because of the missing distinction in difficulty. This treatment is needed because we want to be able to train the network on all difficulty levels but also the masks only on the easy split. Additionally, we want to evaluate its performance on the hard difficulty. In this way, we are able to determine whether specific weights are needed exclusively for performing the hard split. Note that the same rules govern the samples in all sets.

First, we train the network on all difficulty levels (easy, medium, and hard). Then we freeze its weights. Next, we train masks on the easy split and test on the hard split. If this results in a performance drop, then this indicates that the network requires a separate set of weights for different difficulty levels, which is undesirable. Nonetheless, we observe precisely this behavior (Fig. 3.6), which confirms once more that NNs tend to violate $\mathrm{P}_{\text{reuse}}$.

Interpreting the size of the drop is nontrivial due to how the easy, medium, and hard splits differ. The more difficult splits may include some samples from the easier splits, but never the other way around. This means that the hard test set's performance will be nonzero even if none of the hard samples are solved correctly. This behavior is inherent to the original dataset and can not be changed without regenerating it.

We use the Transformer [Vaswani et al., 2017] model from Saxton et al. [2019]. It has a $d_{model} = 256$, inner-layer dimensionality of $d_{ff} = 512$, $h = 4$ heads. Both the encoder and decoder have 3 layers. The word embeddings of the encoder and decoder are shared, and the output layer is tied to the word embedding. The network is always trained with teacher forcing. We use the Adam optimizer with a learning rate of 10^{-4}, $\epsilon = 10^{-9}, \beta_1 = 0.9, \beta_2 = 0.995$ and a gradient clipping of 1. We use 8 masks samples for each batch. We found that some tasks require a linear learning rate warmup for 5k iterations at the beginning of network training in order to converge. No warmup is used for training the masks. Individual tasks use different hyperparameters, listed in Tab. B.3. Batch sizes are chosen so that the experiments fit on a single GPU with 16Gb of VRAM (2 GPUs for "Poly. collect").

B.3.8 CNN Experiments on CIFAR10

Simple CNN

We use a learning rate of 10^{-3} and $\beta = 10^{-4}$. We train the network for 20k steps before freezing its weights and then use an additional 20k steps for training each of the masks, including the reference mask. See Tab. B.4 for details regarding the architecture.

Fig. B.11 shows the difference in the confusion matrix for all classes of CIFAR 10. The most surprising observation is that the decrease in performance for each of the classes is substantial, ranging from 40 to 60%. This shows the heavy reliance on class-exclusive features.

Analyzing confusion matrix differences yields interesting insights. "Airplane" is confused with"bird" and "ship", which is likely due to having a similar blue background. Classes "cat" and "dog" tend to be confused with each other—removing exclusive feature detectors for one improves the performance of the

What is the difference between 1801791.2422 and -0.7?
1801791.9422

Solve $-719*o + 3179*o + 135275 = -628*o - 777*o$ for o.
-35

What is the derivative of $30595*j**4 + 254*j**3 + 1559873$ wrt j?
$122380*j**3 + 762*j**2$

Sort $-3/5$, -1355.6, 703, 2, $-2/3$ in ascending order.
-1355.6, $-2/3$, $-3/5$, 2, 703

Collect the terms in $-26*v - 67 + 29*v + 12*v - 3 - 155$.
$15*v - 225$

Figure B.9: Examples from Mathematics Dataset. One sample for every task we use.

other. "Truck" and "car" are highly related, likely due to the similarities in terms of shape, such as having tires, and similar backgrounds, such as the road.

Simple CNN Without Dropout

The CNN architecture used for experiments in Sec. 3.4 uses dropout, as shown in Tab. B.4. A natural question to ask is how this affects the modularity of the resulting network. Fig. B.10 indicates that, as expected, removing dropout results in a few percent of performance loss. When comparing Figures B.11 and B.12 it can be seen that adding dropout causes a higher degradation in the class performance when the class-exclusive feature detectors are removed (roughly 30%-40% higher drop per class). This indicates that network with dropout depends more on class-specific modules, which is in line with findings presented by Filan et al. [2020].

ResNet-110

To demonstrate that these behaviors apply to more complex models, we train a ResNet-110 [He et al., 2016] model[1] which achieves competitive 93% validation accuracy following `https://github.com/bearpaw/pytorch-classification`. The network is built from non-bottleneck blocks ("BasicBlocks", Fig. 5, left by Vaswani et al. [2017]). It is trained with SGD using a weight decay of 10^{-4}, batch size of 128 and a starting learning rate of 0.1. The learning rate is divided by 10 at iterations $32\,000$ and $48\,000$ (corresponding roughly to epoch 81 and 122). The network is trained for $64\,000$ iterations (164 epochs). Data augmentation of random horizontal flipping and random crop (with padding 4 and output size of 32x32) is used. Masks are trained with Adam, batch size of 256, learning rate of 0.03, $\beta = 2 * 10^{-5}$, for $30\,000$ iterations each. Gradient clipping is not applied during the initial stage of training the weights, but the usual clipping to norm of 1.0 is applied when training the masks.

As Figures 3.7 and B.13 show, the performance drop per class is even more dramatic than in the simple CNN case, reaching almost 100%.

Inspecting the confusion matrix differences of different architectures as seen in Figures B.11, B.12 and B.13 highlight their similarity. This suggests that the interdependence between classes previously observed is mostly data driven an independent of the actual network architecture.

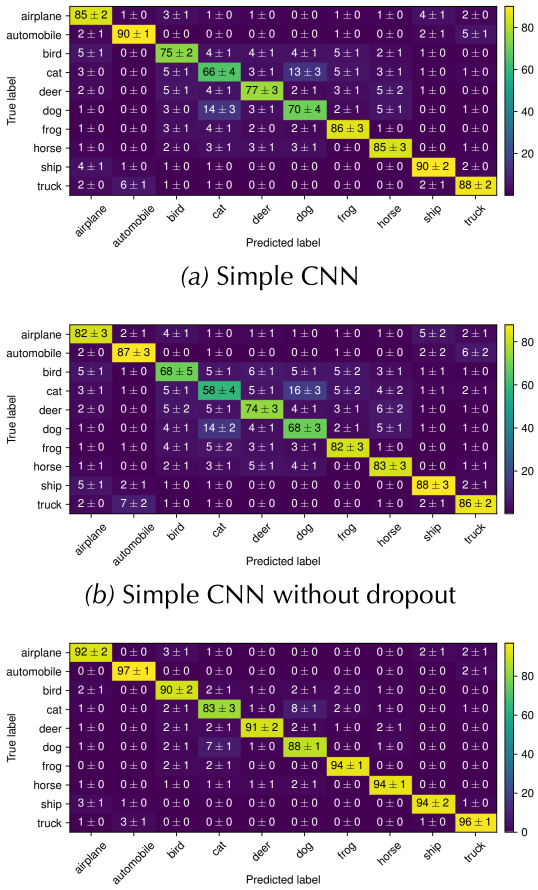
[1] We would like to note that ResNets are special case of Highway Networks [Srivastava et al., 2015b] with the gates fixed open.

Hyperparameter	Add or sub	Linear 1D	Differentiate	Sort	Poly. Collect
Batch size (net)	256	512	128	256	128
Batch size (mask)	256	400	128	256	256
Mask regularizer (β)	$2 * 10^{-5}$	10^{-6}	10^{-5}	$3 * 10^{-6}$	10^{-6}
Training iters (net)	30k	200k	40k	30k	200k
Training iters (masks)	30k	50k	40k	30k	50k
Learning rate (masks)	0.03	0.02	0.03	0.03	0.02
Warmup steps	-	5k	-	-	5k

Table B.3: Hyperparameters for different tasks on the Mathematics Dataset

Index	Operation	Inputs	Outputs	Kernel	Padding	Activation	Dropout
1	Conv	3	32	3x3	1	ReLU	-
2	Max pooling	32	32	2x2	0	-	-
3	Conv	32	64	3x3	1	ReLU	-
4	Max pooling	64	64	2x2	0	-	-
5	Conv	64	128	3x3	1	ReLU	0.25
6	Max pooling	128	128	2x2	0	-	-
7	Conv	128	256	3x3	1	ReLU	0.5
8	Spatial average	256	256	-	-	-	-
6	Feedforward	256	10	-	-	Softmax	-

Table B.4: Architecture of the simple CNN used for CIFAR 10 experiments



True label \ Predicted label	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	85 ± 2	1 ± 0	3 ± 1	1 ± 0	1 ± 0	0 ± 0	1 ± 0	1 ± 0	4 ± 1	2 ± 0
automobile	2 ± 1	90 ± 1	0 ± 0	0 ± 0	0 ± 0	0 ± 0	1 ± 0	0 ± 0	2 ± 1	5 ± 1
bird	5 ± 1	0 ± 0	75 ± 2	4 ± 1	4 ± 1	4 ± 1	5 ± 1	2 ± 1	1 ± 0	0 ± 0
cat	3 ± 0	0 ± 0	5 ± 1	66 ± 4	3 ± 1	13 ± 3	5 ± 1	3 ± 1	1 ± 0	1 ± 0
deer	2 ± 0	0 ± 0	5 ± 1	4 ± 1	77 ± 3	2 ± 1	3 ± 1	5 ± 2	1 ± 0	0 ± 0
dog	1 ± 0	0 ± 0	3 ± 0	14 ± 3	3 ± 1	70 ± 4	2 ± 1	5 ± 1	0 ± 0	1 ± 0
frog	1 ± 0	0 ± 0	3 ± 1	4 ± 1	2 ± 0	2 ± 1	86 ± 3	1 ± 0	0 ± 0	0 ± 0
horse	1 ± 0	0 ± 0	2 ± 0	3 ± 1	3 ± 1	3 ± 1	0 ± 0	85 ± 3	0 ± 0	1 ± 0
ship	4 ± 1	1 ± 0	1 ± 0	1 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	90 ± 2	2 ± 0
truck	2 ± 0	6 ± 1	1 ± 0	1 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	2 ± 1	88 ± 2

(a) Simple CNN

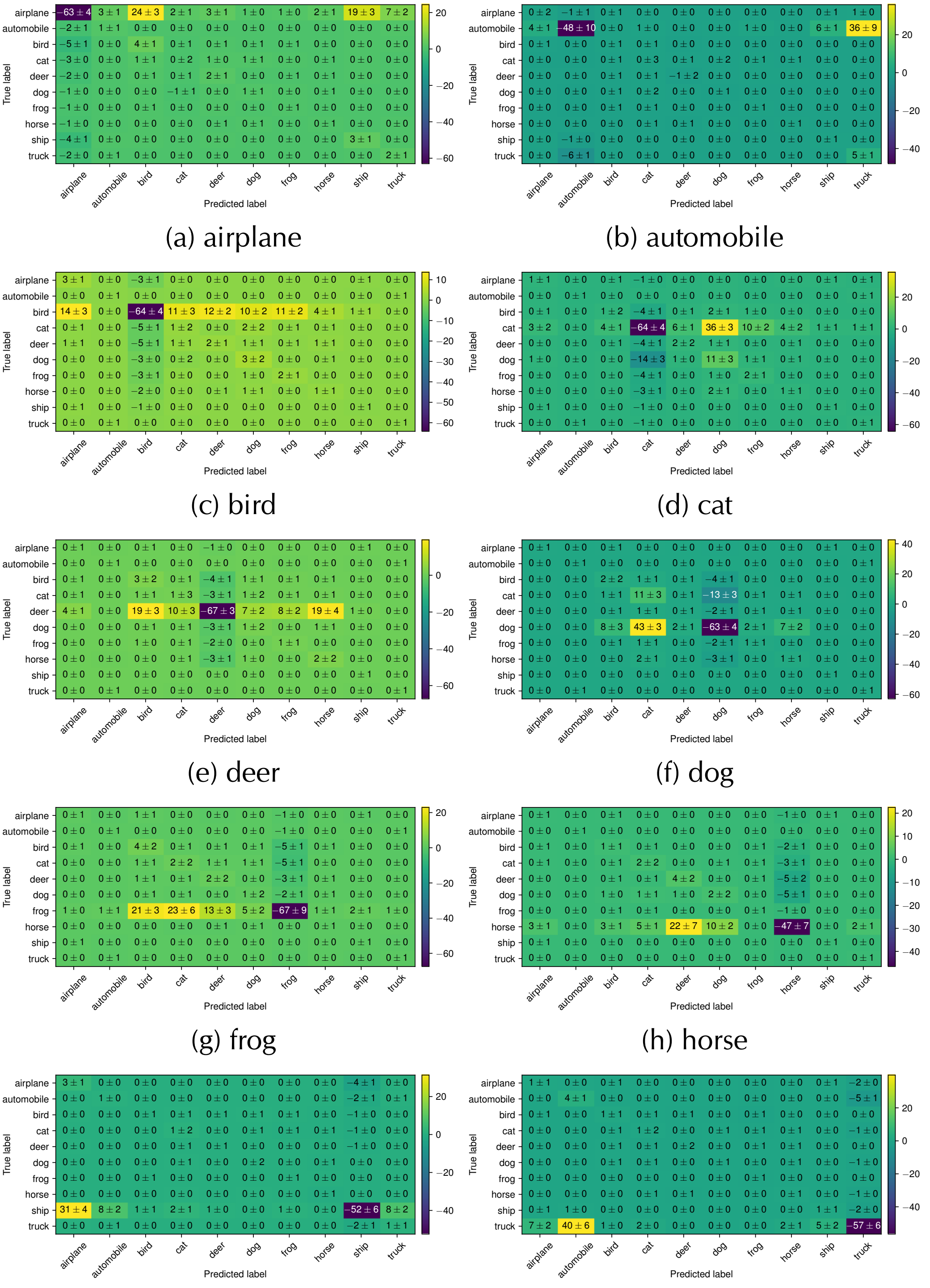
True label \ Predicted label	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	82 ± 3	2 ± 1	4 ± 1	1 ± 0	1 ± 1	1 ± 0	1 ± 0	1 ± 0	5 ± 2	2 ± 1
automobile	2 ± 0	87 ± 3	0 ± 0	1 ± 0	0 ± 0	0 ± 0	1 ± 0	0 ± 0	2 ± 2	6 ± 2
bird	5 ± 1	1 ± 0	68 ± 5	5 ± 1	6 ± 1	5 ± 1	5 ± 2	3 ± 1	1 ± 1	1 ± 0
cat	3 ± 1	1 ± 0	5 ± 1	58 ± 4	5 ± 1	16 ± 3	5 ± 2	4 ± 2	1 ± 1	2 ± 1
deer	2 ± 0	0 ± 0	5 ± 2	5 ± 1	74 ± 3	4 ± 1	3 ± 1	6 ± 2	1 ± 0	1 ± 0
dog	1 ± 0	0 ± 0	4 ± 1	14 ± 2	4 ± 1	68 ± 3	2 ± 1	5 ± 1	1 ± 0	1 ± 0
frog	1 ± 0	1 ± 0	4 ± 1	5 ± 2	3 ± 1	3 ± 1	82 ± 3	1 ± 0	0 ± 0	1 ± 0
horse	1 ± 1	0 ± 0	2 ± 1	3 ± 1	5 ± 1	4 ± 1	0 ± 0	83 ± 3	0 ± 0	1 ± 1
ship	5 ± 1	2 ± 1	1 ± 0	1 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	88 ± 3	2 ± 1
truck	2 ± 0	7 ± 2	1 ± 0	1 ± 0	0 ± 0	0 ± 0	0 ± 0	1 ± 0	2 ± 1	86 ± 2

(b) Simple CNN without dropout

True label \ Predicted label	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	92 ± 2	0 ± 0	3 ± 1	1 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	2 ± 1	2 ± 1
automobile	0 ± 0	97 ± 1	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	2 ± 1
bird	2 ± 1	0 ± 0	90 ± 2	2 ± 1	1 ± 0	2 ± 1	2 ± 0	1 ± 0	0 ± 0	0 ± 0
cat	1 ± 0	0 ± 0	2 ± 1	83 ± 3	1 ± 0	8 ± 1	2 ± 0	1 ± 0	0 ± 0	1 ± 0
deer	1 ± 0	0 ± 0	2 ± 1	2 ± 1	91 ± 2	2 ± 1	1 ± 0	2 ± 1	0 ± 0	0 ± 0
dog	1 ± 0	0 ± 0	2 ± 1	7 ± 1	1 ± 0	88 ± 1	0 ± 0	1 ± 0	0 ± 0	0 ± 0
frog	0 ± 0	0 ± 0	2 ± 1	2 ± 1	0 ± 0	0 ± 0	94 ± 1	0 ± 0	0 ± 0	0 ± 0
horse	1 ± 0	0 ± 0	1 ± 0	1 ± 1	1 ± 1	2 ± 1	0 ± 0	94 ± 1	0 ± 0	0 ± 0
ship	3 ± 1	1 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	94 ± 2	1 ± 0
truck	1 ± 0	3 ± 1	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	1 ± 0	96 ± 1

(c) ResNet-110

Figure B.10: Confusion matrix on CIFAR10 with masks trained on all classes. It can be seen that performance without dropout is a few percent lower, as expected. ResNet-110 has a significantly better performance in all classes.



(a) airplane

(b) automobile

(c) bird

(d) cat

(e) deer

(f) dog

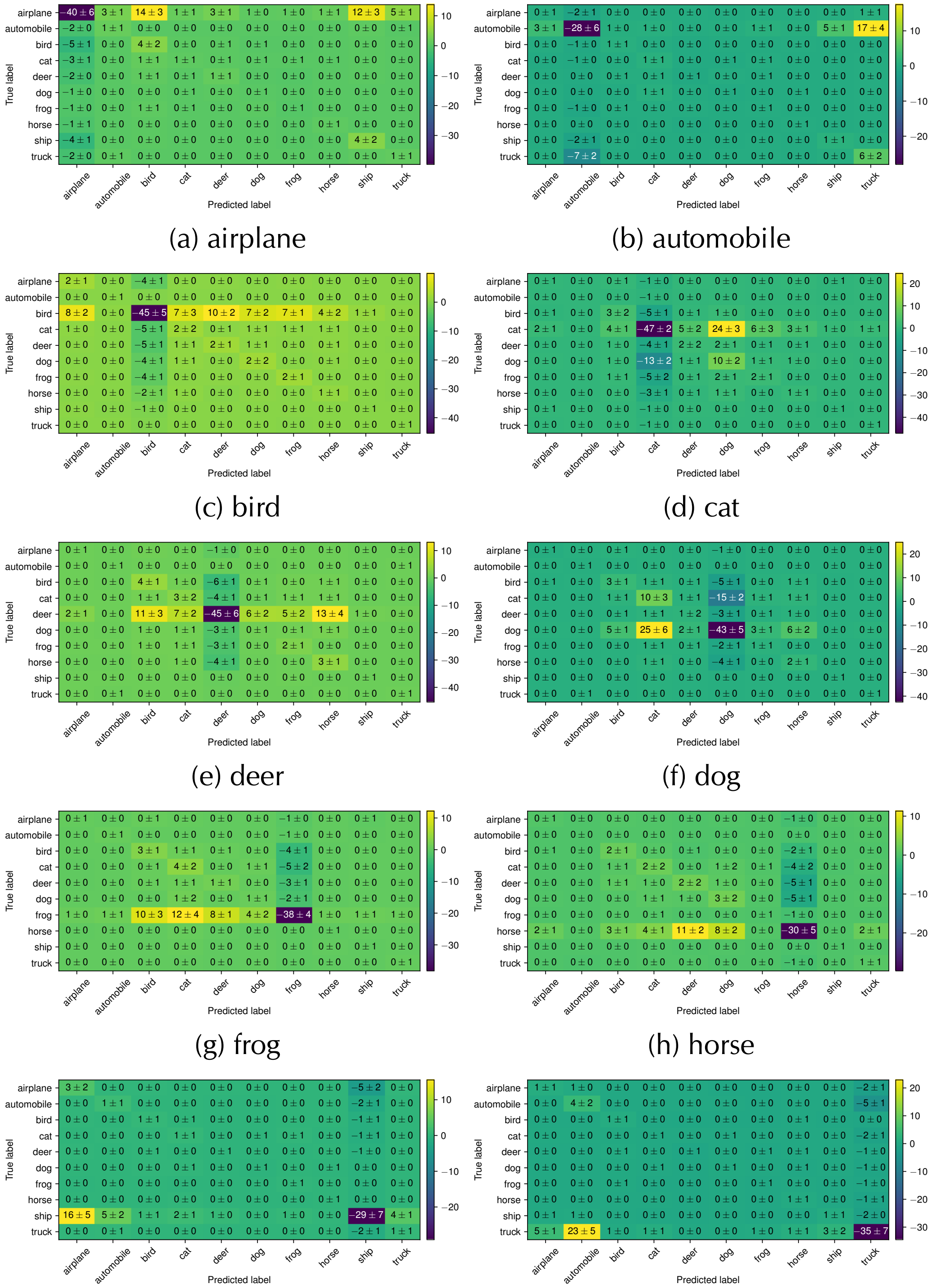
(g) frog

(h) horse

(i) ship

(j) truck

Figure B.11: Simple CNN: The change in confusion matrix for all CIFAR10 classes, when class indicated by the caption, is removed.



(a) airplane

(b) automobile

(c) bird

(d) cat

(e) deer

(f) dog

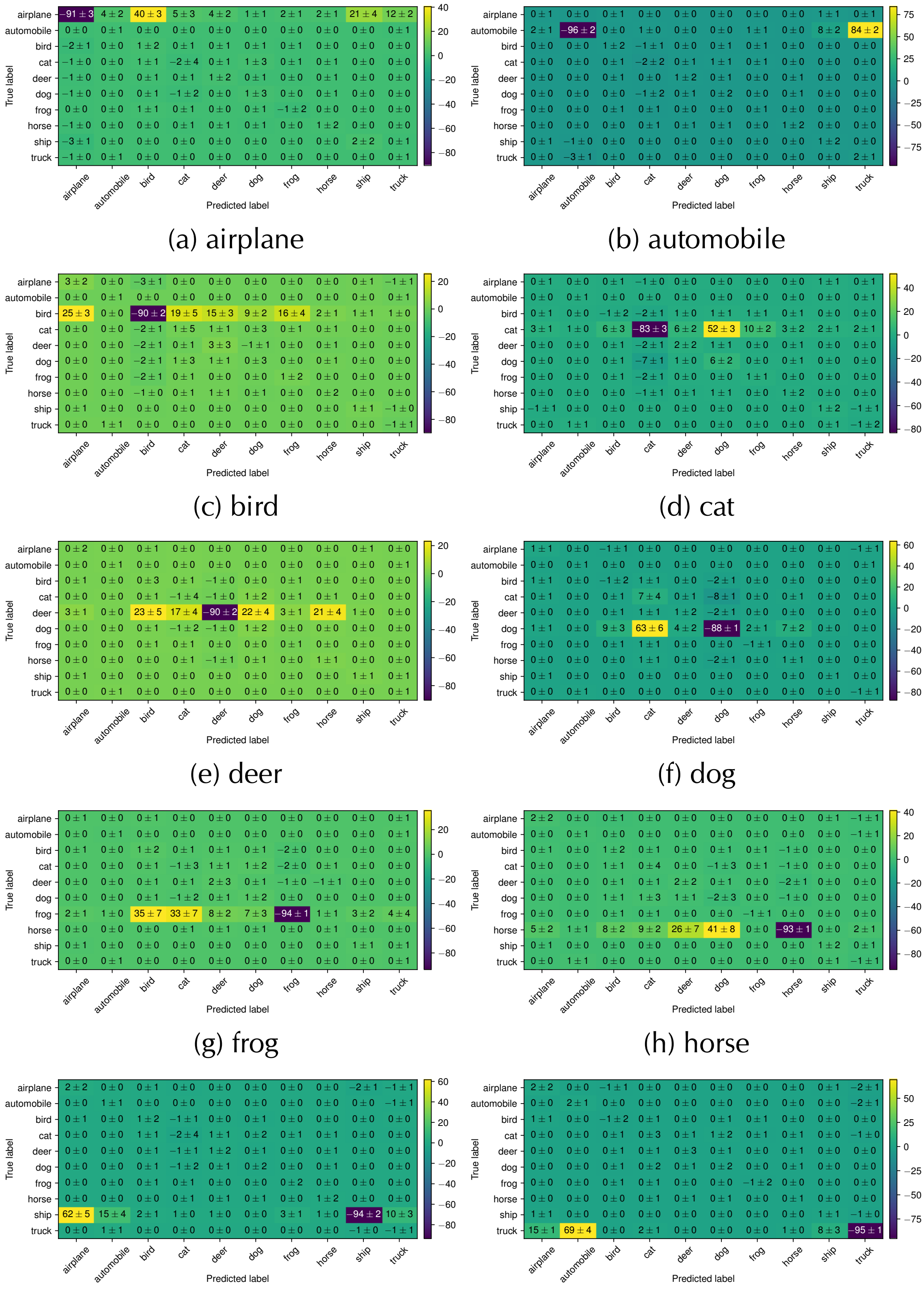
(g) frog

(h) horse

(i) ship

(j) truck

Figure B.12: Simple CNN without dropout: The change in confusion matrix for all CIFAR10 classes, when class indicated by the caption, is removed. The network has the same architecture as Tab. B.4, but without the dropout layers. The performance drop is reduced by roughly 30%-40% compared to the same architecture with dropout (Fig. B.11).



(a) airplane

(b) automobile

(c) bird

(d) cat

(e) deer

(f) dog

(g) frog

(h) horse

(i) ship

(j) truck

Figure B.13: ResNet-110: The change in confusion matrix for all CIFAR10 classes, when class indicated by the caption, is removed.

Appendix C

Additional Details on Improving the Systematic Generalization of Transformers

C.1 Evaluation Metrics

For all tasks, the accuracy is computed on the sequence level, i.e. all tokens in the sequence should be correct for the output to be counted as correct. For the losses, we always report the average token-wise cross-entropy loss.

C.2 Hyperparameters

For all of our models, we use an Adam optimizer with the default hyperparameters of PyTorch [Paszke et al., 2019]. We only change the learning rate. We use dropout with probability of 0.1 after each component of the transformer: both after the attention heads and linear transformations. We specify the dataset-specific hyperparameters in Tab. C.1. For all universal transformer experiments, we use both the "No scaling" and the "Positional Embedding Downscaling" methods. For the standard transformers with absolute positional embedding we test different scaling variants on different datasets shown in Tab. C.3. When multiple scaling methods are available, we choose the best performing ones when reporting results in Tab. 4.4. We always use the same number of layers for both encoder and decoder. The embedding and the final softmax weights of the decoder are always shared (tied embeddings). All of our transformers are post-layenorm [Xiong et al., 2020], matching the configuration of Dai et al.

[2019].

The number of parameters for different models and the corresponding to representative execution time is shown in Tab. C.2.

C.3 Relative Positional Embedding

We use the relative positional embedding variant of self-attention from Dai et al. [2019]. Here, we use a decomposed attention matrix of the following form:

$$\boldsymbol{A}_{i,j}^{\text{rel}} = \underbrace{\boldsymbol{H}_i^\top \boldsymbol{W}_q^\top \boldsymbol{W}_{k,E} \boldsymbol{H}_j}_{(a)} + \underbrace{\boldsymbol{H}_i^\top \boldsymbol{W}_q^\top \boldsymbol{W}_{k,P} \boldsymbol{P}_{i-j}}_{(b)} + \underbrace{\boldsymbol{u}^\top \boldsymbol{W}_{k,E} \boldsymbol{H}_j}_{(c)} + \underbrace{\boldsymbol{v}^\top \boldsymbol{W}_{k,P} \boldsymbol{P}_{i-j}}_{(d)}$$

where $\boldsymbol{H}_i$ is the hidden state of the i^{th} column of the transformer, $\boldsymbol{P}_i$ is an embedding for position (or in this case distance) i. Matrix $\boldsymbol{W}_q$ maps the states to queries, $\boldsymbol{W}_{k,E}$ maps states to keys, while $\boldsymbol{W}_{k,P}$ maps positional embedding to keys. $\boldsymbol{u}$ and $\boldsymbol{v}$ are learned vectors. Component (a) corresponds to content-based addressing, (b) to content based relative positional addressing, (c) represents a global content bias, while (d) represents a global position bias.

We use sinusoidal positional embedding $\boldsymbol{P}_i \in \mathbb{R}^{d_{\text{model}}}$. The relative position, i, can be both positive and negative. Inspired by Vaswani et al. [2017], we define $\boldsymbol{P}_{i,j}$ as:

$$\boldsymbol{P}_{i,j} = \begin{cases} \sin(i/10000^{2j/d_{\text{model}}}), & \text{if } j = 2k \\ \cos(i/10000^{2j/d_{\text{model}}}) & \text{if } j = 2k+1 \end{cases} \tag{C.1}$$

Before applying softmax, $\boldsymbol{A}_{i,j}^{\text{rel}}$ is scaled by $\frac{1}{\sqrt{d_{\text{model}}}}$, as in the original model by Vaswani et al. [2017].

We never combine absolute with relative positional embedding. In case of a relative positional variant of any transformer model, we do not add absolute positional encoding to the word embeddigs. We use relative positional attention in every layer, except at the interface between encoder and decoder, where we use the standard formulation from Vaswani et al. [2017], without adding any positional embedding.

C.4 Embedding Scaling

In this section, we provide full descriptions of embedding scaling strategies that we investigated. In the following, w_i denotes the word index at input position

i, $\boldsymbol{E}_w \in \mathbb{R}^{d_{\text{model}}}$ denotes learned word embedding for word index w. Positional embedding for position i is defined as in Eq. C.1.

Token Embedding Upscaling. Vaswani et al. [2017] combine the input word and positional embeddings for each position i as $\boldsymbol{H}_i = \sqrt{d_{\text{model}}}\boldsymbol{E}_{w_i} + \boldsymbol{P}_i$. Although in the original paper, the initialization of $\boldsymbol{E}$ is not discussed, most implementations use Glorot initialization [Glorot and Bengio, 2010], which in this case means that each component of $\boldsymbol{E}$ is drawn from $\mathcal{U}(-\sqrt{\frac{6}{d_{\text{model}}+N_{\text{words}}}}, \sqrt{\frac{6}{d_{\text{model}}+N_{\text{words}}}})$ where $\mathcal{U}(a, b)$ represents the uniform distribution in range $[a, b]$.

No scaling. This corresponds to how PyTorch initializes embedding layers by default: each element of $\boldsymbol{E}$ is drawn from $\mathcal{N}(0, 1)$. $\mathcal{N}(\mu, \sigma)$ is the normal distribution with mean μ and standard deviation of σ. The word embeddings are combined with the positional embeddings without any scaling: $\boldsymbol{H}_i = \boldsymbol{E}_{w_i} + \boldsymbol{P}_i$

Position Embedding Downscaling. We propose to use Kaiming initialization [He et al., 2015] for the word embeddings: each element of $\boldsymbol{E} \sim \mathcal{N}(0, \frac{1}{\sqrt{d_{\text{model}}}})$. Instead of scaling up the word embeddings, the positional embeddings are scaled down: $\boldsymbol{H}_i = \boldsymbol{E}_{w_i} + \frac{1}{\sqrt{d_{\text{model}}}}\boldsymbol{P}_i$

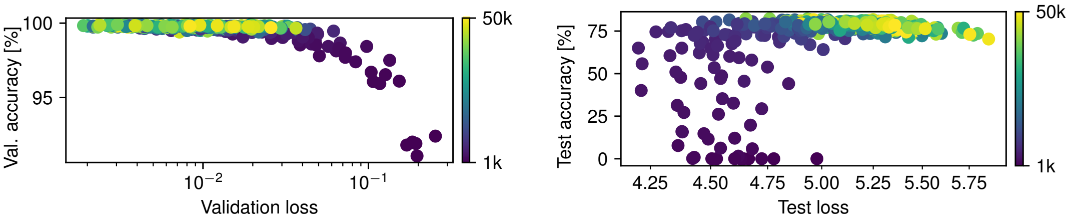
C.5 Analyzing the Positively Correlated Loss and Accuracy

In Sec. 4.2.2, we reported that on the generalization splits of some datasets both the accuracy and the loss grow together during training. Here, we further analyze this behavior in Fig. C.2 (see the caption).

C.6 Additional Results

Fig. C.1 shows that both the test loss and accuracy grows on COGS dataset during training. Additionally, it shows the expected, IID behavior on the same dataset for contrast.

Fig. C.3 shows the relative change in convergence speed when using relative positional embeddings.



(a) COGS: IID Validation set

(b) COGS: Generalization test set

Figure C.1: Relationship between the loss and accuracy on (a) IID validation set and (b) the generalization test set on COGS (it comes without a validation set for the generalization splits). Standard transformers are used. The color shows the training step. Five runs are shown. The loss is shown on a logarithmic scale. On the IID validation set (a), the accuracy increases when the loss decreases, as expected. On the contrary, in the generalization split (b), high accuracy corresponds to higher loss. For generalization validation loss versus generalization accuracy on CFQ MCD 1, see Fig. 4.2. For an analysis of the underlying reason, see Fig. C.2.

	d_{model}	d_{FF}	n_{head}	n_{layers}	batch size	learning rate	warmup	scheduler
SCAN	128	256	8	3	256	10^{-3}	-	-
CFQ - Non-uni.	128	256	16	2	4096	0.9*	4000	Noam
CFQ - Uni.	256	512	4	6	2048	2.24*	8000	Noam
PCFG	512	2048	8	6	64	10^{-4}	-	-
COGS	512	512	4	2	128	10^{-4}	-	-
COGS Noam	512	512	4	2	128	2	4000	Noam
Mathematics	512	2048	8	6	256	10^{-4}	-	-

Table C.1: Hyperparameters used for different tasks. We denote the feedforward size as d_{FF}. For the learning rate of CFQ (denoted by *), the learning rate seemingly differs from Keysers et al. [2020]. In fact, although Keysers et al. [2020] use Noam learning rate scheduling, scaling by $\frac{1}{\sqrt{d_{\text{model}}}}$ is not used, so we had to compensate for this to make them functionally equivalent.

Dataset	Model	#params	Duration	GPU type
SCAN	Standard	992k	1:30	Titan X Maxwell
	Universal	333k	1:15	
	Relative Pos.	1.1M	1:45	
	Universal, Relative Pos.	366k	1:30	
CFQ MCD 2	Standard	685k	10:00	Tesla V100
	Universal	1.4M	12:00	
	Relative Pos.	751k	14:15	
	Universal, Relative Pos.	1.5M	14:00	
PCFG Systematicity	Standard	44.7M	20:30	Tesla V100
	Universal	7.9M	17:00	
	Relative Pos.	47.8M	21:30	
	Universal, Relative Pos.	8.4M	21:30	
COGS	Standard	9.3M	17:30	Tesla V100
	Universal	5.1M	17:15	
	Relative Pos.	10.3M	21:00	
	Universal, Relative Pos.	5.6M	20:00	
Math: add_or_sub	Standard	4.4M	8:00	Tesla P100
	Universal	7.4M	7:30	
	Relative Pos.	4.7M	8:30	
	Universal, Relative Pos.	7.9M	8:00	

Table C.2: Model sizes and execution times. One representative split is shown per dataset. Other splits have the same number of parameters, and their execution time is in the same order of magnitude.

	TEU	No scaling	PED
SCAN		✓	✓
CFQ MCD		✓	✓
CFQ Length	✓	✓	✓
PCFG Productivity	✓	✓	✓
PCFG Systematicity	✓	✓	✓
COGS	✓	✓	✓
Mathematics		✓	✓

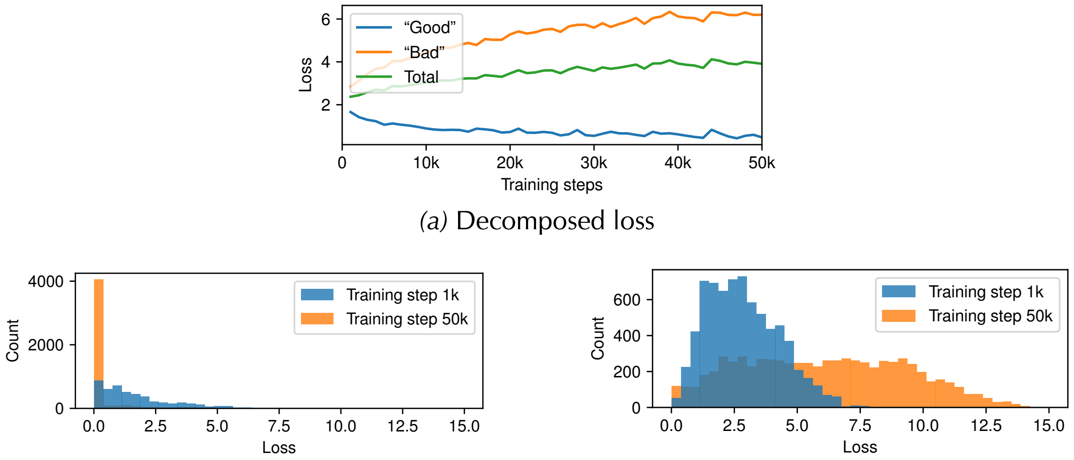
Table C.3: Scaling types used for standard transformers with absolute positional embedding on different datasets. TEU denotes Token Embedding Upscaling, PED denotes Position Embedding Downscaling.

	Init	Trafo	Uni. Trafo	Rel. Trafo	Rel. Uni. T.	Reported
SCAN (cutoff=26)	PED	**0.30** ± **0.02**	**0.21** ± **0.01**	-	-	0.00[1]
	NoS	0.15 ± 0.07	0.14 ± 0.05	**0.72** ± **0.21**	**1.00** ± **0.00**	
CFQ Output len.	PED	0.56 ± 0.02	0.60 ± 0.34	-	-	∼ 0.66[2]
	TEU	**0.57** ± **0.00**	0.74 ± 0.02 †	-	-	
	NoS	0.53 ± 0.04	**0.77** ± **0.02**	**0.64** ± **0.06**	**0.81** ± **0.01**	
CFQ MCD 1	PED	0.36 ± 0.02	0.37 ± 0.05	-	-	0.37 ± 0.02[3]
	NoS	**0.40** ± **0.01**	**0.39** ± **0.03**	**0.39** ± **0.01**	**0.39** ± **0.04**	
CFQ MCD 2	PED	0.08 ± 0.01	**0.09** ± **0.01**	-	-	0.08 ± 0.02[3]
	NoS	**0.10** ± **0.01**	**0.09** ± **0.02**	**0.09** ± **0.01**	**0.10** ± **0.02**	
CFQ MCD 3	PED	0.10 ± 0.00	**0.11** ± **0.00**	-	-	0.11 ± 0.00[3]
	NoS	**0.11** ± **0.00**	**0.11** ± **0.01**	**0.11** ± **0.01**	**0.11** ± **0.03**	
CFQ MCD mean	PED	0.18 ± 0.13	0.19 ± 0.14	-	-	0.19 ± 0.01[2]
	NoS	**0.20** ± **0.14**	**0.20** ± **0.14**	**0.20** ± **0.14**	**0.20** ± **0.14**	

Table C.4: Test accuracy of different transformer (Trafo) variants and different initializations on the considered datasets. This is a more detailed version of Tab. 4.4. We shorten the "No scaling" variant as "NoS". The last column shows previously reported accuracies. References: [1] Newman et al. [2020], [2] Keysers et al. [2020], [3] `https://github.com/google-research/google-research/tree/master/cfq`. Results marked with $*$ cannot be directly compared because of different training setups. $\sim$ denotes imprecise numbers read from charts in prior works. For the configuration marked by †, the results are obtained by running 8 seeds from which 3 crashed, resulting in 5 useful runs reported below. Crashed runs suddenly drop their accuracy to 0, which never recovers. The reason for the crashing is the overly big learning rate (2.24, from the baseline). We run another 10 seeds with learning rate of 2.0, obtaining similar final accuracy of 0.75 ± 0.02, but without any crashed runs. Part 1/2. For part 2/2, see Tab. C.5.

	Init	Trafo	Uni. Trafo	Rel. Trafo	Rel. Uni. T.	Reported
			...			
PCFG Prod. split	PED	**0.65** ± **0.03**	**0.78** ± **0.01**	-	-	0.50 ± 0.02[4]
	TEU	0.47 ± 0.27	**0.78** ± **0.01**	-	-	
	NoS	0.63 ± 0.02	0.76 ± 0.01	-	**0.85** ± **0.01**	
PCFG Sys. split	PED	**0.87** ± **0.01**	**0.93** ± **0.01**	-	-	0.72 ± 0.00[4]
	TEU	0.75 ± 0.08	0.92 ± 0.01	-	-	
	NoS	0.86 ± 0.02	0.92 ± 0.00	**0.89** ± **0.02**	**0.96** ± **0.01**	
COGS	PED	**0.80** ± **0.00**	0.77 ± 0.02	-	-	0.35 ± 0.06[5]
	TEU	0.78 ± 0.03	**0.78** ± **0.03**	-	-	
	NoS	0.62 ± 0.06	0.51 ± 0.07	**0.81** ± **0.01**	**0.77** ± **0.01**	
Math: add_or_sub	PED	0.80 ± 0.01	0.92 ± 0.02	-	-	~ 0.91[6]*
	NoS	**0.89** ± **0.01**	**0.94** ± **0.01**	**0.91** ± **0.03**	**0.97** ± **0.01**	
Math: place_value	PED	0.00 ± 0.00	**0.20** ± **0.02**	-	-	~ 0.69[6]*
	NoS	**0.12** ± **0.07**	0.12 ± 0.01	-	**0.75** ± **0.10**	

Table C.5: Test accuracy of different transformer (Trafo) variants and different initializations on the considered datasets. This is a more detailed version of Tab. 4.4. We shorten the "No scaling" variant as "NoS". The last column shows previously reported accuracies. References: [4] Hupkes et al. [2020], [5] Kim and Linzen [2020], [6] Saxton et al. [2019]. Results marked with $*$ cannot be directly compared because of different training setups. $\sim$ denotes imprecise numbers read from charts in prior works. For the configuration marked by †, the results are obtained by running 8 seeds from which 3 crashed, resulting in 5 useful runs reported below. Crashed runs suddenly drop their accuracy to 0, which never recovers. The reason for the crashing is the overly big learning rate (2.24, from the baseline). We run another 10 seeds with learning rate of 2.0, obtaining similar final accuracy of 0.75 ± 0.02, but without any crashed runs. Part 2/2. For part 1/2, see Tab. C.4.



(a) Decomposed loss

(b) Histogram of "good" loss (first and last measurement)

(c) Histogram of "bad" loss (first and last measurement)

Figure C.2: Analysis of the growing test loss on the systematically different test set on CFQ MCD 1 split. We measure the loss individually for each sample in the test set. We categorize samples as "good" if the network output on the corresponding input matched the target exactly any point during the training, and as "bad" otherwise. (a) The total loss (increasing) can be decomposed to the loss of the "good" samples (decreasing), and the loss of the "bad" samples (increasing). (b, c) The histogram of the loss for the "good" and "bad" samples at the beginning and end of the training. The loss of the "good" samples concentrates near zero, while the "bad" samples spread out and the corresponding loss can be very high. The net effect is a growing total loss.

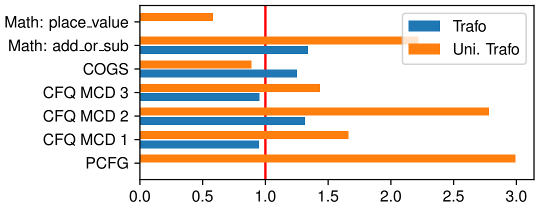


Figure C.3: Relative change in convergence speed by using relative positional embeddings instead of absolute. Convergence speed is measured as the mean number of steps needed to achieve 80% of the final performance of the model. Relative variants usually converge faster. Universal transformers benefit more than the non-universal ones. The non-universal variants are not shown for PCFG and "Math: place_value", because the relative variants do not converge (see Sec. 4.2.1).

	Variant	Transformer	Rel. Trafo	Uni. Trafo	Rel. Uni. Trafo
CFQ MCD 1	Big	0.40 ± 0.01	0.39 ± 0.02	0.41 ± 0.03	0.42 ± 0.02
	Small	0.26 ± 0.02	0.32 ± 0.01	0.28 ± 0.00	0.36 ± 0.01
	Ratio	0.65	0.80	0.68	**0.85**
CFQ MCD 2	Big	0.10 ± 0.01	0.09 ± 0.01	0.09 ± 0.00	0.09 ± 0.02
	Small	0.05 ± 0.01	0.07 ± 0.01	0.04 ± 0.01	0.10 ± 0.01
	Ratio	0.51	0.76	0.50	**1.05**
CFQ MCD 3	Big	0.11 ± 0.00	0.11 ± 0.01	0.11 ± 0.01	0.12 ± 0.02
	Small	0.09 ± 0.00	0.09 ± 0.00	0.09 ± 0.01	0.11 ± 0.01
	Ratio	0.80	0.85	0.85	**0.98**
CFQ Out. len.	Big	0.57 ± 0.02	0.64 ± 0.04	0.76 ± 0.03	0.81 ± 0.02
	Small	0.41 ± 0.03	0.51 ± 0.02	0.55 ± 0.02	0.70 ± 0.03
	Ratio	0.72	0.80	0.73	**0.87**

Table C.6: Accuracy of different transformer variants on CFQ. "Big" variant has a batch size of 4096, and is trained with Noam scheduler (learning rate 0.9). "Small" variant has a batch size of 512 and a fixed learning rate of 10^{-4}. The ratio of accuracies of "small" and "big" variants are also shown in the "Ratio" column, indicating the relative performance drop caused by decreasing the batch size. Relative variants experience less accuracy drop.

Dataset	#train	#IID val.	#gen. test	#gen. val.	Voc. size	Train len.	Test len.
Scan (cutoff = 26)	16458	1828	2624	-	19	9/26	9/48
CFQ MCD 1	95743	-	11968	11968	181	29/95	30/103
CFQ MCD 2	95743	-	11968	11968	181	29/107	30/91
CFQ MCD 3	95743	-	11968	11968	181	29/107	30/103
CFQ Output Len.	100654	-	9512	9512	181	29/77	29/107
PCFG Prod.	81010	-	11333	-	535	53/200	71/736
PCFG Sys.	82168	-	10175	-	535	71/736	71/496
COGS	24155	3000	21000	-	871	22/153	61/480
Math: add_or_sub	1969029	10000	10000	-	69	60/19	62/23
Math: place_value	1492268	9988	10000	-	69	50/1	52/1

Table C.7: Dataset statistics. "#" denotes number of samples. Vocabulary size shows the union of input and output vocabularies. Train and test length denotes the maximum input/output length in the train and test set, respectively.

Appendix D

Additional Details for Achieving Length Generalization with Transformers

D.1 Ablations

Readout from the first instead of the last column. In our experiments with the transformer models, the last column was used for the readout of the result. Under this configuration, the readout position depends on the length of the sequence, which might increase the difficulty of the problem, in particular for the models using absolute positional embeddings. Tab. D.1 shows the corresponding ablation study. We observe that this choice has only a marginal impact on the model performance. As a side note, we also tried the variant in which an additional cross-attention layer is used for the readout. Again, the generalization performance was not better. In fact, these results are not surprising since none of these changes fundamentally addresses the problem of length generalization.

Does Adaptive Computation Time (ACT) help? In this work, we determined the number of layers/steps to be used in the model based on heuristics (see Appendix D.3.1). We could also consider using Adaptive Computation Time (ACT) to dynamically determine the number of steps. Furthermore, ACT introduces a form of gating that creates shortcuts in the credit assignment path between the output and a result of an intermediate layer. This "copying" mechanism resulting from the ACT (i.e. stop computation at a certain time and copy the result to the output) is fundamentally different from our copy gate (Sec. 5.1.1). Our copy gate allows transformer columns to keep the input unchanged until it is their turn to

		IID		Longer	
Model	Readout	Forward	Backward	Forward	Backward
Transformer	First	1.00 ± 0.00	0.82 ± 0.39	0.12 ± 0.01	0.13 ± 0.01
	Last	1.00 ± 0.00	0.82 ± 0.39	0.13 ± 0.01	0.12 ± 0.01
+ rel	First	1.00 ± 0.00	1.00 ± 0.00	0.12 ± 0.01	0.22 ± 0.05
	Last	1.00 ± 0.00	1.00 ± 0.00	0.23 ± 0.05	0.13 ± 0.01
+ rel + gate	First	1.00 ± 0.00	1.00 ± 0.00	0.17 ± 0.02	1.00 ± 0.00
	Last	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.01	0.19 ± 0.04

Table D.1: Accuracy on **compositional table lookup** dataset with the results read from the first or last column (**Readout**).

be processed (a crucial property to implement control flow like behavior). This behavior cannot be simulated by the ACT. Here we provide some experimental results on models with ACT which confirm that the proposed copy gate is a crucial component for generalization which cannot be replaced by ACT.

We note that there are various versions of ACT in the literature, e.g., the variant used by Dehghani et al. [2019] in universal transformers is different from the one used by Graves [2016]. Here, we focus on two variants: one in which we directly apply Graves [2016] to transformers, and another one used by Dehghani et al. [2019]. We start with the description of the former.

An extra sigmoidal unit $\hat{p}^{(i,t)}$ is computed for each column i in each timestep t as:

$$\hat{p}^{(i,t)} = \sigma(\boldsymbol{W}_{\mathsf{H}}\boldsymbol{h}^{(i,t)} + b_{\mathsf{H}}) \tag{D.1}$$

where $\boldsymbol{W}_{\mathsf{H}} \in \mathbb{R}^{1\times d}$ and $\boldsymbol{b}_{\mathsf{H}} \in \mathbb{R}$ are trainable parameters. By comparing the cumulative sum of $\hat{p}^{(i,t)}$ over time steps to a certain threshold value $(1-\epsilon)$ with a hyper-parameter ϵ $(0.01$ in our experiment), we determine the termination step T^i for column i as:

$$T^i = \min\{T_{\mathsf{max}}, \min\{t' : \sum_{t=1}^{t'} \hat{p}^{(i,t)} \geq 1-\epsilon\}\} \tag{D.2}$$

where T_{max} is the pre-defined maximum number of steps.

The corresponding *halting probability* $p^{(i,t)}$ is then computed as:

$$p^{(i,t)} = \begin{cases} \hat{p}^{(i,t)} & \text{if } t < T^i \\ R^i & \text{if } t = T^i \end{cases} \tag{D.3}$$

$$R^i = 1 - \sum_{t=1}^{T^i - 1} \hat{p}^{(i,t)} \tag{D.4}$$

which is used to compute the final output of column i as:

$$\boldsymbol{o}^i = \sum_{t=1}^{T^i} p^{(i,t)} \boldsymbol{h}^{(i,t)} \tag{D.5}$$

In the variant of Dehghani et al. [2019], a different equation is used in lieu of Eq. D.5 above, and the calculation of the reminder term R^i in Eq. D.4 above is not properly handled in case where Eq. D.2 terminates due to the first condition on T_{max}. For further details, we refer the reader to Listings 1 and 2 by Dehghani et al. [2019] and/or our public code.

One subtlety introduced by Dehghani et al. [2019] which we note here is that the calculation of the final output $\boldsymbol{o}^i$ of column i effectively "halts" after T^i (since $\boldsymbol{o}^i$ only depends on $\boldsymbol{h}^{(i,t)}$ for $0 < t < T^i$), but column i itself still continues to transform the hidden states $\boldsymbol{h}^{(i,t)}$ for steps $t > T^i$ until all columns reach the termination step, and its updated states can be attended/read by another column j which has not halted yet (i.e. $T^j > T^i$). In this sense, computation is never stopped independently for each column. The mechanism described above instead finds the *readout* steps for each column (as used in Eq. D.5). We follow this decision in our implementation of both variants.

In addition, a new regularizer term $L_{\text{ACT}} = \alpha \frac{1}{N} \sum_{i=1}^{N} R^i$ is added to the loss function, where N is the length of the input sequence. This makes the network prefer short computations. We ran a hyperparameter search for α from the following values: 0.001, 0.003, 0.01, 0.03, 0.1. We found that $\alpha = 0.03$ works the best.

We conducted experiments on the compositional table lookup task. We first noticed that ACT helps training our baseline transformer models with a maximum step of 14 layers, which was not possible without ACT (our baseline transformer had only 11 layers for this reason; see Tab. D.3). The shortcut in the credit assignment path introduced by ACT certainly helps training of this 14-layer model. As we noticed that the models with ACT learn slower than those with gating, we increased the number of training steps to 60k steps, which is twice as many as 30k used for the models without ACT. Tab. D.2 shows the results. We observe that, interestingly, ACT enables generalization for longer lengths in the forward direction of the transformer with relative positional encoding and the one with geometric attention. However, we were not able to find any configuration that generalizes in the backward case. This demonstrates that the copy gate is effectively a crucial component for generalization that cannot be replaced by ACT. Furthermore, the convergence of models with ACT is significantly slower than those of models with our gating, and they are more unstable and very sensitive to the value of α on the regularization term, even in the successful forward case. Overall, the only benefit of ACT is therefore the adaptive depth, as illustrated in Fig. D.1, which is orthogonal to our study.

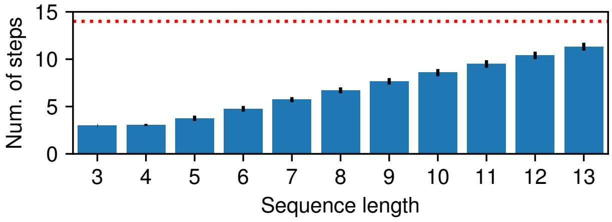


Figure D.1: Average number of steps/layers for different sequence lengths on the compositional table lookup task for the transformer with relative positional encodings and the ACT variant described in Appendix D.1. The red line shows $T_{\text{max}} = 14$. Note that the sequence length shown here includes the begin and end tokens. Thus, the sequence length of 4 corresponds to one function application (3 for the identity function i.e. no function is applied).

D.2 Details of Attention with Combined Absolute/Relative Positional Encoding

The use of copy gates enables transformers to generalize to longer lengths in the forward presentation order of the CTL task (Sec. 5.2.1), but that alone was not enough to make the model generalize in the backward order variant of the task. Examining the attention maps reveals that the model uses position-based attention to read out the result instead of content-based attention. In the backward presentation order, the last column of the transformer should focus on the second column, whose relative position changes dynamically with the length of the sequence. To verify that this is indeed the root cause of the network failure, we added an option to choose between absolute and relative positional encodings to the attention head.

In what follows, we describe the operation within a single layer/step. This allows us to omit the layer/step-index t for better readability, and thus denote the state of column/position i as $\boldsymbol{h}_i$ instead of $\boldsymbol{h}^{(i,t)}$. We use the relative positional embedding variant of self-attention by Dai et al. [2019]. Our attention matrix with the gated absolute/relative positional encodings can be decomposed as follows:

$$r_i = \sigma(\boldsymbol{h}_i \boldsymbol{W}_{ar} + b_{ar}) \tag{D.6}$$

$$\hat{\boldsymbol{A}}_{i,j} = \underbrace{\boldsymbol{h}_i^\top \boldsymbol{W}_q^\top \boldsymbol{W}_{k,E} \boldsymbol{h}_j}_{(a)} + \underbrace{\boldsymbol{b}_{q,E}^\top \boldsymbol{W}_{k,E} \boldsymbol{h}_j}_{(c)} + \big(\underbrace{\boldsymbol{h}_i^\top \boldsymbol{W}_q^\top \boldsymbol{W}_{k,P}}_{(b)} \tag{D.7}$$
$$+ \underbrace{\boldsymbol{b}_{q,P}^\top \boldsymbol{W}_{k,P}}_{(d)} \big) \underbrace{\big(\boldsymbol{p}_{i-j} r_i + \boldsymbol{p}_j (1 - r_i) \big)}_{(e)}$$

where the matrix $\boldsymbol{W}_q \in \mathbb{R}^{d_{\text{head}} \times d}$ maps the states to queries, $\boldsymbol{W}_{k,E} \in \mathbb{R}^{d_{\text{head}} \times d}$ maps states to keys, while $\boldsymbol{W}_{k,P} \in \mathbb{R}^{d_{\text{head}} \times d}$ maps positional embeddings to keys. d_{head} is the size of the key, query and value vectors for each head, set as $d_{\text{head}} = \frac{d}{n_{\text{head}}}$. $\boldsymbol{b}_{q,E}, \boldsymbol{b}_{q,P} \in \mathbb{R}^{d_{\text{head}}}$ are learned vectors. $\boldsymbol{p}_i \in \mathbb{R}^d$ is the standard sinusoidal embedding for position i [Vaswani et al., 2017]. Softmax is applied to the second dimension of $\hat{\boldsymbol{A}}$ to obtain the final attention scores, $\boldsymbol{A}$. Component (a) corresponds to content-based addressing, (b, e) to content-based positional addressing, (c) represents a global content bias, while (d, e) represent a global position bias.

We introduce the term (e) for the positional embedding which can switch between absolute and relative positional encodings using the scalar gate r_i (Eq. D.6; parameterized by $\boldsymbol{W}_{ar} \in \mathbb{R}^{d \times 1}$ and $b_{ar} \in \mathbb{R}$), which is the function of the state at the target position i.

As can be seen from Tab. 5.1, Tab. 5.3 and Tab. 5.4, this helps in specific settings, but unlike geometric attention, it fails in others. This is as expected since more complex access plates can still overfit to either the absolute or relative position component.

D.3 Implementation Details

A PyTorch implementation of our models together with the experimental setup is available under `https://github.com/robertcsordas/ndr`. The performance of all models is reported as mean and standard deviations over 5 different seeds.

D.3.1 Choosing the number of layers

In Sec. 5.1, we hypothesized that one of the conditions for our model to generalize is to be "sufficiently" deep such that elementary operations are learned in separate layers which would then become composable. In practice, a "sufficient" depth can be determined by the basic units of compositions implicitly defined by the dataset. The depth of the model must be at least as deep as the deepest path in the computation graph defined by these basic operations. This hypothesis was empirically validated in the ablation study presented above (Appendix D.1). In general, we used the following heuristics to choose the depth of the transformers:

$$\text{(length of the deepest path in the graph)} \times \text{(steps per operation)} + \text{a few more layers.}$$

Determining the number of steps needed by the elementary operation is not straightforward, but can be done empirically. For example, for ListOps, as shown in Sec. 5.3, requires two steps per operation: one step in which the operands attend to the operation, followed by another one where the result is written back to the operation. For other tasks, we found that a single step per operation was enough. Choosing more layers than needed is safe, and it can be used to determine the required number of layers, for example, by looking at the gate activity. Finally, "+ a few more layers" are needed because an additional layer should be used to read out the final result, and one or a few more can be needed for communication between columns (e.g., to determine operator precedence).

Since the parameters are shared across layers, we can optionally train models with a certain number of layers and increase the number of computational steps at test time. This allows us to train models using a depth that is "sufficient" to solve the training set but increases it at test time to generalize to a test set requiring more computational steps. We did this for the ListOps experiment (Sec. 5.2.3): the model was trained with 20 layers and tested with 24. Our preliminary experiments confirmed that this practice has no performance penalty, while it speeds up training.

D.3.2 Dataset Details

Compositional table lookup. Our implementation uses 8 symbols as input arguments and 9 randomly sampled bijective functions denoted by lowercase letters of the English alphabet. All functions are included in the train set in combination with all possible input symbols. The rest of the training set consists of random combinations of functions applied to a random symbol as an argument, up to length 5. The total size of the train set is 53,704 samples. The samples are roughly balanced so that there are similar numbers of samples for each depth. There are different validation sets: an IID set, which matches the distribution of the train set, and a depth validation, which includes samples of lengths 6, 7 and 8. The test set consists of sequences of lengths 9 and 10.

Simple arithmetic. The dataset is constructed by sampling random digits (0-9) and operations $+$ (add) and $*$ (multiply). The operations are performed modulo 10. Parentheses surround the arguments of the operations. The depth of the resulting tree is computed, and rejection sampling is used to ensure that the same number of samples from each depth is present in the given split. The maximum length of samples is 50 tokens, sub-operations are sampled with probability 0.2. 100 K samples are used for training, 1 K for both test and validation sets. The train set consists of 0-5 operations, the validation set of 6 and the test set of 7 operations.

ListOps. Random digits are sampled from range 0-9. Operations are sampled from the set sum-modulo (`SM`), which is a sum modulo 10, min (`MIN`), max (`MAX`) and median followed by the floor function (`MED`). The maximum number of arguments for each operation is 5. A sub-operation is sampled with probability 0.3. 1 M samples are used for training, 1 K for test and validation. The train set consists of 0-5 operations, 6 for the validation set, and 7 for the test set.

For each sample, we calculate a number that we call *dependence depth*. To understand it, note that the MIN and MAX operations only select one of their operands, MED selects 1 or 2. In SUM, all operands are needed to perform the operation. If we construct a parse tree and prune away the branches that were *not* selected by any operation and measure the depth of such a tree, the resulting number is the dependency depth. This ensures that the deeper parts of the tree contribute to the result calculation, preventing shallow heuristics, such as ignoring all branches of the tree that are too deep and still getting the correct result with a high probability. We also ensure that the number of samples is the same for all possible dependency depths in each split.

D.3.3 Model Details

We use the AdamW optimizer [Loshchilov and Hutter, 2019] for all our models. Standard hyperparameters are listed in Tab. D.3, D.4 and D.5. Additionally, models with gating use dropout [Hanson, 1990; Srivastava et al., 2014] applied to the content-based query and the position-query components of 0.1 for most models, except for non-gated transformers on ListOps, where this value is 0.05. In the case of geometric attention, since the channels of the directional encoding does not have any redundancy, dropout is applied just to the content query.

In the case of transformers with the copy gate but without geometric attention, we use $\tanh$ instead of $\mathrm{LayerNorm}$ in Eq. 5.2. The transformer/NDR layer with a copy gate is illustrated in Fig. 5.1.

The hyperparameters of the gateless transformers differ significantly from the gated ones. This is because they were very hard to train to achieve good performance even on the IID set, requiring extensive hyperparameter tuning. One might argue that fewer layers make them less competitive on longer sequences. However, we were unable to train them to perform well even on IID data with comparable sizes.

All transformer variants have a begin (B) and end (E) token included in the sequence. RNNs (LSTM and DNC) do not have such tokens. All transformers are encoders only, and the results are read from the last column (corresponding to the end token).

The DNC has 21 memory cells, 4 read heads, and an LSTM controller. It contains recently introduced improvements [Csordás and Schmidhuber, 2019].

We use gradient clipping with magnitude 5 (for CTL) or 1 (for simple arithmetic and ListOps) for all of our models.

The hyperparameters were obtained by a Bayesian hyperparameter search of Weights & Biases[1] over the systematically different (OOD) validation set for the `+abs/rel + gate` models and were reused for all other gated models. For the non-gated models, we used the `+rel` variant for tuning. It was not possible to tune the baselines using only the OOD validation set because their performance was too bad on that set. Therefore, we used a mixture of IID and OOD validation sets to tune the hyperparameters for the baselines. Tab. D.6 shows the range of hyperparameters used for tuning. "FF multiplier" is used to calculate d_{FF} from d_{model}.

We train all models for a fixed number of n_{iters} iterations and measure their validation performance every 1000 iterations. For each model, we select the best checkpoint according to the validation performance, and report its test accuracy.

[1] `https://wandb.ai/`

D.4 Additional Analysis

D.4.1 Compositional Table Lookup

An idealized sequence of computations in a transformer for an example from CTL task is shown in Fig. D.2. Each column waits for its input from the left side, then performs an update. Finally, the last column copies the result. So far, in the main text, we only had space to show the gate and attention activity of the NDR for a few timesteps. Here we show the corresponding visualization of all steps in Figures D.6 and D.7, as well as the attention map for the baseline transformer with relative positional encoding in Fig. D.3. We also show the `Transformer + abs/rel + gate` variant in Fig. D.4 and Fig. D.5. Please directly refer to the caption of the figures for the corresponding analysis. In general, the visualization for our NDR and the `abs/rel + gate` variant is easily interpretable, unlike that of the baseline transformer model.

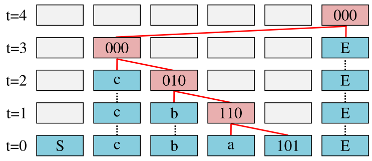


Figure D.2: An ideal sequence of computations in a transformer for an example CTL task.

D.4.2 ListOps

Figures D.8 and D.10 show the attention and gate patterns of our NDR architecture on an example from the ListOps dataset. We highlighted notable attention patterns in Sec. 5.3.

Different heads seem to specialize in different functions. As already mentioned in Sec. 5.3, *head 13* of the NDR architecture, shown in Fig. D.9, seems to specialize in selecting which arguments belong to which operator.

The gating patterns are also very interesting. In the early stages, the deepest parts of the input are updated: `[MAX 2 4 0 8 9]` and `[MED 8 5 8]`, which are independent branches of the parse tree that can be processed in parallel. In the following steps, the update patterns spread up in the parse tree, updating the operations that have their arguments available. In this task, the input is read from the first column, which is written in a very late stage.

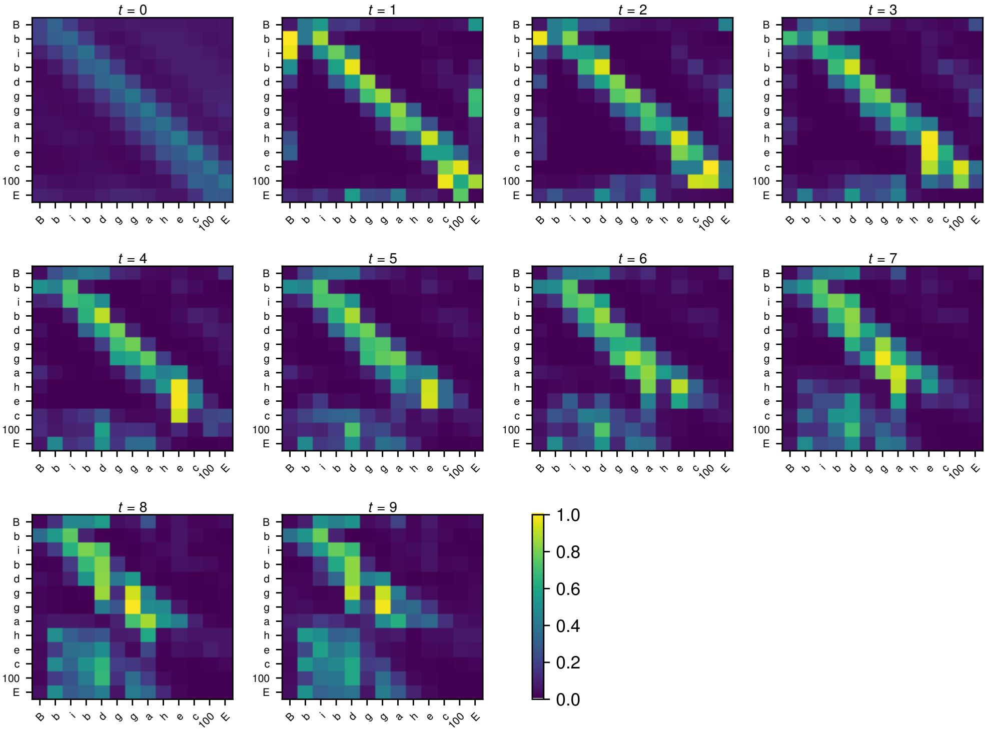


Figure D.3: Attention map for every computational step for a baseline transformer with relative positional encoding on CTL. The attention pattern becomes blurry very quickly and the model does not generalize to longer sequences.

		IID		Longer, 60k		Longer, 30k	
Model	ACT	Forward	Backward	Forward	Backward	Forward	Backward
Transformer		1.00 ± 0.00	0.82 ± 0.39	-	-	0.13 ± 0.01	0.12 ± 0.01
	A	1.00 ± 0.00	1.00 ± 0.00	0.12 ± 0.02	0.12 ± 0.01	0.13 ± 0.01	0.13 ± 0.01
	U	1.00 ± 0.00	1.00 ± 0.00	0.12 ± 0.01	0.11 ± 0.01	0.13 ± 0.01	0.12 ± 0.01
+ rel		1.00 ± 0.00	1.00 ± 0.00	-	-	0.23 ± 0.05	0.13 ± 0.01
	A	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.02	0.13 ± 0.00	0.84 ± 0.22	0.13 ± 0.01
	U	1.00 ± 0.00	1.00 ± 0.00	0.92 ± 0.14	0.12 ± 0.02	0.67 ± 0.41	0.12 ± 0.00
+ geo		0.96 ± 0.04	0.93 ± 0.06	-	-	0.16 ± 0.02	0.15 ± 0.02
	A	1.00 ± 0.00	1.00 ± 0.00	0.97 ± 0.05	0.45 ± 0.21	0.58 ± 0.16	0.30 ± 0.17
	U	0.96 ± 0.10	1.00 ± 0.00	0.72 ± 0.35	0.44 ± 0.19	0.31 ± 0.22	0.21 ± 0.07
+ rel + gate		1.00 ± 0.00	1.00 ± 0.00	-	-	**0.99 ± 0.01**	0.19 ± 0.04
+ abs/rel + gate		1.00 ± 0.00	1.00 ± 0.00	-	-	**0.98 ± 0.02**	**0.98 ± 0.03**
+ geo + gate (NDR)		1.00 ± 0.00	1.00 ± 0.00	-	-	**1.00 ± 0.00**	**1.00 ± 0.00**

Table D.2: Accuracy on **compositional table lookup** dataset with adaptive computation time (ACT). Two variants of ACT are shown: "U" corresponds to Dehghani et al. [2019], while "A" is the variant described in Appendix D.1. We also include baselines without ACT from Tab. 5.1 as a reference. Generalization performance after 30k and 60k **training steps** are shown.

	d_{model}	d_{FF}	n_{heads}	n_{layers}	batch s.	learning rate	wd.	do.	n_{iters}
LSTM	200	-	-	1	256	$10 * 10^{-4}$	-	0.5	20k
Bidirectional LSTM	400	-	-	1	256	$10 * 10^{-4}$	-	0.5	20k
DNC	200	-	-	1	256	$10 * 10^{-4}$	-	0.5	20k
Transformer	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.1	30k
+ rel	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.1	30k
+ rel + gate	256	512	1	14	512	$2 * 10^{-4}$	0.01	0.5	30k
+ abs/rel + gate	256	512	1	14	512	$2 * 10^{-4}$	0.01	0.5	30k
+ geom. att.	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.1	30k
+ NDR	256	512	1	14	512	$1.5 * 10^{-4}$	0.01	0.5	30k

Table D.3: Hyperparameters used for different models on the compositional table lookup task. We denote the feedforward size as d_{FF}, weight decay as “wd.”, dropout as “do.”. The model is trained for n_{iters} iterations.

	d_{model}	d_{FF}	n_{heads}	n_{layers}	batch s.	learning rate	wd.	do.	n_{iters}
LSTM	200	-	-	2	256	$10 * 10^{-4}$	-	0.5	200k
Bidirectional LSTM	400	-	-	2	256	$10 * 10^{-4}$	-	0.5	200k
Transformer	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.5	200k
+ rel	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.5	200k
+ abs/rel + gate	256	1024	4	15	512	$1.5 * 10^{-4}$	0.01	0.5	100k
+ NDR	256	1024	4	15	512	$1.5 * 10^{-4}$	0.01	0.5	100k

Table D.4: Hyperparameters used for different models on the simple arithmetic task. We denote the feedforward size as d_{FF}, weight decay as "wd.", dropout as "do.". The model is trained for n_{iters} iterations.

	d_{model}	d_{FF}	n_{heads}	n_{layers}	batch s.	learning rate	wd.	do.	n_{iters}
LSTM	512	-	-	4	512	$10 * 10^{-4}$	0.08	0.1	200k
Bidirectional LSTM	1024	-	-	4	512	$10 * 10^{-4}$	0.08	0.1	200k
Transformer	256	1024	16	6	512	$4 * 10^{-4}$	0.05	0.015	200k
+ rel	256	1024	16	6	512	$4 * 10^{-4}$	0.05	0.015	200k
+ abs/rel + gate	512	1024	16	20	512	$2 * 10^{-4}$	0.09	0.1	100k
+ NDR	512	1024	16	20	512	$2 * 10^{-4}$	0.09	0.1	100k

Table D.5: Hyperparameters used for different models on the ListOps task. We denote the feedforward size as d_{FF}, weight decay as "wd.", dropout as "do.". The model is trained for n_{iters} iterations.

Parameter	Range
learning rate	0.00005 ... 0.001
n_{layers}	4 ... 20
d_{model}	128, 256, 512
n_{heads}	2, 4, 8, 16
weight decay	0.0 ... 0.1
dropout	0.0 ... 0.5
attention dropout	0.0 ... 0.5
FF multiplier	1, 2, 4

Table D.6: Parameter ranges for hyperparameter tuning

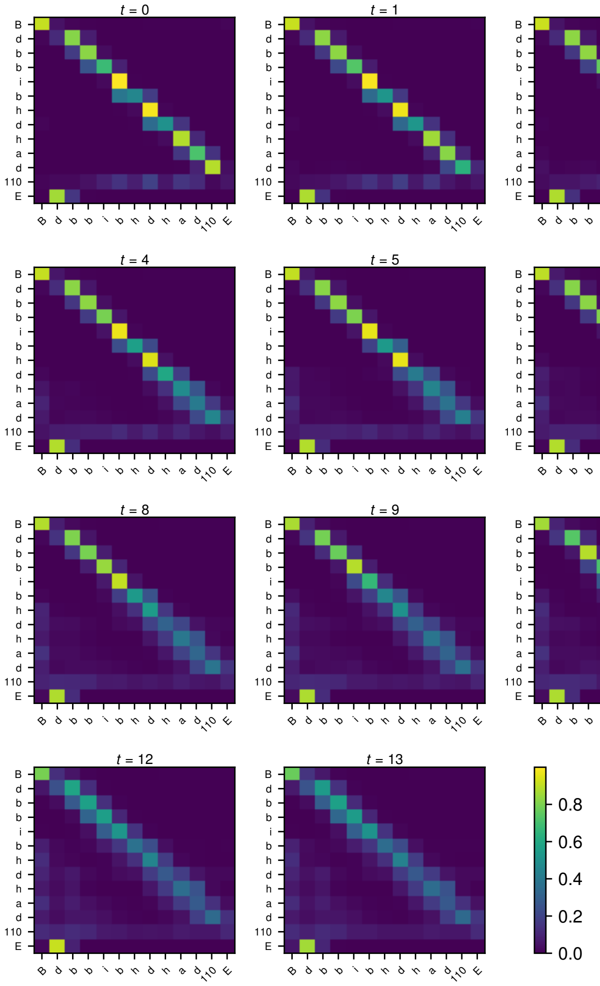


Figure D.4: Attention map for every computational step for a transformer with gating and relative/absolute positional encoding (presented in Fig. 5.3) on CTL. The attention pattern is relatively stable over time, and it gets blurrier only after the given column is processed and updated. The gate sequence for the same input can be seen in Fig. D.5.

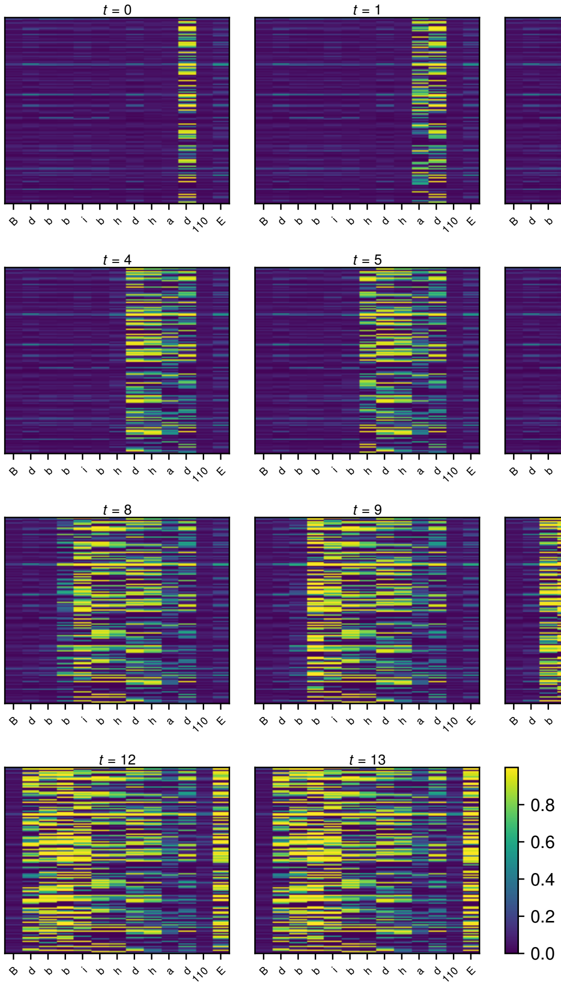


Figure D.5: Gates for every computational step for a transformer with gating and relative/absolute positional encoding on CTL. The gates are closed until all arguments of the given operation become available. The attention maps for the same input can be seen in Fig. D.4.

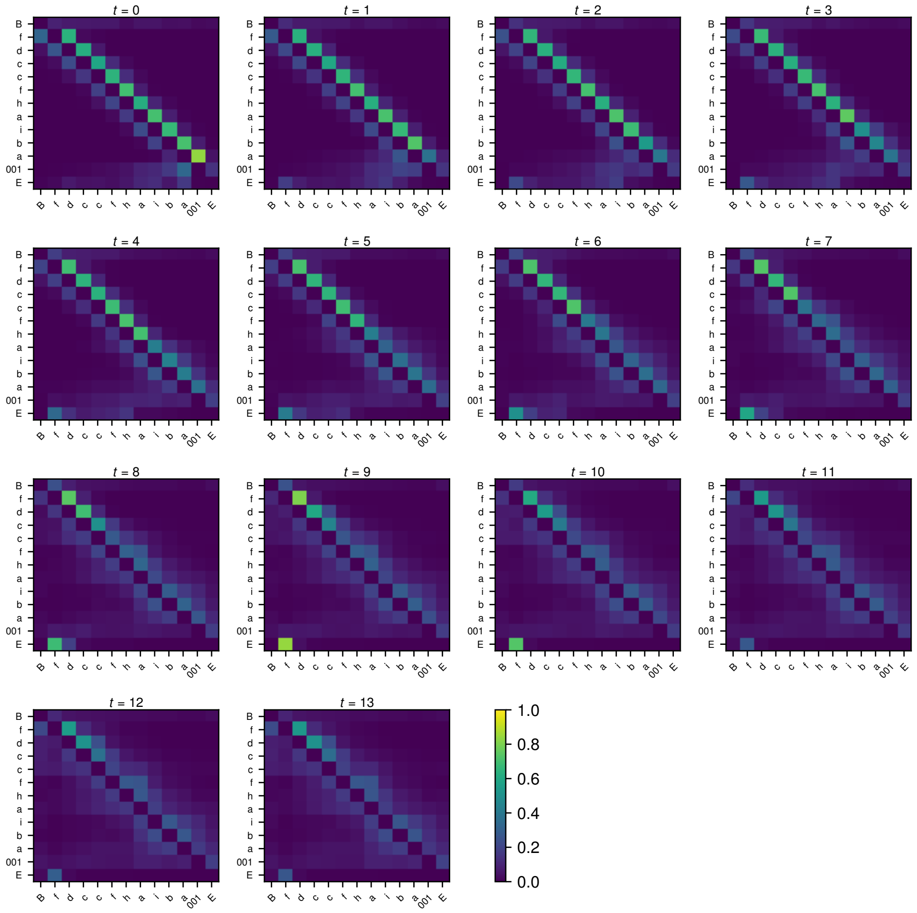


Figure D.6: Attention map for every computational step of the NDR on CTL. The network correctly and clearly focuses on the last element of the sequence, and the last sharp read happens in step 10 - corresponding to the 10 function calls in the example. The gate sequence for the same input can be seen in Fig. D.7.

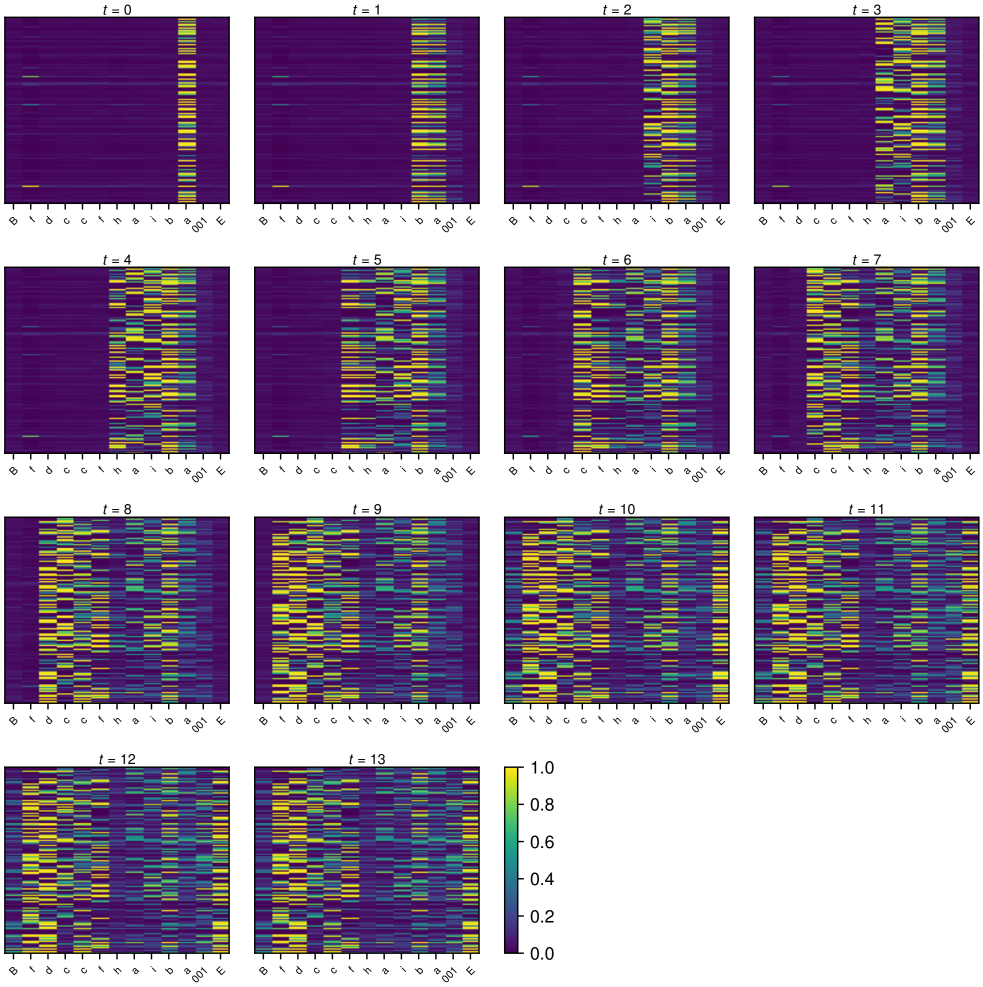


Figure D.7: Gates for every computational step of the NDR on CTL. The gates remain closed until all arguments of the given operations become available. The attention maps for the same input can be seen in Fig. D.6.

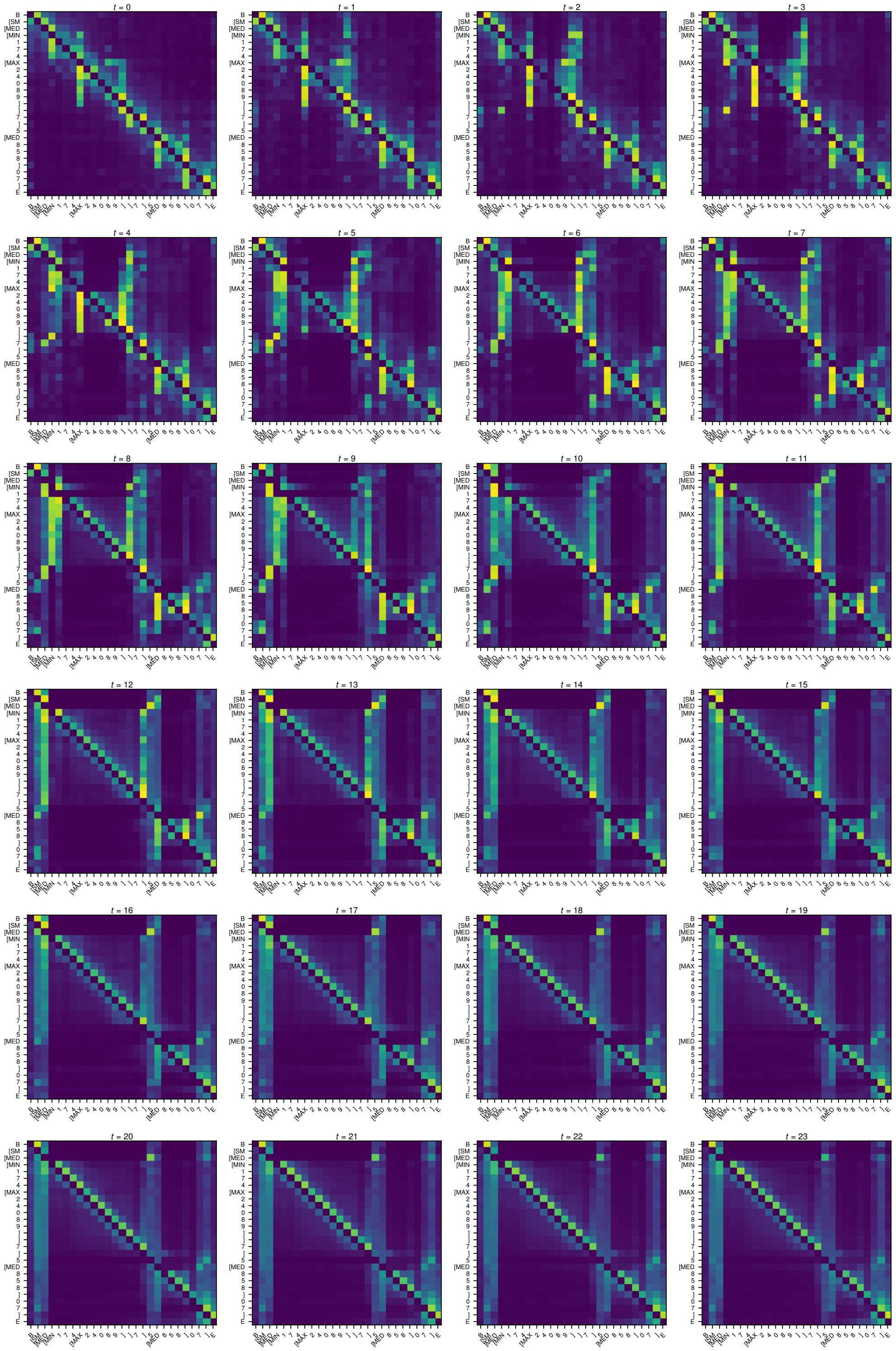


Figure D.8: Attention maps for every computational step of the NDR on ListOps. The network has 16 heads; the max of them is shown. The input has only depth 4, which explains the early stopping of the computation, roughly after 8-9 steps, after which the attention barely changes. The corresponding gate maps for the same input can be seen in Fig. D.10.

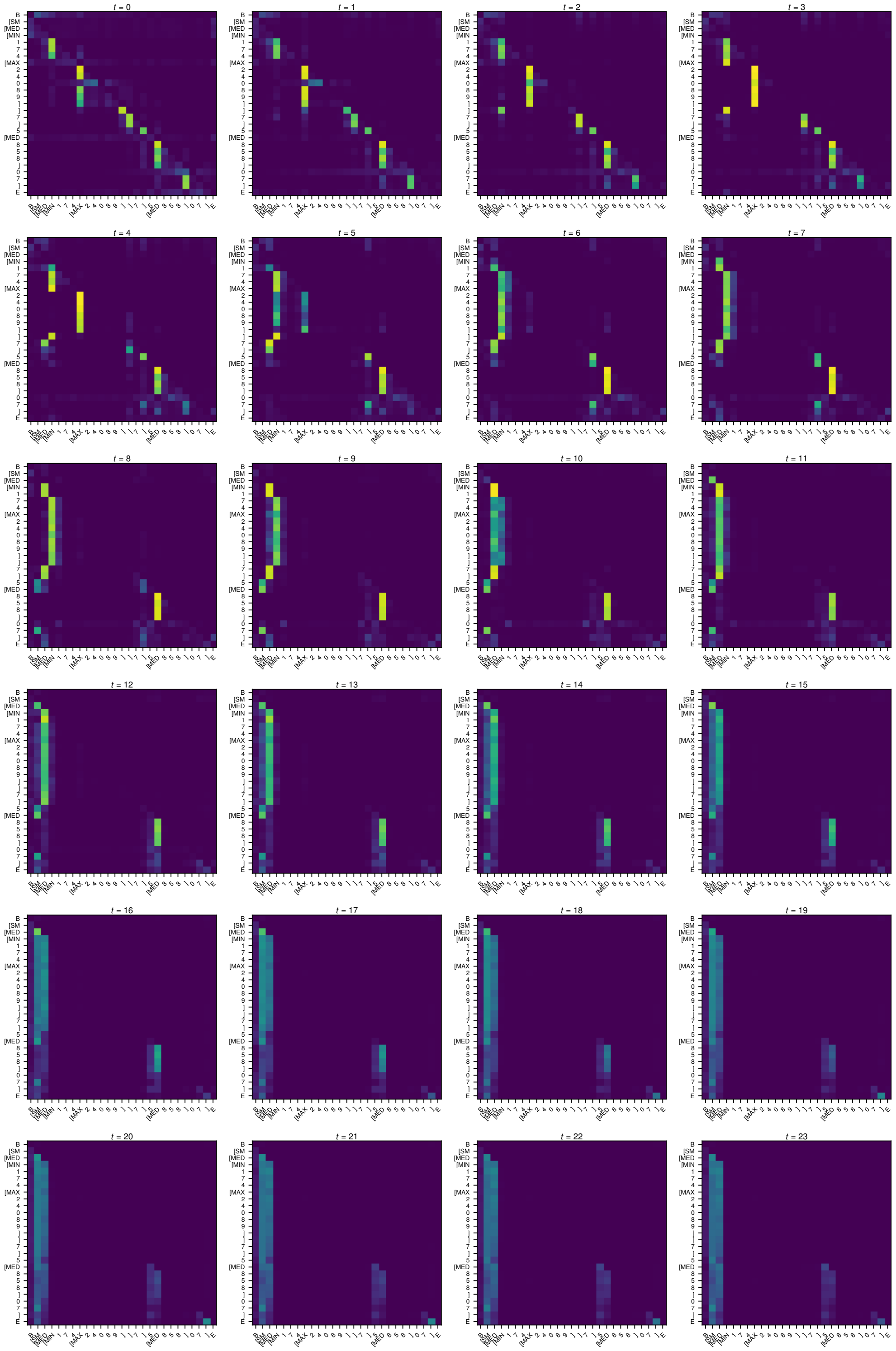


Figure D.9: Attention maps for *head 13* of the NDR in every computational step on ListOps. This head shows the operands for each operation. Following it, we observe the hierarchy and the order in which the operations are performed.

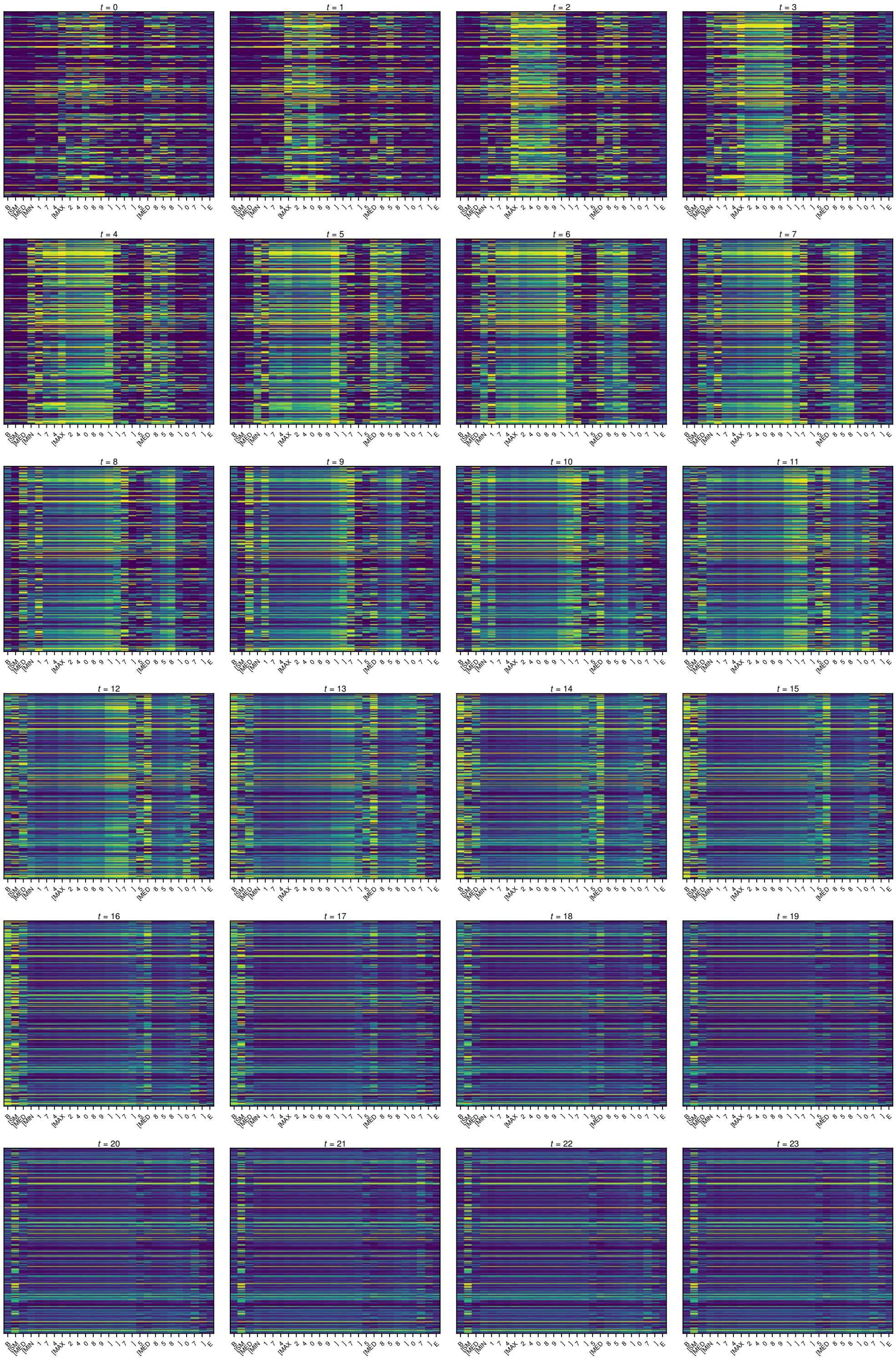


Figure D.10: Gates for every computational step of the NDR on ListOps. Gates open for the deepest operations in the tree, processing proceeds upwards in the computational tree. The input has only depth 4, which explains the early stopping of the computation, roughly after 8-9 steps. The attention maps for the same input can be seen in Fig. D.8.

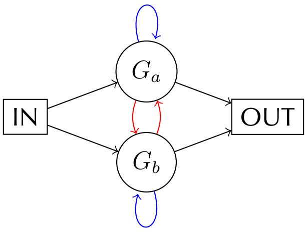


Figure E.1: Sampling graph for variant 'R.'

Appendix E

More Details on Inspecting Systematicity of Neural Networks

E.1 Experimental Details

E.1.1 Modified NDR architecture

We experimentally found some architectural modifications to the original NDR [Csordás et al., 2022a] that yield faster convergence and produce more stable results than the original architecture. Here we describe our modifications. We use the GELU activation function [Hendrycks and Gimpel, 2016] instead of ReLU [Fukushima, 1969], and a residual connection in the feedforward data path. Concretely, Eq. 5 by Csordás et al. [2022a] is replaced by Eq. E.1 below, while Eq. 2 is replaced by Eq. E.2 below. We do not use any dropout in Eq. E.1.

$$\mathrm{FFN}(\boldsymbol{x}) = \boldsymbol{W}_2\,\mathrm{GELU}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \tag{E.1}$$

$$\boldsymbol{u}^{(i,t+1)} = \mathrm{LN}(\mathrm{FFN}^{\mathrm{data}}(\boldsymbol{a}^{(i,t+1)}) + \boldsymbol{a}^{(i,t+1)}) \tag{E.2}$$

where LN denotes layer normalization [Ba et al., 2016].

E.1.2 Hyperparameters

Dataset. The train set in all of our experiments consists of 300k examples. The maximum number of composed functions is 6. We make sure that we obtain an equal number of samples for different lengths whenever possible (in some cases this is impossible because by construction, there are fewer short examples than long ones). In 'A' and 'R' variants, single functions are always part of the training set with all possible symbols. There are in total 8 symbols and 32 functions. All samples are presented in a right-to-left manner (e.g. "c b a 3"). The IID and OOD test sets contain 1000 examples in all cases. Our code generates the data for given dataset specifications (number of functions etc). The seed for data generation is fixed.

Training. Unless noted otherwise, for all of our models, we use a batch size of 512, a learning rate of 0.00015, and a dropout rate [Hanson, 1990; Srivastava et al., 2014] of 0.5. We also use a linear learning rate warmup for the first 500 iterations. We use PyTorch's [Paszke et al., 2019] adaptive mixed precision and bin the batches by length for greater efficiency. We use the AdamW optimizer [Loshchilov and Hutter, 2019]. The NDR and bi-LSTM is trained for 80k and the transformers for 300k iterations. We find the standard transformer to be very unstable even in the IID setting. In fact, for Tab. 6.1, unlike other models trained for 25 seeds, we train the transformer for 50 seeds: the 25 seeds used to report mean and std in Tab. 6.1 are those among 50 which converged within 300k training iterations. For Figs. 6.4, E.3 and E.4, five seeds were used for each configuration. All of our models are trained on a single P100 GPU. The corresponding number of parameters, training steps and average wall-clock time is shown in Tab. E.1.

Model	Num. params	Num. steps	Runtime
Bi-LSTM	408k	80k	0:18
Transformer	672k	300k	3:37
NDR	679k	80k	1:22

Table E.1: Training details for different models. Runtime is in hour:min.

Models. For transformer and NDR, we use 8 layers, and 4 heads. NDR uses a gate dropout of 0.1, state size of 256, feedforward size of 1024. Transformers use a state size of 128 and feedforward size of 512, layer sharing [Dehghani et al., 2019], and Transformer-XL-style [Dai et al., 2019] relative positional encoding. For bidirectional LSTM, we use 1 layer with 256 units (128 per direction). The gradient is clipped to max norm of 1 for NDR and 5 for transformer and LSTM. Transformers use a weight decay of 0.0025.

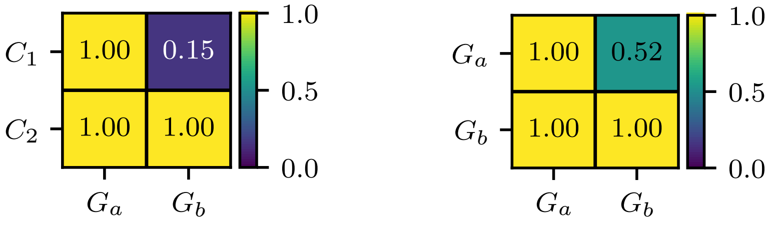
E.2 More Analyses and Plots

E.2.1 Quantitative Analysis of Incompatibility

Here we provide additional results on the analysis of Sec. 6.3.1 conducted for variant 'R.' To quantify the correlation between the clusters (C_1 and C_2) identified in Fig. 6.3b and the compatibility of representations, we measured the proportion of correct output classifications, by taking the first function from a given cluster and the second one from a given group, for all pairs of functions for each pair of the form (cluster, group). The results are shown in Fig. E.2. Fig. E.2a shows that the first cluster C_1 is effectively compatible only with functions from G_a, while the second one C_2 works with both G_a and G_b, as predicted by Fig. 6.3b. Fig. E.2b shows the same analysis, but uses the groups to define the cluster. As predicted by Fig. 6.3b, only roughly half of the functions from G_a generate representations compatible with G_b, while all representations generated by functions in G_b are compatible with all in G_a.

E.2.2 Representative Cosine Similarities

Here we show additional visualizations similar to those of Fig. 6.3. In Fig. E.5 and E.6, we plot cosine similarities of functional outputs for all possible symbols for successful and failed seeds of NDR on variant 'R.' The symbol representations are taken from the layer right below the final classification layer. They are representative examples; the observation holds over all seeds we inspected. Fig. E.7 shows a similar example for variant 'A.'



(a) Performance of each cluster vs. each group

(b) Performance for each pair of groups

Figure E.2: Accuracy measured after two successive function applications, for the symbol corresponding to Fig. 6.3b. (a) shows the proportion of correct outputs when the first function is taken from a given cluster (y-axis), and the second from a given group (x-axis). Clusters are shown in the main diagonal of Fig. 6.3b. (b) is analogous to (a) but using groups as clusters.

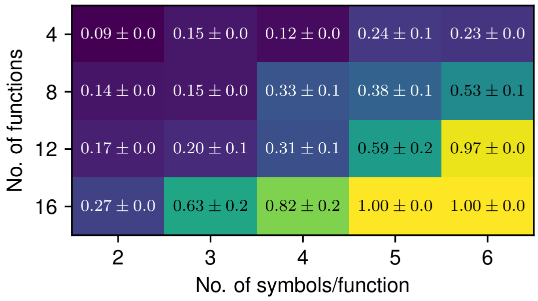


Figure E.3: Final test performance of the transformer on variant 'S.' The behavior is similar to the one shown in Fig. 6.4. For lower numbers of shared functions, performance is worse. Interestingly, however, with 16 shared functions, it outperforms NDR.

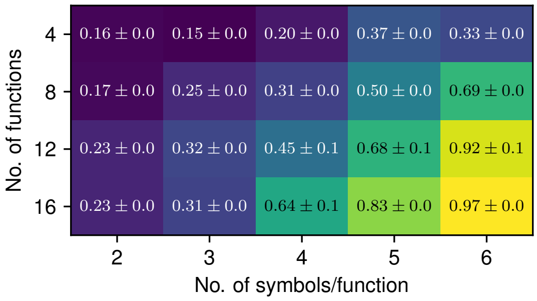


Figure E.4: Final test performance of bi-directional LSTM on variant 'S.' The behavior is similar to the one shown in Fig. 6.4. For lower numbers of shared functions, the performance is worse. Interestingly, however, with 16 shared functions, it significantly underperforms NDR.

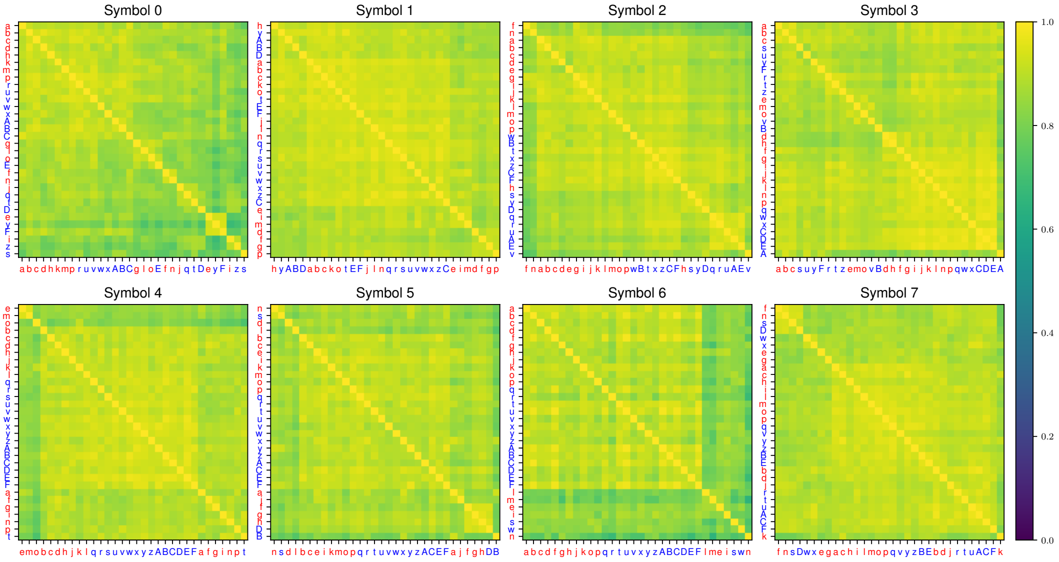


Figure E.5: Symbol cosine similarity between different functions for NDR on variant 'R.' A representative example from a seed that performs **perfectly** on unseen compositions. Functions indicated by red belong to G_a, by blue to G_b.

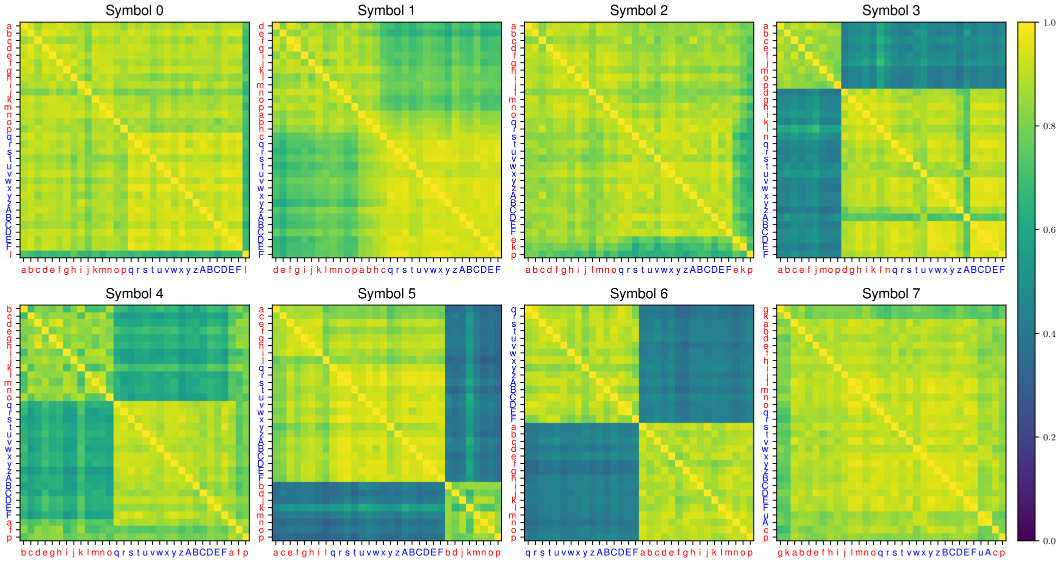


Figure E.6: Symbol cosine similarity between different functions for NDR on variant 'R.' A representative example from a seed that performs **poorly** on unseen compositions. Functions indicated by red belong to G_a, by blue to G_b.

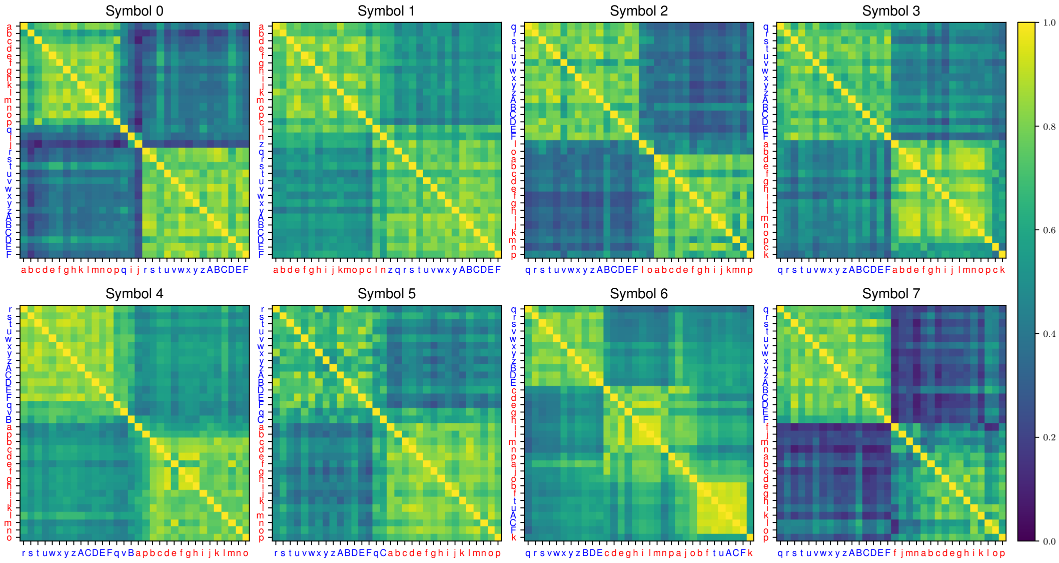


Figure E.7: Symbol cosine similarity between different functions for NDR on variant 'A.' A representative example from a seed that performs **poorly** on unseen compositions. Functions indicated by red belong to G_a, by blue to G_b.

Appendix F

Additional Details of Accelerating Transformer MLP Layers: a Path Towards Scalable NDRs

F.1 Further Details and Analyses

F.1.1 Definition of normalised Top-K

Using the setting of Sec. 7, we define the *normalized top-K* operation as follows:

$$\mathcal{E}_{\boldsymbol{x}} = \arg\operatorname{topk}(\boldsymbol{s}, K) \tag{F.1}$$

$$\operatorname{topk}(\boldsymbol{s})[i] = \begin{cases} \boldsymbol{s}[i] & \text{if } i \in \mathcal{E}_{\boldsymbol{x}} \\ 0 & \text{otherwise} \end{cases} \tag{F.2}$$

$$\operatorname{norm\,topk}(\boldsymbol{s}) = \frac{\operatorname{topk}(\boldsymbol{s})}{\sum_i \operatorname{topk}(\boldsymbol{s})[i]} \tag{F.3}$$

F.1.2 Measuring the Number of Active Channels in

In order to explore whether a ($\boldsymbol{k}_i$ - $\boldsymbol{v}_i$) sparsity-based approach is feasible, we measure the number of nonzero entries in the up-projected vector $\boldsymbol{u}$ in our baseline models (which, because of the ReLU activation function, is the same as the positive entries). We show the results of our 47M model in Fig. 7.1. Note that $d_{\text{ff}} = 2053$ (See Tab. F.2) for the same model, which means that on average only 1-10% of the channels are active. We show the same analysis for the 262M model in Fig. F.1. Interestingly, the counts remain the same, even though $d_{\text{ff}} = 4110$ for this model. The 41M parameter model on Enwik8 shows a stark difference in the distribution of the channels between layers; see Fig. F.2. This suggests that the key factor determining the count distribution is the dataset, and the size of the model plays only a secondary role. Fortunately, the sparsity is very high for all models considered.

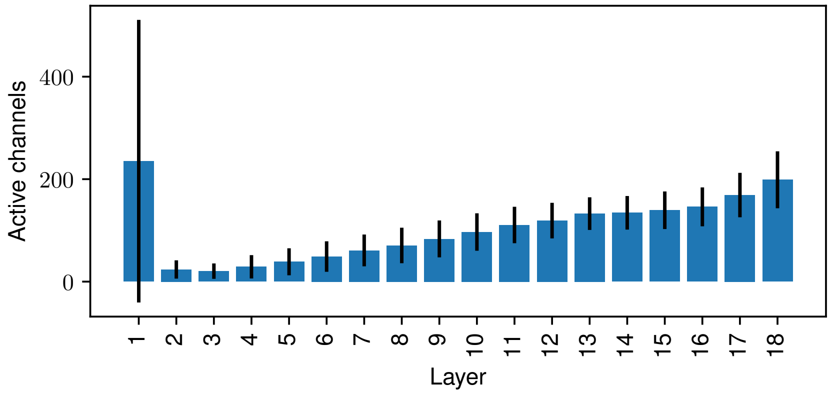


Figure F.1: Number of active channels in $\boldsymbol{u}$ in our dense 262M parameter model on Wikitext-103. $d_{\text{ff}} = 4110$ for this model, so the sparsity is below $\sim 5\%$. Standard deviation over all tokens of the test and validation set.

F.1.3 More Details and Results on PKM

Our PKM (Sec. 7.2.2) is based on Lample et al. [2019] with the following basic modifications. First, we do not use batch normalization (BN). As Lample et al. [2019] shows that BN is only beneficial for models with a very large memory size, we remove it as it simplifies inference where the effective batch size varies over time. Also, we directly divide the input vectors into two sub-keys without an additional projection. Finally, unlike Lample et al. [2019], we use the same learning rate for all parts of the network.

In addition to the parameter-equal comparison of Sec. 7.5.2, there is another possibly "fair" way of setting the size of the PKM-based model: match the number of values (this would result in fewer parameters because of the key approximation), even though Elhage et al. [2022] suggest that the keys typically play a vital role, and reducing their capacity will cause a performance loss. See Tab. F.1 for the corresponding results. Note that, for Enwik8 and Wikitext-103 small, the parameter-equal setting increases the number of sub-keys from 46 to 62 (2116 vs. 3844 values). This helps significantly.

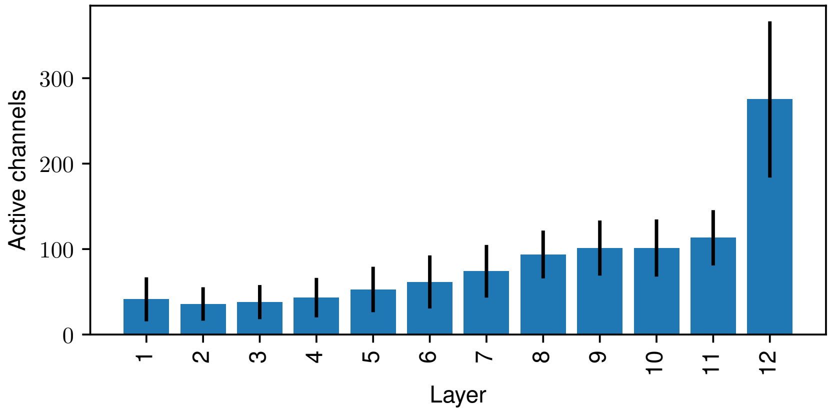


Figure F.2: Number of active channels in $\boldsymbol{u}$ in our dense 41M parameter model on Enwik8. $d_{\text{ff}} = 2053$ for this model, thus the sparsity is below $\sim 15\%$. Standard deviation over all tokens of the test and validation set.

F.1.4 Further Analyses of Our σ-MoE

We also examine the best (G, K) given a constant number $(G \cdot K)$ of active pairs $\boldsymbol{k}_i$, $\boldsymbol{v}_i$. In this setting, reducing K by a factor of m $(K' = \frac{K}{m})$ involves increasing G $(G' = mG)$, which, for a constant number of parameters, reduces N_E to $N'_E = \frac{N_E}{m}$. The results can be seen in the 2^{nd} block of Tab. F.4. We find that a higher K is beneficial. Given this, we ask the question how the selection distribution of the models with $K > 1$ is different from selecting the same experts together and acting as a larger expert. Are these models combining experts in more meaningful ways? To test this, we measure the distribution of experts that are used together on Wikitext-103 with our 47M MoE model with $K = 4$. The result can be seen in Fig. F.3: the network combines experts in a rich way, further supporting the use of $K > 1$. Note that, it remains an open question whether such "compositions" may help the generalization and compositional behavior of the network [Fodor et al., 1988; Pagin and Westerståhl, 2010; Hupkes et al., 2020].

We also include expert usage statistics for all layers of our WT-S* model from Tab. F.4. The results are shown on Fig. F.4. For a more detailed discussion, please refer to Sec. 7.5.3.

Variant	Setting	Nonlinearity	WT-S	WT-M	E8
Dense Baseline		ReLU	11.81	9.46	1.08
PKM	value-count	Softmax	14.11	11.29	1.20
PKM	value-count	ReLU	13.32	10.16	1.12
PKM	# total params.	Softmax	13.96	11.10	1.16
PKM	# total params.	ReLU	12.77	9.98	1.11
PKM + init	# total params.	ReLU	12.75	9.96	1.11

Table F.1: The performance of the PKM model variants. Both value-count and parameter-matched variants are shown. Additionally, we show the effect of the initialization inspired by our unified view, which is marginal for PKMs.

F.1.5 More on Resource Efficiency

For execution time and memory usage, both the dense MLP and the MoE layers are linear in d_{model} (Fig. F.6), the MLP is linear in d_{ff}, and MoE is linear in G (Fig. F.5) and K. For the same number of parameters (except for the selection network, which is negligible), $d_{\text{model}} = G \cdot N_E$. However, both the memory usage and the execution time of the MoE are almost independent of N_E, except for a small linear factor due to the selection network (see Fig. 7.2). Figures 7.2, F.5 and F.6 shows the actual measured execution time and memory usage on a RTX 3090 GPU.

F.2 Implementation Details

We train all of our models for 100k steps with cosine learning rate decay, starting from the initial learning rate of 0.00025 and decaying to 0. We use the Adam optimizer [Kingma and Ba, 2015] with default PyTorch parameters [Paszke et al., 2019]. We use gradient clipping with a max gradient norm of 0.25. We show the other hyperparameters of our dense models in Tab. F.2. We train our models with an XL memory of the same size as the context size. However, following Dai et al. [2019], we evaluate the models using a longer memory. Unlike the hyperparameter-tuned memory sizes in Transformer XL, we use 4 times the context size (this approximates the size of the memory by Dai et al. [2019], while being simple).

The hyperparameters of the MoE models match those of their dense counterparts with the same number of parameters, except for the MoE-specific ones,

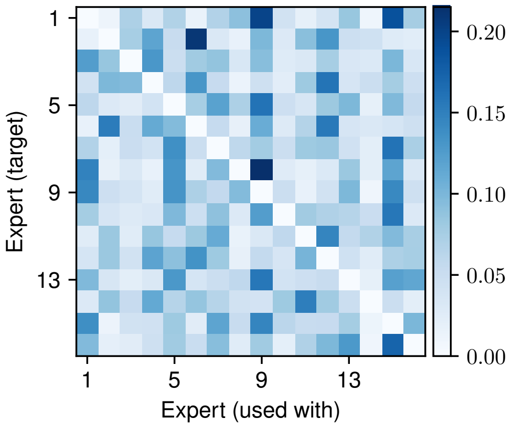


Figure F.3: Expert co-occurrence in a σ-MoE model with $N_E = 16$ experts and $K = 4$. Each row shows the distribution of experts used together with the one corresponding to the row. Measured on the validation set of Wikitext-103 in the 3^{rd} layer of our 47M σ-MoE model. The other layers and models behave qualitatively the same.

which are shown in Tab. F.3. δ denotes the expert dropout and γ denotes the regularization strength used for the loss L (See Eq. 7.20). For the non-MoE layers, the same dropout is used as for the baselines. For Switch Transformers, we use $\gamma = 0.01$ with regularization of the form presented in Eq. 7.16, following Fedus et al. [2022]. The other variants, including S-BASE, use the regularizer proposed by us (Eq. 7.20).

Our small PKM models use 46 subkeys resulting in $46^2 = 2116$ values for the d_{ff}-matched case and 62 subkeys (3844 values) for the parameter-matched case. The PKM equivalent of the 262M parameter model on Wikitext-103 has 64 subkeys (4096 values) for the d_{ff}-matched and 89 subkeys (7921 values) for the parameter-matched case. The PKM models do not use dropout in the PKM layers, and have 4 heads.

F.2.1 A Few Words on the CUDA Kernel

We call the key operation for our MoE layers conditional vector-matrix multiplication, or CVMM, and we define it as follows. Given a batch of vectors, $\boldsymbol{V} \in \mathbb{R}^{N \times M}$, where N is the batch size and M is the number of channels, a set of K matrices $\boldsymbol{M} \in \mathbb{R}^{K \times M \times L}$ and selection indices $\boldsymbol{S} \in \{0, ..., K-1\}^N$,

$\mathrm{CVMM}(\boldsymbol{V}, \boldsymbol{S}, \boldsymbol{M}) \in \mathbb{R}^{N \times L}$ is:

$$\mathrm{CVMM}(\boldsymbol{V}, \boldsymbol{S}, \boldsymbol{M})[n, l] = \sum_{m=0}^{M-1} \boldsymbol{V}[n, m]\boldsymbol{M}[\boldsymbol{S}[n], m, l]$$

Our CUDA kernel is based on the blog post developing a matrix multiplication kernel by Simon Boehm (`https://siboehm.com/articles/22/CUDA-MMM`). However, there are major differences: unlike standard matrix multiplication, in our case, different matrices could be used for different batch elements of the input. In order to be able to reuse matrices fetched from the global memory of the GPU, we first do a preprocessing step: we sort the selection indices, and obtain a reordering vector. This gives us an ordering of the input and output batch elements, such that the consecutive indices are multiplied by the same matrix with high probability. Fortunately, multiple channels have to be fetched/written out at once, so this reordering has minimal overhead. Our kernel has an additional grid dimension compared to standard matrix multiplication, iterating over the matrix index, $k \in \{0, ..., K-1\}$. We find that skipping matrices that do not have any corresponding inputs has minimal overhead. To avoid checking all elements of the reordering vector, we precompute their offsets.

Our kernel uses shared memory and register caching; however, it does not use asynchronous loads, which makes it I/O bound. It also does not support tensor cores and mixed precision. The pre-processing step uses the radix sort from the CUB library. However, computing the offsets requires counting the number of vectors assigned to a single matrix. This information, as well as the offset, which is their sum, are freely available as sub-results that the radix sort computes anyways; however, we found no way of extracting it from the CUB implementation. We estimate that by implementing a more efficient preprocessing step, asynchronous loads, and tensor core support, our kernel can be further accelerated by a factor of two.

F.2.2 Additional Results on MoEs

Additional results of different MoE variants with more model details are shown in Tab. F.4. We repeat the entries from Tab. 7.4 for easier comparison.

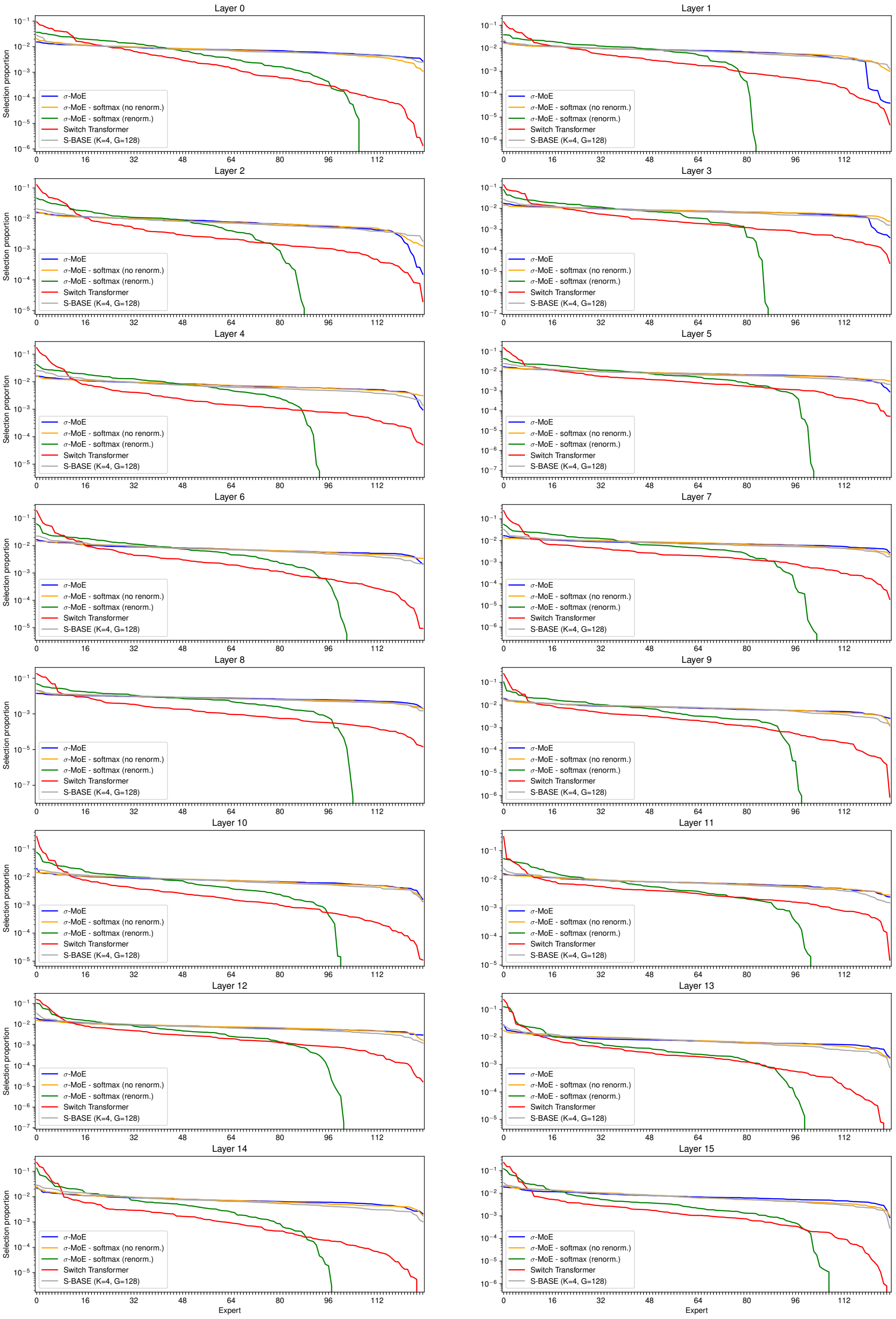


Figure F.4: The total proportion of selection weights assigned to a given expert (indicated on the x-axis) on the validation set of Wikitext-103 with our WT-S* model from Tab. F.4. Experts are sorted by their popularity. All layers are shown. The models with a big performance gap can be distinguished easily (Switch Transformer and σ-MoE with a softmax and renormalization, "softmax (renom.)"). Their performance gap can be at least partially attributed to expert collapse. However, it seems to be difficult to distinguish the fine performance difference between the rest of the models based solely on the expert collapse phenomenon.

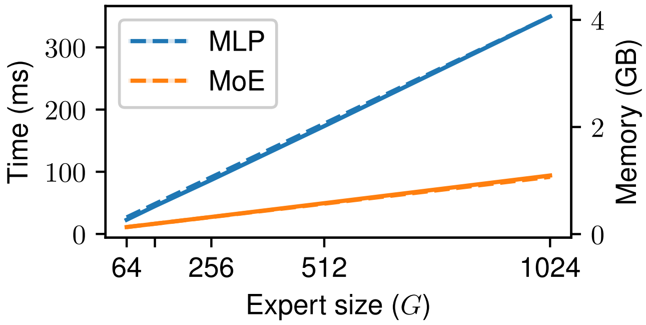


Figure F.5: Measured execution time and memory usage of a forward-backward pass of a single MLP and MoE layer. $|\mathcal{B}| = 32768$, corresponding to the realistic scenario of a batch size 64 and sequence length 512, $d_{\text{model}} = 512$, $K = 4$, $N_E = 32$ and $d_{\text{ff}} = G \cdot N_E$. Full lines show the execution time, and dashed ones the memory consumption. Because they are both linear with similar slopes, they are almost indistinguishable. Even with our suboptimal CUDA kernel, the wall-clock time is faster starting from 16 experts.

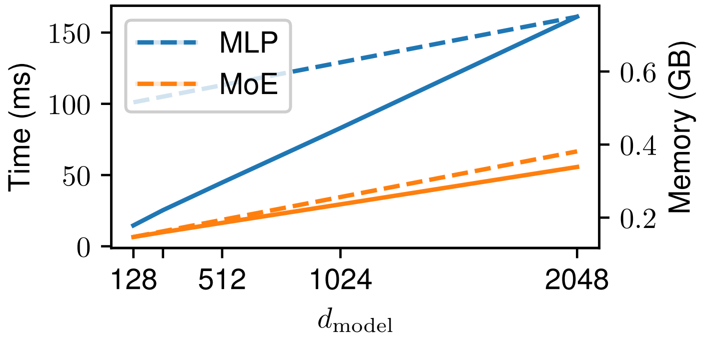


Figure F.6: Measured execution time and memory usage of a forward-backward pass of a single MLP and MoE layer. $|\mathcal{B}| = 32768$, corresponding to the realistic scenario of a batch size 64 and sequence length 512, $K = 4$, $N_E = 32$, $G = 128$ and $d_{\text{ff}} = G \cdot N_E$. Full lines show the execution time, and dashed ones the memory consumption. Even with our suboptimal CUDA kernel, the wall-clock time is faster starting from 16 experts.

Dataset	#params	d_{model}	d_{ff}	n_{layers}	n_{heads}	h.s.	c.s.	b.s.	dropout	lr warm.
Wikitext-103	47M	412	2053	16	10	41	256	64	0.1	-
Wikitext-103	238M	412	16480	16	10	41	256	64	0.1	-
Wikitext-103	262M	1024	4110	18	16	64	512	64	0.2	4000
Enwik8	41M	512	2053	12	8	64	512	32	0.1	-

Table F.2: Hyperparameters of dense baselines and their MoE counterparts. For the MoE-specific hyperparameters, please refer to Tab. F.3. "h.s." is the head size, "c.s." is the context size, "b.s" is the batch size, "lr warm." is the learning rate warmup.

Dataset	#params	d_{model}	N_E	G	K	δ	γ
Wikitext-103	47M	412	16	128	4	-	0.001
Wikitext-103	237M	412	128	128	4	0.05	0.001
Wikitext-103	262M	1024	32	128	4	0.2	0.001
Enwik8	41M	512	16	128	4	0.05	0.0001

Table F.3: MoE-specific hyperparameters for different model variants. γ denotes the scaler for the load balancing term in the loss and δ is the probability of the expert dropout. The standard, transformer-specific hyperparameters are the same as for the baselines. Please refer to Tab. F.2.

Variant			WT-S	WT-S*	WT-B	E8
d_{model}			412	412	1024	512
# params			47M	237M	262M	41M
	G	K				
σ-MoE (ours)	128	4	11.59	10.37	9.44	1.08
standard dropout	128	4	12.01	10.27	9.53	1.08
softmax (after top-k)	128	4	11.89	11.27	9.58	1.09
softmax (before top-k)	128	4	12.05	10.54	9.62	1.09
standard init	128	4	11.80	10.59	9.67	1.08
no reg ($\gamma = 0, \delta = 0$)	128	4	11.83	10.41	9.51	1.08
$K = 8, G = 64$	64	8	11.63	10.30	9.58	1.08
$K = 2, G = 256$	256	2	11.84	10.44	9.56	1.09
$K = 1, G = 512$	512	1	11.90	10.83	9.58	1.09
$N'_E = 2N_E, G = 64$	64	4	11.81	10.53	-	1.08
$K = 1$	128	1	12.26	11.30	-	1.09
$K = 2$	128	2	11.90	10.66	-	1.09
$K = 8$	128	8	11.58	10.22	-	1.08
Switch, $K = 1, G = 512$	512	1	12.27	11.24	9.68	1.08
no dropout	512	1	11.88	11.10	9.77	1.10
$K = 4, G = 128$	128	4	12.05	11.37	-	1.10
$K = 1, G = 128$	128	1	12.61	11.89	-	1.11
no dropout	128	1	12.35	11.78	-	1.10
S-BASE, $K = 4$, $G = 128$	128	4	13.01	10.96	10.50	1.17
$K = 1, G = 512$	512	1	12.32	11.31	9.77	1.32

Table F.4: Detailed ablation results. WT-S* is obtained by naively scaling N_E in WT-S. More details in Sec. 7.5.3. We do not evaluate all versions of the 262M Wikitext-103 model due to its long training time. However, we aim to include what we believe are the most interesting variants. $\gamma = 0$ means no regularization applied to the selection scores (See Eq. 7.20), $\delta = 0$ denotes no expert dropout.

Bibliography

Ekin Akyürek and Jacob Andreas. Lexicon learning for few-shot neural sequence modeling. In *Proc. Association for Computational Linguistics (ACL)*, Virtual only, August 2021.

Ekin Akyürek and Jacob Andreas. Compositionality as lexical symmetry. *Preprint arXiv:2201.12926*, 2022.

Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. In *ICLR, Workshop*, Toulon, France, April 2017. OpenReview.net.

Shun-ichi Amari. A theory of adaptive pattern classifiers. *IEEE Trans. Electron. Comput.*, 16(3):299–307, 1967.

Shun-ichi Amari. Learning patterns and pattern sequences by self-organizing nets of threshold elements. *IEEE Trans. Computers*, 21(11):1197–1206, 1972.

Jacob Andreas. Good-enough compositional data augmentation. In *Proc. Association for Computational Linguistics (ACL)*, pages 7556–7566, Virtual only, July 2020.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proc. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, USA, June 2016.

Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay V. Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, New Orleans, LA, USA, December 2022.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *Preprint arXiv:1607.06450*, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Int. Conf. on Learning Representations (ICLR)*, San Diego, CA, USA, May 2015.

Dzmitry Bahdanau, Harm de Vries, Timothy J O'Donnell, Shikhar Murty, Philippe Beaudoin, Yoshua Bengio, and Aaron Courville. CLOSURE: Assessing systematic generalization of CLEVR models. In *ViGIL workshop, NeurIPS*, Vancouver, Canada, December 2019a.

Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron Courville. Systematic generalization: What is required and can it be learned? In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019b.

Pierre Baldi and Yves Chauvin. Hybrid modeling, HMM/NN architectures, and protein applications. *Neural Computation*, 8(7):1541–1565, 1996.

Andrea Banino, Jan Balaguer, and Charles Blundell. PonderNet: Learning to ponder. *Preprint arXiv:2107.05407*, 2021.

Horace B Barlow, Tej P Kaushal, and Graeme J Mitchison. Finding minimum entropy codes. *Neural Computation*, 1(3):412–423, 1989.

Anthony Bau and Jacob Andreas. How do neural sequence models generalize? local and global context cues for out-of-distribution prediction. *Preprint arXiv:2111.03108*, 2021.

Itamar Ben-Ari and Alan Joseph Bekker. Differentiable memory allocation mechanism for neural computing. *MLSLP*, 2017.

Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine Learning*, 79(1-2):151–175, 2010.

Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *Preprint arXiv:1511.06297*, 2015.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, page 41–48, Montreal, Quebec, Canada, June 2009.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013a.

Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, 2013b.

John Blitzer, Ryan T. McDonald, and Fernando Pereira. Domain adaptation with structural correspondence learning. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 120–128, Sydney, Australia, July 2006.

Ben Bogin, Shivanshu Gupta, and Jonathan Berant. Unobserved local structures make compositional generalization hard. *Preprint arXiv:2201.05899*, 2022.

Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Process. Mag.*, 34(4):18–42, 2017.

Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velickovic. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *Preprint arXiv:2104.13478*, 2021.

Ethan A. Brooks, Janarthanan Rajendran, Richard L. Lewis, and Satinder Singh. Reinforcement learning of implicit and explicit control flow instructions. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 1082–1091, Virtual only, July 2021.

Tom B Brown et al. Language models are few-shot learners. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *Preprint arXiv:2303.12712*, 2023.

Rahma Chaabouni, Roberto Dessì, and Eugene Kharitonov. Can transformers jump around right in natural language? Assessing performance transfer from SCAN. *Preprint arXiv:2107.01366*, 2021.

Michael Chang, Abhishek Gupta, Sergey Levine, and Thomas L. Griffiths. Automatically composing representation transformations as a means for generalization. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, USA, May 2019.

Francois Charton, Amaury Hayat, and Guillaume Lample. Learning advanced mathematical computations from examples. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. Neurosymbolic programming. *Foundations and Trends in Programing Languages*, 7(3):158–243, December 2021.

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 551–561, Austin, TX, USA, November 2016.

Zewen Chi, Li Dong, Shaohan Huang, Damai Dai, Shuming Ma, Barun Patra, Saksham Singhal, Payal Bajaj, Xia Song, Xian-Ling Mao, Heyan Huang, and Furu Wei. On the representation collapse of sparse mixture of experts. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, New Orleans, Louisiana, USA, December 2022.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality, March 2023. URL `https://lmsys.org/blog/2023-03-30-vicuna/`.

N. Chomsky. *Explanatory Models in Linguistics*. 1962.

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking attention with performers. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *Preprint arXiv:2204.02311*, 2022.

Jishnu Ray Chowdhury and Cornelia Caragea. Modeling hierarchical structures with continuous recursive neural networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 1975–1988, Virtual only, July 2021.

Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Proc. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3642–3649, Providence, RI, USA, June 2012.

Aidan Clark, Diego de Las Casas, Aurelia Guy, Arthur Mensch, Michela Paganini, Jordan Hoffmann, Bogdan Damoc, Blake A. Hechtman, Trevor Cai, Sebastian Borgeaud, George van den Driessche, Eliza Rutherford, Tom Hennigan, Matthew Johnson, Katie Millican, Albin Cassirer, Chris Jones, Elena Buchatskaya, David Budden, Laurent Sifre, Simon Osindero, Oriol Vinyals, Jack W. Rae, Erich Elsen, Koray Kavukcuoglu, and Karen Simonyan. Unified scaling laws for routed language models. *Preprint arXiv:2202.01169*, 2022.

Jeff Clune, Jean-Baptiste Mouret, and Hod Lipson. The evolutionary origins of modularity. *Proceedings of the Royal Society B: Biological Sciences*, 280, 2013.

Jeff Clune, Jean-Baptiste Mouret, and Hod Lipson. Summary of "the evolutionary origins of modularity". In *Proc. Int. Conf. on the Simulation and Synthesis of Living Systems (ALIFE)*, pages 41–42, New York, NY, USA, July 2014.

Henry Conklin, Bailin Wang, Kenny Smith, and Ivan Titov. Meta-learning to compositionally generalize. In *Proc. of Annual Meeting of the Association for Computational Linguistics and the International Joint Conf. on Natural*

Language Processing, ACL-IJCNLP, pages 3322–3335, Virtual only, August 2021a.

Henry Conklin, Bailin Wang, Kenny Smith, and Ivan Titov. Meta-learning to compositionally generalize. In *Proc. of Annual Meeting of the Association for Computational Linguistics and the International Joint Conf. on Natural Language Processing, ACL-IJCNLP*, pages 3322–3335, August 2021b.

Róbert Csordás and Jürgen Schmidhuber. Improving Differentiable Neural Computers through memory masking, de-allocation, and link distribution sharpness control. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. The devil is in the detail: Simple tricks improve systematic generalization of Transformers. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, Punta Cana, Dominican Republic, November 2021.

Róbert Csordás, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Are neural nets modular? inspecting functional modularity through differentiable weight masks. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. The neural data router: Adaptive control flow in transformers improves systematic generalization. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, April 2022a.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. CTL++: Evaluating generalization on never-seen compositional patterns of known functions, and compatibility of neural representations. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, Abu Dhabi, United Arab Emirates, December 2022b.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. Approximating two-layer feedforward networks for efficient transformers. *Preprint* *https://openreview.net/forum?id=Gq_fdudbb9*, 2023.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proc. Association for Computational Linguistics (ACL)*, pages 2978–2988, Florence, Italy, 2019.

Verna Dankers, Elia Bruni, and Dieuwke Hupkes. The paradox of the compositionality of natural language: a neural machine translation case study. *Preprint arXiv:2108.05885*, August 2021.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, New Orleans, Louisiana, USA, December 2022.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 933–941, Sydney, Australia, August 2017.

Brian Davis, Umang Bhatt, Kartikeya Bhardwaj, Radu Marculescu, and José MF Moura. On network science and mutual information for explaining deep neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8399–8403, Barcelona, Spain, May 2020.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal Transformers. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

Xiang Deng and Zhongfei (Mark) Zhang. Is the meta-learning idea able to improve the generalization of deep neural networks on the standard supervised learning? In *International Conference on Pattern Recognition, ICPR 2020, Virtual Event / Milan, Italy, January 10-15, 2021*, pages 150–157, Virtual only, January 2020.

Roberto Dessì and Marco Baroni. CNNs found to jump around more skillfully than RNNs: Compositional generalization in seq2seq convolutional networks. In *Proc. Association for Computational Linguistics (ACL)*, pages 3919–3923, Florence, Italy, July 2019.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, New Orleans, Louisiana, USA, December 2022.

Yves Deville and Kung-Kiu Lau. Logic program synthesis. *The Journal of Logic Programming*, 19:321–350, 1994.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional Transformers for language understanding. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pages 4171–4186, Minneapolis, MN, USA, June 2019.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Yann Dubois, Gautier Dagan, Dieuwke Hupkes, and Elia Bruni. Location attention for extrapolation to longer sequences. In *Proc. Association for Computational Linguistics (ACL)*, pages 403–413, Virtual only, July 2020.

Andrew Dudzik and Petar Velickovic. Graph neural networks are dynamic programmers. *Preprint arXiv:2203.15544*, 2022.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaïd Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality. *Preprint arXiv:2305.18654*, abs/2305.18654, 2023.

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy models of superposition. *Transformer Circuits Thread*, 2022.

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *International Conference on Programming Language Design and Implementation (PLDI)*, pages 835–850, Virtual only, 2021.

Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

Jeffrey L. Elman. Learning and development in neural networks: the importance of starting small. *Cognition*, 48(1):71–99, 1993.

Enryu. Gpt4 and coding problems. `https://medium.com/@enryu9000/gpt4-and-coding-problems-8fbf04fa8134`, 2023.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research (JMLR)*, 23(1):5232–5270, 2022.

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. *Preprint arXiv:1701.08734*, August 2017.

Daniel Filan, Shlomi Hod, Cody Wild, Andrew Critch, and Stuart Russell. Neural networks are surprisingly modular. *Preprint arXiv:2003.04881*, 2020.

Jerry Fodor and Brian P McLaughlin. Connectionism and the problem of systematicity: Why Smolensky's solution doesn't work. *Cognition*, 35(2): 183–204, 1990.

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

Jerry Alan Fodor. *The language of thought*, volume 5. Harvard university press, 1975.

Jörg Franke, Jan Niehues, and Alex Waibel. Robust and scalable differentiable neural computer for question answering. *Workshop on Machine Reading for Question Answering (MRQA), ACL*, July 2018.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

Karlis Freivalds, Emils Ozolins, and Agris Sostaks. Neural shuffle-exchange networks - sequence processing in O(n log n) time. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Vancouver, Canada, December 2019.

Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Trans. Syst. Sci. Cybern.*, 5(4):322–333, 1969.

Daniel Furrer, Marc van Zee, Nathan Scales, and Nathanael Schärli. Compositional generalization in semantic parsing: Pre-training vs. specialized architectures. *Preprint arXiv:2007.08970*, 2020.

Adam Gaier and David Ha. Weight agnostic neural networks. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 5365–5379, Vancouver, BC, Canada, December 2019.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800GB dataset of diverse text for language modeling. *Preprint arXiv:2101.00027*, 2021.

Marta Garnelo and Murray Shanahan. Reconciling deep learning with symbolic artificial intelligence: representing objects and relations. *Current Opinion in Behavioral Sciences*, 29:17–23, 2019.

Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5484–5495, Punta Cana, Dominican Republic, November 2021.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proc. Int. Conf. on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256, Sardinia, Italy, May 2010.

Siavash Golkar, Michael Kagan, and Kyunghyun Cho. Continual learning via neural pruning. In *NeurIPS 2019 Workshop Neuro AI*, 2019.

Christoph Goller and Andreas Küchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN)*, pages 347–352, Washington, USA, 1996.

Jonathan Gordon, David Lopez-Paz, Marco Baroni, and Diane Bouchacourt. Permutation equivariant models for compositional generalization in language. In *Int. Conf. on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, April 2020.

Anirudh Goyal, Alex Lamb, Phanideep Gampa, Philippe Beaudoin, Charles Blundell, Sergey Levine, Yoshua Bengio, and Michael Curtis Mozer. Factorizing declarative and procedural knowledge in structured, dynamical environments. In *Int. Conf. on Learning Representations (ICLR)*, Virtual Only, May 2021a.

Anirudh Goyal, Alex Lamb, Jordan Hoffmann, Shagun Sodhani, Sergey Levine, Yoshua Bengio, and Bernhard Schölkopf. Recurrent independent mechanisms. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021b.

Alex Graves. Sequence transduction with recurrent neural networks. In *Workshop on Representation Learning, ICML*, Edinburgh, Scotland, June 2012.

Alex Graves. Adaptive computation time for recurrent neural networks. In *Int. Conf. on Learning Representations (ICLR) Workshop Track*, Vancouver, Canada, April 2016.

Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional LSTM networks for improved phoneme classification and recognition. In *Proc. Int. Conf. on Artificial Neural Networks (ICANN)*, volume 3697 of *Lecture Notes in Computer Science*, pages 799–804, Warsaw, Poland, September 2005.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *Preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John P. Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. On the binding problem in artificial neural networks. *Preprint arXiv:2012.05208*, 2020.

Yinuo Guo, Zeqi Lin, Jian-Guang Lou, and Dongmei Zhang. Hierarchical poset decoding for compositional generalization in language. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

Stephen José Hanson. A stochastic version of the delta rule. *Physica D: Nonlinear Phenomena*, 42(1-3):265–272, 1990.

Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 164–171, Denver, Colorado, USA, November 1992.

John Haugeland. Artificial intelligence: the very idea, 1985.

Serhii Havrylov, Germán Kruszewski, and Armand Joulin. Cooperative learning of disjoint syntax and semantics. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pages 1118–1128, Minneapolis, USA, June 2019.

Horace He. `https://twitter.com/cHHillee/status/1635790330854526981`, 2023.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proc. IEEE Int. Conf. on Computer Vision (ICCV)*, pages 1026–1034, Santiago, Chile, December 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, June 2016.

Mikael Henaff, Jason Weston, Arthur Szlam, Antoine Bordes, and Yann LeCun. Tracking the world state with recurrent entity networks. In *Int. Conf. on Learning Representations (ICLR)*, Toulon, France, April 2017. OpenReview.net.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (GELUs). *Preprint arXiv:1606.08415*, 2016.

Jonathan Herzig and Jonathan Berant. Span-based semantic parsing for compositional generalization. *Preprint arXiv:2009.06040*, 2020.

Jonathan Herzig, Peter Shaw, Ming-Wei Chang, Kelvin Guu, Panupong Pasupat, and Yuan Zhang. Unlocking compositional generalization in pre-trained models using intermediate representations. *Preprint arXiv:2104.07478*, 2021.

Irina Higgins, David Amos, David Pfau, Sebastien Racaniere, Loic Matthey, Danilo Rezende, and Alexander Lerchner. Towards a definition of disentangled representations. *Preprint arXiv:1812.02230*, 2018.

Geoffrey Hinton. Neural networks for machine learning. *Coursera, video lectures.*, 2012.

Geoffrey E Hinton. Distributed representations. *Technical report*, 1984.

Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München*, 91(1):31, 1991.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, pages 1735–1780, 1997.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *Preprint arXiv:2203.15556*, April 2022.

John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. ISBN 0-201-02988-X.

Xiao Shi Huang, Felipe Perez, Jimmy Ba, and Maksims Volkovs. Improving transformer optimization through better initialization. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 4475–4483, Virtual only, July 2020.

Drew A. Hudson and Christopher D. Manning. Compositional attention networks for machine reasoning. In *Int. Conf. on Learning Representations (ICLR)*, Vancouver, Canada, April 2018.

Dieuwke Hupkes, Sara Veldhoen, and Willem Zuidema. Visualisation and 'diagnostic classifiers' reveal how recurrent and recursive neural networks process hierarchical structure. *Journal of Artificial Intelligence Research*, 61: 907–926, 2018.

Dieuwke Hupkes, Anand Singh, Kris Korrel, German Kruszewski, and Elia Bruni. Learning compositionally through attentive guidance. In *Proc. Int. Conf. on Computational Linguistics and Intelligent Text Processing*, La Rochelle, France, April 2019.

Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. Compositionality decomposed: How do neural networks generalise? *Journal of Artificial Intelligence Research*, pages 757–795, 2020.

DeLesley Hutchins, Imanol Schlag, Yuhuai Wu, Ethan Dyer, and Behnam Neyshabur. Block-recurrent transformers. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, New Orleans, LA, USA, December 2022.

Marcus Hutter. A theory of universal artificial intelligence based on algorithmic complexity. *Preprint arXiv:cs/0004001*, 2000.

Kazuki Irie. *Advancing Neural Language Modeling in Automatic Speech Recognition*. PhD thesis, Computer Science Department, RWTH Aachen University, Aachen, Germany, May 2020.

Kazuki Irie, Shankar Kumar, Michael Nirschl, and Hank Liao. RADMM: Recurrent adaptive mixture model with applications to domain robust language modeling. In *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6079–6083, Calgary, Canada, April 2018.

Kazuki Irie, Albert Zeyer, Ralf Schlüter, and Hermann Ney. Language modeling with deep Transformers. In *Proc. Interspeech*, pages 3905–3909, Graz, Austria, September 2019.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear Transformers with recurrent fast weight programmers. *Preprint arXiv:2106.06295*, 2021.

Kazuki Irie, Róbert Csordás, and Jürgen Schmidhuber. The dual form of neural networks revisited: Connecting test time predictions to training patterns via spotlights of attention. In *Proc. Int. Conf. on Machine Learning (ICML)*, Baltimore, MD, USA, July 2022.

E Ising. *Beitrag zur Theorie des Ferround Paramagnetismus*. PhD thesis, PhD thesis, PhD thesis (Mathematisch-Naturwissenschaftliche Fakultät der ..., 1924.

Alekseĭ Grigorievitch Ivakhnenko and Valentin Grigorévich Lapa. *Cybernetic Predicting Devices*. CCM Information Corporation, 1965a.

Aleksey Grigorievitch Ivakhnenko. The group method of data handling – a rival of the method of stochastic approximation. *Soviet Automatic Control*, 13(3):43–55, 1968.

Aleksey Grigorievitch Ivakhnenko. Polynomial theory of complex systems. *IEEE Transactions on Systems, Man and Cybernetics*, 1(4):364–378, 1971.

Aleksey Grigorievitch Ivakhnenko and Valentin Grigorevich Lapa. Cybernetic predicting devices. In *Information and Control*, 1965b.

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

Arthur Jacot, Clément Hongler, and Franck Gabriel. Neural tangent kernel: Convergence and generalization in neural networks. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 8580–8589, Montréal, Canada, December 2018.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparametrization with gumbel-softmax. In *Int. Conf. on Learning Representations (ICLR)*, Toulon, France, April 2017.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C. Lawrence Zitnick, and Ross B. Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proc. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI, USA, July 2017.

Philip N Johnson-Laird, Paolo Legrenzi, and Maria Sonino Legrenzi. Reasoning and a sense of reality. *British journal of Psychology*, 63(3):395–400, 1972.

Armand Joulin and Tomás Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 190–198, Montreal, Quebec, Canada, December 2015.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

Lukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. In *Int. Conf. on Learning Representations (ICLR)*, San Juan, Puerto Rico, May 2016.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 119, pages 5156–5165, Virtual Only, 2020.

Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10): 947–954, 1960.

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. Measuring compositional generalization: A comprehensive method on realistic data. In *Int. Conf. on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.

Eugene Kharitonov and Rahma Chaabouni. What they do when in doubt: a study of inductive biases in seq2seq learners. In *Int. Conf. on Learning Representations (ICLR)*, May 2021.

Najoung Kim and Tal Linzen. COGS: A compositional generalization challenge based on semantic interpretation. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Virtual only, 2020.

Yoon Kim. Sequence-to-sequence learning with latent neural grammars. *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 34, December 2021.

Yoon Kim and Yacine Jernite David Sontag Alexander Rush. Character-aware neural language models. In *Proc. AAAI Conference on Artificial Intelligence*, Phoenix, AZ, USA, February 2016.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *Int. Conf. on Learning Representations (ICLR)*, San Diego, CA, USA, May 2015.

Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning. *Preprint arXiv:2111.09794*, 2021.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13): 3521–3526, 2017a.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114: 3521–3526, 2017b.

Louis Kirsch, Julius Kunze, and David Barber. Modular networks: Learning to decompose neural computation. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 2408–2418, Montréal, CANADA, December 2018.

Tim Klinger, Dhaval Adjodah, Vincent Marois, Josh Joseph, Matthew Riemer, Alex'Sandy' Pentland, and Murray Campbell. A study of compositional generalization in neural models. *Preprint arXiv:2006.09437*, 2020.

Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Irena Gao, Tony Lee, Etienne David, Ian Stavness, Wei Guo, Berton Earnshaw, Imran Haque, Sara M Beery, Jure Leskovec, Anshul Kundaje, Emma Pierson, Sergey Levine, Chelsea Finn, and Percy Liang. WILDS: A benchmark of in-the-wild distribution shifts. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 139, pages 5637–5664, July 2021.

Teuvo Kohonen. Correlation matrix memories. *IEEE Trans. Computers*, 21(4): 353–359, 1972.

Andrei N Kolmogorov. Three approaches to the quantitative definition of information'. *Problems of information transmission*, 1(1):1–7, 1965.

Soheil Kolouri, Nicholas Ketz, Xinyun Zou, Jeffrey Krichmar, and Praveen Pilly. Attention-based structural-plasticity. *Preprint arXiv:1903.06070*, 2019.

Wouter Kool, Chris J Maddison, and Andriy Mnih. Unbiased gradient estimation with balanced assignments for mixtures of experts. In *I (Still) Can't Believe It's Not Better Workshop, NeurIPS*, Virtual Only, 2021.

Kris Korrel, Dieuwke Hupkes, Verna Dankers, and Elia Bruni. Transcoding compositionally: Using attention to find more generalizable solutions. In *Proc. BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, ACL*, pages 1–11, Florence, Italy, 2019.

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

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, volume 25, Stateline, NV, USA, December 2012.

Andreas Küchler and Christoph Goller. Inductive learning in symbolic domains using structure-driven recurrent neural networks. In *Advances in Artificial Intelligence*, volume 1137 of *Lecture Notes in Computer Science*, pages 183–197, Dresden, Germany, September 1996.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 66–71, Brussels, Belgium, October 2018.

Brenden M Lake. Compositional generalization through meta sequence-to-sequence learning. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 9788–9798, Vancouver, Canada, December 2019.

Brenden M. Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 2873–2882, Stockholm, Sweden, July 2018.

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40:e253, 2017.

Yair Lakretz, Germán Kruszewski, Theo Desbordes, Dieuwke Hupkes, Stanislas Dehaene, and Marco Baroni. The emergence of number and syntax units in LSTM language models. In *confNAACL*, pages 11–20, Minneapolis, MN, USA, June 2019. Association for Computational Linguistics.

Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 8546–8557, Vancouver, Canada, December 2019.

Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 598–605, Denver, Colorado, USA, November 1989.

Yann LeCun, Corinna Cortes, and CJ Burges. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Wilhelm Lenz. Beitrag zum verständnis der magnetischen erscheinungen in festen körpern. *Z. Phys.*, 21:613–615, 1920.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. BASE layers: Simplifying training of large, sparse models. In Marina Meila and Tong Zhang, editors, *Proc. Int. Conf. on Machine Learning (ICML)*, volume 139, pages 6265–6274, Virtual only, July 2021.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, New Orleans, LA, USA, December 2022.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *Int. Conf. on Learning Representations (ICLR)*, Toulon, France, April 2017.

Margaret Li, Suchin Gururangan, Tim Dettmers, Mike Lewis, Tim Althoff, Noah A Smith, and Luke Zettlemoyer. Branch-train-merge: Embarrassingly parallel training of expert language models. *Preprint arXiv:2208.03306*, 2022a.

Yuanpeng Li, Liang Zhao, Jianyu Wang, and Joel Hestness. Compositional generalization for primitive substitutions. In *Proc. Conf. on Empirical Methods in Natural Language Processing and Int.Joint Conf. on Natural Language Processing (EMNLP-IJCNLP)*, pages 4292–4301, Hong Kong, China, November 2019.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Preprint arXiv:2203.07814*, March 2022b.

Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J. Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, and Sanjiv Kumar. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. In *Int. Conf. on Learning Representations (ICLR)*, Kigali, Rwanda, May 2023.

Seppo Linnainmaa. *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*. PhD thesis, Master's Thesis, Univ. Helsinki, 1970.

Adam Liska, Germán Kruszewski, and Marco Baroni. Memorize or generalize? Searching for a compositional RNN in a haystack. In *AEGAP Workshop ICML*, Stockholm, Sweden, July 2018.

Chenyao Liu, Shengnan An, Zeqi Lin, Qian Liu, Bei Chen, Jian-Guang Lou, Lijie Wen, Nanning Zheng, and Dongmei Zhang. Learning algebraic recombination for compositional generalization. In *Proc. of Annual Meeting of the Association for Computational Linguistics and the International Joint Conf. on Natural Language Processing, ACL-IJCNLP*, pages 1129–1144, Virtual Only, August 2021.

Dianbo Liu, Alex Lamb, Xu Ji, Pascal Notsawo, Mike Mozer, Yoshua Bengio, and Kenji Kawaguchi. Adaptive discrete communication bottlenecks with dynamic vector quantization. *Preprint arXiv:2202.01334*, 2022.

Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. Evaluating the logical reasoning ability of chatgpt and GPT-4. *Preprint arXiv:2304.03439*, 2023.

Qian Liu, Shengnan An, Jian-Guang Lou, Bei Chen, Zeqi Lin, Yan Gao, Bin Zhou, Nanning Zheng, and Dongmei Zhang. Compositional generalization by learning analytical expressions. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

João Loula, Marco Baroni, and Brenden M Lake. Rearranging the familiar: Testing compositional generalization in recurrent networks. In *BlackboxNLP@ EMNLP*, pages 108–114, Brussels, Belgium, November 2018.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *Int. Conf. on Learning Representations (ICLR)*, Toulon, France, April 2017.

Arun Mallya and Svetlana Lazebnik. PackNet: Adding multiple tasks to a single network by iterative pruning. In *Proc. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7765–7773, Salt Lake City, USA, "June" 2018.

Arun Mallya, Dillon Davis, and Svetlana Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proc. European Conference on Computer Vision (ECCV)*, volume 11208, pages 72–88, Munich, Germany, September 2018.

Christoph Von Der Malsburg. The correlation theory of brain function. Technical Report 81-2, Dept. of Neurobiology, MaxPlanck-Institute for Biophysical Chemistry, Göttingen, 1981.

Gary F Marcus. Rethinking eliminative connectionism. *Cognitive psychology*, pages 243–282, 1998.

Gary F Marcus. *The algebraic mind: Integrating connectionism and cognitive science*. MIT press, 2003.

Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.

Brian P. McLaughlin. Systematicity redux. *Synthese*, 170(2):251–274, 2009.

Alexander H. Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. Key-value memory networks for directly reading documents. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1400–1409, Austin, Texas, USA, November 2016.

Jovana Mitrovic, Brian McWilliams, Jacob C. Walker, Lars Holger Buesing, and Charles Blundell. Representation learning via invariant causal mechanisms. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Sarthak Mittal, Alex Lamb, Anirudh Goyal, Vikram Voleti, Murray Shanahan, Guillaume Lajoie, Michael Mozer, and Yoshua Bengio. Learning to combine top-down and bottom-up signals in recurrent neural networks with attention over modules. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 119, pages 6972–6986, July 2020.

Sarthak Mittal, Sharath Chandra Raparthy, Irina Rish, Yoshua Bengio, and Guillaume Lajoie. Compositional attention: Disentangling search and retrieval. *Preprint arXiv:2110.09419*, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, Stateline, NV, USA, December 2013.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proc. Int. Conf. on Machine Learning (ICML)*, pages 807–814, Haifa, Israel, June 2010.

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. In *Int. Conf. on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, April 2019.

Nikita Nangia and Samuel R. Bowman. ListOps: A diagnostic dataset for latent tree learning. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pages 92–99, New Orleans, USA, June 2018.

Allen Newell and Herbert Alexander Simon. GPS, a program that simulates human thought. 1961.

Allen Newell, John C Shaw, and Herbert A Simon. Report on a general problem solving program. In *IFIP congress*, volume 256, page 64, 1959.

Benjamin Newman, John Hewitt, Percy Liang, and Christopher D Manning. The EOS decision and length extrapolation. In *Proc. BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, EMNLP*, pages 276–291, Virtual only, 2020.

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks. In *ICLR 2021 Mathematical Reasoning in General Artificial Intelligence Workshop*, Virtual only, May 2021.

Nostalgebraist. Chinchilla's wild implications. `https://www.lesswrong.com/posts/6Fpvch8RR29qLEWNH/chinchilla-s-wild-implications`, 2022.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. In *ICLR 2022 DL4C Workshop*, August 2022.

Maxwell I Nye, Armando Solar-Lezama, Joshua B Tenenbaum, and Brenden M Lake. Learning compositional rules via neural program synthesis. *Preprint arXiv:2003.05562*, December 2020.

Santiago Ontañón, Joshua Ainslie, Vaclav Cvicek, and Zachary Fisher. Making Transformers solve compositional tasks. *Preprint arXiv:2108.04378*, 2021.

OpenAI. Chatgpt. `https://openai.com/blog/chatgpt`, 2022.

OpenAI. GPT-4 technical report. *Preprint arXiv:2303.08774*, 2023.

Inbar Oren, Jonathan Herzig, Nitish Gupta, Matt Gardner, and Jonathan Berant. Improving compositional generalization in semantic parsing. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, Virtual only, November 2020.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, New Orleans, LA, USA, December 2022.

Peter Pagin and Dag Westerståhl. Compositionality I: Definitions and variants. *Philosophy Compass*, 5(3):250–264, 2010.

Ankur P. Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2249–2255, Austin, TX, USA, November 2016.

Emilio Parisotto, H. Francis Song, Jack W. Rae, Razvan Pascanu, Çaglar Gülçehre, Siddhant M. Jayakumar, Max Jaderberg, Raphaël Lopez Kaufman, Aidan Clark, Seb Noury, Matthew Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing Transformers for reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 119, pages 7487–7498, Virtual only, July 2020.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, Vancouver, Canada, December 2019.

Judea Pearl. *Causality*. Cambridge university press, 2009.

Judea Pearl and Dana Mackenzie. *The Book of Why: The New Science of Cause and Effect*. Basic Books, Inc., 2018.

Gordon Plotkin. Automatic methods of inductive inference. 1972.

Jordan Bruce Pollack. On connectionist models of natural language processing. *PhD dissertation. University of Illinois*, 1987.

Senthil Purushwalkam, Maximilian Nickel, Abhinav Gupta, and Marc'Aurelio Ranzato. Task-driven modular networks for zero-shot compositional learning. In *Proc. IEEE Int. Conf. on Computer Vision (ICCV)*, pages 3592–3601, Seoul, South Korea, November 2019.

Zenon Walter Pylyshyn. Computation and cognition. 1984.

Linlu Qiu, Peter Shaw, Panupong Pasupat, Paweł Krzysztof Nowak, Tal Linzen, Fei Sha, and Kristina Toutanova. Improving compositional generalization with latent structure and data augmentation. *Preprint arXiv:2112.07610*, 2021.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Jack W. Rae, Jonathan J. Hunt, Ivo Danihelka, Timothy Harley, Andrew W. Senior, Gregory Wayne, Alex Graves, and Tim Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 3621–3629, Barcelona, Spain, December 2016.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew J. Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training Gopher. *Preprint arXiv:2112.11446*, 2021.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William S. Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *Preprint arXiv:2112.11446*, January 2022.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research (JMLR)*, 21:140:1–140:67, 2020.

Mark Ring. Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies. In *Machine Learning Proceedings*, pages 343–347, San Francisco, USA, 1991. ISBN 978-1-55860-200-7.

Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5): 465–471, 1978.

Rebecca Roelofs. *Measuring Generalization and overfitting in Machine learning*. PhD thesis, UC Berkeley, 2019.

Fabio Roli, Sebastiano B. Serpico, and Gianni Vernazza. Image recognition by integration of connectionist and symbolic approaches. *Int. Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 9(3):485–515, 1995.

Clemens Rosenbaum, Ignacio Cases, Matthew Riemer, and Tim Klinger. Routing networks and the challenges of modular and compositional computation. *Preprint arXiv:1904.12774*, 2019.

Laura Ruis and Brenden Lake. Improving systematic generalization through modularity and augmentation. *Preprint arXiv:2202.10745*, 2022.

Luana Ruiz, Joshua Ainslie, and Santiago Ontañón. Iterative decoding for compositional generalization in transformers. *Preprint arXiv:2110.04169*, 2021.

Jake Russin, Jason Jo, Randall C O'Reilly, and Yoshua Bengio. Compositional generalization in a deep seq2seq model by separating syntax and semantics. *Preprint arXiv:1904.09708*, 2019.

Adam Santoro, David Raposo, David G. T. Barrett, Mateusz Malinowski, Razvan Pascanu, Peter W. Battaglia, and Tim Lillicrap. A simple neural network module for relational reasoning. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, Long Beach, CA, USA, December 2017.

Laurent Sartran, Samuel Barrett, Adhiguna Kuncoro, Miloš Stanojević, Phil Blunsom, and Chris Dyer. Transformer grammars: Augmenting transformer language models with syntactic inductive biases at scale. *Preprint arXiv:2203.00633*, 2022.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

Imanol Schlag and Jürgen Schmidhuber. Learning to reason with third order tensor products. *Advances in neural information processing systems*, 31, 2018.

Imanol Schlag, Paul Smolensky, Roland Fernandez, Nebojsa Jojic, Jürgen Schmidhuber, and Jianfeng Gao. Enhancing the transformer with explicit relational encoding for math problem solving. *Preprint arXiv:1910.06611*, 2019.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 139, pages 9355–9366, Virtual only, 2021.

Jeffrey C. Schlimmer and Douglas H. Fisher. A case study of incremental concept induction. In *Proceedings of the 5th National Conference on Artificial Intelligence*, pages 496–501, Philadelphia, USA, August 1986.

J. Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Diploma thesis, Inst. f. Inf., Tech. Univ. Munich, 1987.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. Technical Report FKI-147-91, Institut für Informatik, Technische Universität München, March 1991.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. *Neural Computation*, 4(1):131–139, 1992a.

Jürgen Schmidhuber. Learning factorial codes by predictability minimization. *Neural computation*, 4(6):863–879, 1992b.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992c.

Jürgen Schmidhuber. On decreasing the ratio between learning complexity and number of time-varying variables in fully recurrent nets. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 460–463. Springer, 1993.

Jürgen Schmidhuber. Self-delimiting neural networks. *Preprint arXiv:1210.0118*, 2012.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

Jürgen Schmidhuber. Annotated history of modern AI and deep learning. *Preprint arXiv:2212.11279*, December 2022.

Jürgen Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992d.

Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54: 211–254, 2004.

Bernhard Schölkopf. Causality for machine learning. *Preprint arXiv:1911.10500*, December 2019.

Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2021.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proc. Association for Computational Linguistics (ACL)*, pages 1715–1725, Berlin, Germany, August 2016.

Ehud Y. Shapiro. The model inference system. In Patrick J. Hayes, editor, *International Joint Conference on Artificial Intelligence, IJCAI*, page 1064, Vancouver, BC, Canada, August 1981. William Kaufmann.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pages 464–468, New Orleans, Louisiana, USA, June 2018a.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pages 464–468, New Orleans, Louisiana, USA, June 2018b.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? In *Proc. Association for Computational Linguistics (ACL)*, Virtual only, August 2021.

Noam Shazeer. GLU variants improve transformer. *Preprint arXiv:2002.05202*, 2020.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Int. Conf. on Learning Representations (ICLR)*, Toulon, France, April 2017.

Kai Shen, Junliang Guo, Xu Tan, Siliang Tang, Rui Wang, and Jiang Bian. A study on ReLU and softmax in transformer. *Preprint arXiv:2302.06461*, 2023.

Yikang Shen, Shawn Tan, Seyed Arian Hosseini, Zhouhan Lin, Alessandro Sordoni, and Aaron C. Courville. Ordered memory. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 5038–5049, Vancouver, Canada, December 2019.

Edward H Shortliffe and Bruce G Buchanan. A model of inexact reasoning in medicine. *Mathematical biosciences*, 23(3-4):351–379, 1975.

Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 70, pages 3145–3153, Sydney, Australia, August 2017.

Hava T Siegelmann and Eduardo D Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016a.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016b.

Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *Preprint arXiv:1312.6034*, 2013.

Richard Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. *The annals of mathematical statistics*, 35(2): 876–879, 1964.

Richard Sinkhorn and Paul Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.

Ray Solomonoff. A formal theory of inductive inference. part I. *Information and Control*, 7(1):1–22, 1964a.

Ray Solomonoff. A formal theory of inductive inference. part II. *Information and Control*, 7(2):224–254, 1964b.

Alessandro Sperduti. Encoding labeled graphs by labeling RAAM. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 1125–1132, Denver, Colorado, USA, 1993.

Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. In *Int. Conf. on Learning Representations (ICLR), Workshop*, San Diego, CA, USA, May 2015.

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.

Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 2368–2376, Montreal, Canada, December 2015a.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *ICML Deep Learnig Workshop*, July 2015b.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. Weakly supervised memory networks. *Preprint arXiv:1503.08895*, 2015.

Ron Sun and Lawrence A Bookman. *Computational architectures integrating neural and symbolic processes: A perspective on the state of the art*. Springer Science & Business Media, 1994.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 70, pages 3319–3328, Sydney, Australia, 2017.

Gerald J Sussman. A computational model of skill acquisition. 1973.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 3104–3112, Montréal, Canada, December 2014.

Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov, and Stuart M. Shieber. Memory-augmented recurrent neural networks can learn generalized dyck languages. *Preprint arXiv:1911.03329*, 2019.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Int. Conf. on Learning Representations (ICLR)*, April 2014.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long Range Arena : A benchmark for efficient transformers. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. Is ChatGPT the ultimate programming assistant - how far is it? *Preprint arXiv:2304.11938*, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. *Preprint arXiv:2302.13971*, 2023.

Geofrey G Towell, Jude W Shavlik, Michiel O Noordewier, et al. Refinement of approximate domain theories by knowledge-based neural networks. In *Proc. AAAI Conf. on Artificial Intelligence*, volume 861866, pages 861–866, Boston, Massachusetts, USA, 1990.

Ankit Vani, Max Schwarzer, Yuchen Lu, Eeshan Dhekane, and Aaron C. Courville. Iterated learning for emergent systematicity in VQA. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Vladimir Vapnik. *Statistical learning theory*. Wiley, April 1998.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008, Long Beach, CA, USA, December 2017.

Petar Velickovic and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7), 2021.

Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, April 2020.

Chr. von der Malsburg. Self-organization of orientation sensitive cells in the striate cortex. *Kybernetik*, 14:85–100, 1973.

Bailin Wang, Mirella Lapata, and Ivan Titov. Structured reordering for modeling latent alignments in sequence transduction. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, volume 34, Virtual only, 2021.

Peter C Wason. Reasoning about a rule. *Quarterly journal of experimental psychology*, 20(3):273–281, 1968.

Chihiro Watanabe. Interpreting layered neural networks via hierarchical modular representation. In *International Conference on Neural Information Processing*, pages 376–388, 2019.

Chihiro Watanabe, Kaoru Hiramatsu, and Kunio Kashino. Modular representation of layered neural networks. *Neural Networks*, 97:62–73, 2018.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *Preprint arXiv:2201.11903*, 2022.

Gail Weiss. `https://twitter.com/gail_w/status/1600646811508772866`, 2022.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like Transformers. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 11080–11090, Virtual only, July 2021.

Pia Weißenhorn, Yuekun Yao, Lucia Donatelli, and Alexander Koller. Compositional generalization requires compositional parsers. *Preprint arXiv:2202.11937*, 2022.

Joseph Weizenbaum. ELIZA—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

Paul Werbos. Applications of advances in nonlinear sensitivity analysis. *System Modeling and Optimization*, pages 762–770, 1982.

Jason Weston, Antoine Bordes, Sumit Chopra, and Tomás Mikolov. Towards AI-complete question answering: A set of prerequisite toy tasks. In Yoshua Bengio and Yann LeCun, editors, *Int. Conf. on Learning Representations (ICLR)*, San Juan, Puerto Rico, May 2016.

Alfred North Whitehead. Symbolism: Its meaning and effect. *Journal of Philosophical Studies*, 3(12), 1928.

Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2): 270–280, 1989.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. *Preprint arXiv:2307.02477*, 2023.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 119, pages 10524–10533, Virtual Only, July 2020.

Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. Nyströmformer: A Nyström-based algorithm for approximating self-attention. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 14138–14148, Virtual only, February 2021.

Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural execution engines: Learning to execute subroutines. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, December 2020.

Ruihan Yang, Huazhe Xu, Yi Wu, and Xiaolong Wang. Multi-task reinforcement learning with soft modularization. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual Only, December 2020.

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8BERT: quantized 8bit BERT. In *Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS*, Vancouver, Canada, December 2019.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. In *Int. Conf. on Learning Representations (ICLR)*, San Diego, USA, May 2015.

Biao Zhang, Ivan Titov, and Rico Sennrich. Improving deep transformer with depth-scaled initialization and merged attention. In *Proc. Conf. on Empirical Methods in Natural Language Processing and Int.Joint Conf. on Natural Language Processing (EMNLP-IJCNLP)*, pages 898–909, Hong Kong, China, November 2019.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Michael C. Mozer, and Yoram Singer. Identity crisis: Memorization and generalization under extreme overparameterization. In *Int. Conf. on Learning Representations (ICLR)*, Virtual Only, April 2020.

Dinghuai Zhang, Kartik Ahuja, Yilun Xu, Yisen Wang, and Aaron C. Courville. Can subnetwork structure be the key to out-of-distribution generalization? In *Proc. Int. Conf. on Machine Learning (ICML)*, Virtual only, July 2021.

Hao Zheng and Mirella Lapata. Disentangled sequence to sequence learning for compositional generalization. In *Proc. Association for Computational Linguistics (ACL)*, Dublin, Ireland, May 2022.

Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *Proc. Advances in Neural*

Information Processing Systems (NeurIPS), pages 3592–3602, Vancouver, BC, Canada, 2019.

Chen Zhu, Renkun Ni, Zheng Xu, Kezhi Kong, W Ronny Huang, and Tom Goldstein. Gradinit: Learning to initialize neural networks for stable and efficient training. *Preprint arXiv:2102.08098*, 2021.