ABSTRACT

Title of Dissertation: DEEP THINKING SYSTEMS: LOGICAL EXTRAPOLATION WITH RECURRENT NEURAL NETWORKS

Avi Schwarzschild
Doctor of Philosophy, 2023

Dissertation Directed by: Tom Goldstein
Department of Computer Science

Deep neural networks are powerful machines for visual pattern recognition, but reasoning tasks that are easy for humans are still be difficult for neural models. Humans possess the ability to extrapolate reasoning strategies learned on simple problems to solve harder examples, often by thinking for longer. We study neural networks that have exactly this capability. By employing recurrence, we build neural networks that can expend more computation when needed. Using several datasets designed specifically for studying generalization from easy problems to harder test samples, we show that our recurrent networks can extrapolate from easy training data to much harder examples at test time, and they do so with many more iterations of a recurrent block of layers than are used during training.

DEEP THINKING SYSTEMS: LOGICAL EXTRAPOLATION WITH RECURRENT NEURAL NETWORKS

by

Avi Schwarzschild

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor Tom Goldstein, Chair/Advisor
Professor John Dickerson
Professor Furong Huang
Professor Howard Elman
Professor Dan Butts

© Copyright by
Avi Koplon Schwarzschild
2023

Acknowledgments

The work presented in my dissertation is the product of collaborations with the wonderful people I had the pleasure to work with at the University of Maryland. First, thank you to my advisor, Professor Tom Goldstein, whose crazy ideas and invaluable insights steered my research throughout my PhD. In addition to introducing me to deep learning – first through his course and then through research discussions – Tom showed me the way an advisor can care for their students and foster creative and productive collaboration.

The students, post doctoral researchers, and other professors that I am fortunate to have worked closely with include John Dickerson, Howard Elman, Micah Goldblum, Jonas Geiping, Liam Fowl, Arpit Bansal, Eitan Borgnia, Zeyad Emam, Arjun Gupta, Alex Stein, Michael Curry, Gowthami Somepalli, Valeriia Cherapenova, Roman Levin, and many others. Micah's continued advice as a collaborator and a friend has helped me at every step of the way, starting with the first project I worked on at Maryland. I am also grateful to John for his valuable input on research projects as well as his general guidance in making decisions about my future. To Howard, thank you for helping me navigate life as a graduate student throughout my time at Maryland.

Outside of school, my family's support has made everything easier. My older sister Gila's enthusiasm, especially around my work, brings so much positive energy and makes sharing things with her so much fun. My parents have supported and encouraged me through all 23 years of schooling and helped keep the important things in perspective. My Dad's innate curiosity and

scientific approach to life is a persistent model for how important research is in all aspects of my life. My Mom, who is my favorite high school math teacher, is devoted to academia and to education – in part through her own PhD in applied math – and similarly serves as an ideal I aspire to.

To Jesse, Scott, Jill, Joey, Rachel, Deena, and Hannah Lou, thank you for all the support and for bringing the fun and joy that helped make my time in graduate school so great. And to my friends who constantly make life more enjoyable, thank you.

Finally, without my fiancée Samantha's unwavering support, the last few years would not have been as easy, as fun, or as good. Her never-ending belief in me and her boundless encouragement give me energy at every turn. I am so grateful for having Samantha in my corner for every paper deadline and every major life decision.

Table of Contents

Acknowledgements ii

Table of Contents iv

List of Tables vi

List of Figures viii

Chapter 1: Introduction 1

Chapter 2: Preliminaries 4
- 2.1 Deep Learning Background . . . 4
  - 2.1.1 Neural Networks . . . 4
  - 2.1.2 Training Neural Networks . . . 5
  - 2.1.3 Recurrence in Neural Networks . . . 6
- 2.2 Datasets for Studying Extrapolation from Easy to Hard . . . 8
  - 2.2.1 Prefix Sums . . . 9
  - 2.2.2 Mazes . . . 10
  - 2.2.3 Chess Puzzles . . . 12
  - 2.2.4 Python Package . . . 16

Chapter 3: The Uncanny Similarity of Recurrence and Depth 18
- 3.1 Introduction . . . 18
  - 3.1.1 Related Work . . . 20
- 3.2 Recurrent Architectures . . . 22
  - 3.2.1 Multi-layer Perceptrons . . . 24
  - 3.2.2 Convolutional Networks . . . 25
  - 3.2.3 Residual Networks . . . 26
- 3.3 Matching Image Classification Accuracies . . . 27
  - 3.3.1 Separate Batch Normalization Statistics for Each Recurrence . . . 29
- 3.4 The Maze Problem . . . 30
  - 3.4.1 Maze Solving Networks . . . 31
  - 3.4.2 Solving Mazes Requires Depth (or Recurrence) . . . 31
- 3.5 Recurrent Models Reuse Features . . . 33
  - 3.5.1 Recurrent and Feed-forward Models Similarly Disentangle Classes . . . 35
  - 3.5.2 Visualizing the Roles of Recurrent and Feed-forward Layers . . . 37
- 3.6 Discussion . . . 40

Chapter 4: Can You Learn an Algorithm? 44
4.1 Introduction . . . 44
4.1.1 Related Work . . . 46
4.2 Model Architectures and Training . . . 48
4.3 Recurrent Networks Can Generalize from Easy to Hard Problems . . . 51
4.3.1 Prefix Sums . . . 51
4.3.2 Mazes . . . 54
4.3.3 Chess Puzzles . . . 57
4.4 Further Experimentation . . . 60
4.4.1 Dilated Filters . . . 60
4.4.2 Deeper Feed-forward Models . . . 60
4.4.3 In-distribution Tests . . . 61
4.4.4 Prefix Sum Experiments on Other Datasets . . . 61
4.4.5 Even Harder Chess Puzzles . . . 61
4.5 More Visualizations . . . 61
4.5.1 Prefix Sums . . . 63
4.5.2 Mazes . . . 64
4.5.3 Chess Puzzles . . . 64
4.6 Discussion . . . 64
4.7 Conclusion . . . 67

Chapter 5: Logical Extrapolation Without Overthinking 69
5.1 Introduction . . . 70
5.2 Related Work . . . 72
5.3 Methods . . . 75
5.3.1 Recall Architectures . . . 76
5.3.2 Promoting Forward Progress Through Optimization . . . 78
5.3.3 Datasets . . . 80
5.4 Extreme Extrapolation . . . 82
5.5 The Overthinking Problem . . . 87
5.5.1 Manipulating Feature Maps . . . 89
5.5.2 Manipulating Inputs . . . 89
5.5.3 Converging to a Fixed Point . . . 91
5.5.4 More on the Overthinking Problem . . . 91
5.6 Further Insights . . . 94
5.6.1 In-distribution Results . . . 94
5.6.2 Hard to Easy . . . 95
5.6.3 Hyperparameter Selection . . . 95
5.7 Conclusion . . . 98

Chapter 6: Ongoing and Future Directions 100
6.1 Sudoku Puzzles . . . 100
6.2 Reinforcement Learning . . . 102

Bibliography 104

List of Tables

3.1 MLPs: The size of each fully connected layer in the MLPs we use for image classification. . . . 25
3.2 ConvNets: The number of channels in the output of each layer. . . . 26
3.3 **Models with batch normalization.** Performance shown for three different classification datasets. We report averages (± one standard error) from three trials. . . 30
3.4 **Dilated Filters.** The average accuracy of models with dilated filters trained and tested on small mazes. . . . 33
3.5 Classifiability of features. CIFAR-10 classification accuracy of linear classifiers trained on intermediate features output by each residual block (or each iteration) in networks with 19-layer effective depths. Each cell reports the percentage of training/testing images correctly labeled. . . . 36
3.6 Classifiability of features in ConvNets and MLPs. CIFAR-10 classification accuracy of linear classifiers trained on intermediate features output by each iteration. Each cell reports the percentage of training/testing images correctly labeled. . . . 37

4.1 **Extrapolating to longer input strings.** Shown here are the average accuracies of models trained on 32-bit inputs and tested on 40-bit inputs. The effective depths listed below correspond to 9, 10, and 11 iterations in recurrent models. We report average accuracy ± one standard error. . . . 52
4.2 **The average accuracy (%) of models trained on small mazes and tested on large ones.** Over a range of effective depths, we see that recurrent models generalize to the harder mazes better than their feed-forward counterparts. Figures reflect averages over several trials ± one standard error. . . . 55
4.3 **The average accuracy (%) of models with dilated filters trained on small mazes and tested on large ones.** Figures reflect averages over several trials ± one standard deviation. . . . 56

5.1 Model architectures for all main experiments. Note, we perform ablations where we change the width or the maximum number of iterations and those parameters are indicated where appropriate. . . . 78
5.2 The peak accuracy and corresponding test-time iteration number for prefix sum solving model performance curves in Figures 5.7 and 5.3. . . . 86
5.3 The peak accuracy and corresponding test-time iteration number for maze solving model performance curves in Figures 5.7 and 5.4. . . . 86
5.4 Peak accuracy and iteration number for chess puzzle performance curves in Figure 5.5. . . . 87

5.5 Average iterations to solution, with the peak accuracy in parentheses. We evaluate maze solving models on 13×13 mazes, where we intervene in the solving process in different ways. 93
5.6 Training accuracy and in-distribution validation accuracy for models presented in Figures 5.3 and 5.7. 96
5.7 Training hyperparameters. Dashes indicate that we did not utiltize those options. LR denotes learning rate, Decay column lists the epochs after which the learning rate is multiplied by the decay factor. 97

List of Figures

2.1 *Prefix Sums* input samples and their corresponding targets/labels. We provide multiple sets, each containing problems of a different length, and intend for users to train on shorter strings and test on longer ones. Examples from the sets of length 16 and 28 are shown above. 10

2.2 The *Mazes* generation process for making a 5×5 maze. We start with a 3×3 grid graph. Each side of the grid graph contains 3 nodes and 2 edges ($3 + 2 = 5$ total elements). We then produce a spanning tree for the graph using a randomized algorithm. The tree encodes the allowed and forbidden paths. This tree is then represented as a 5×5 array of maze cells. Finally, we convert this to an image by representing each cell as a 2×2 array of pixels, and adding a 3-pixel border on each side. This creates an image representation that has $5 \times 2 + 3 + 3 = 16$ pixels on each side. The green and red start and end cells are chosen at random. . 11

2.3 Samples from *Mazes*. Mazes (top) and their labels/solutions (bottom) of size 9×9, 13×13, and 21×21. 12

2.4 Fun for the whole family! We leave solving this 201×201 maze as an exercise for the reader. The green starting dot is near the center of the upper right quadrant. The red ending point is on the second row from the bottom. 13

2.5 An example of a *very, very* fun 801×801 maze. The green starting dot is three quarters of the way down on the left-hand side. A model trained using only 9×9 mazes and 30 iterations is able to solve this extreme maze at test time using 10,000 iterations, which is equivalent to 100,004 convolutional layers. 14

2.6 Samples from *Chess Puzzles*. "Easy" chess puzzles, each with their solution below. The board state is represented as a $12 \times 8 \times 8$ tensor, where the first 6 maps encode the pieces belonging to the player to act next. The white player is acting in the two leftmost puzzles, while the black player acts in the puzzle on the right. The solution is represented as a $1 \times 8 \times 8$ tensor that marks the start and end position of the piece to be moved. 16

3.1 Effective depth measures the depth of the unrolled network. For example, a network with 3 iterations of its single-layer recurrent module and 4 additional layers has an effective depth of 7. 24

3.2 Average accuracy of each model. The x-axis represents effective depth and the y-axis measures test accuracy. The marker shapes indicate architectures, and the blue and orange markers indicate feed-forward and recurrent models, respectively. The horizontal bars are $\pm$ one standard error from the averages over several trials. Across all three families of neural networks, recurrence can imitate depth. Note, the y-axes are scaled per dataset to best show each of the trends. 28

3.3 Recurrent models perform similarly to feed-forward models of the same effective depth. All models here are residual models as described in Section 3.4. 31
3.4 Relative activations of the recurrent layer in an untrained CNN. This plot confirms that the distributions in the other similar plots from trained models are significant. 34
3.5 Relative activations of the recurrent layer of a CNN trained on SVHN. 34
3.6 Relative activations at each of the four layers in the recurrent module of a residual network (with a four-layer residual block as the recurrent module) trained on CIFAR-10. The top row is from the first and second layers and the bottom row shows the third and fourth layers. 35
3.7 The number of active neurons after each iteration of a recurrent ConvNet with 5 iterations. Note that the most active iteration is left off this plot since we are normalizing by the number activations at that iteration. 36
3.8 ImageNet filter visualization. Filters from recurrent model iterations alongside corresponding feed-forward layers. 38
3.9 CIFAR-10 filter visualization. Filters from recurrent model iterations and corresponding feed-forward layers. 39
3.10 ImageNet filter visualization. Filters from recurrent model iterations alongside corresponding feed-forward layers. 41
3.11 CIFAR-10 filter visualization. Filters from recurrent model iterations alongside corresponding feed-forward layers. 43

4.1 Generalizing from easy to hard prefix sums. The ability of networks to compute prefix sums on two test sets with longer input strings than were used for training (accuracy on 40-bit inputs in purple and on 44-bit inputs in red). We compare recurrent models to the best feed-forward models of comparable effective depth. The markers are at average values from several trials and the shaded regions indicate $\pm$ one standard error. 52
4.2 A recurrent model's output from each of 11 iterations on a 40-bit input string. Shown here is the confidence that there is a 1 at each index of the output. The first index is at the top for all vectors, the input is in the left-most column and the target is in the right-most column. The model used to produce this plot was trained with fewer iterations (10) on shorter input strings (32-bit). 54
4.3 Generalizing from easy to hard mazes. We compare recurrent models to the best feed-forward models. The markers are at average values from several trials and the shaded regions indicate $\pm$ one standard error. 56
4.4 Input, target, and outputs from different iterations are shown to highlight the model's ability to think sequentially about mazes. We plot the model's confidence that each pixel belongs to the optimal path. This is a representative example from a model trained to solve small mazes in six iterations. 57
4.5 Generalizing from easy to hard chess puzzles. The ability of networks to solve harder puzzles than were used for training. We compare recurrent models to the best feed-forward models of comparable effective depth. The markers are at average values from several trials and the shaded region indicate $\pm$ one standard error. 58

4.6 Input, target, and outputs form different iterations are shown to highlight the model's ability to think about the next move. We plot the model's confidence that each pixel is one of the two that define a move. In this example, black is to move next. For space consideration, iterations 2-14, which look like the first iteration, are left out of this plot. 59
4.7 Generalizing from easy to hard prefix sums. The ability of networks to compute prefix sums on two additional test sets with longer input strings than were used for training (accuracy on 36-bit inputs in yellow and on 48-bit inputs in pink). We compare recurrent models to the best feed-forward models of comparable effective depth. 62
4.8 Generalizing from easy to hard chess puzzles. The ability of networks to solve harder puzzles than were used for training. We trained models on the first 600,000 puzzles and we show their performance with extra iterations on puzzles with index 800,000 to 850,000 (top left), 850,000 to 900,000 (top right), 900,000 to 950,000 (bottom left), and 100,000 to 150,000 (bottom right). 62
4.9 Prefix sum intermediate outputs. 63
4.10 Maze model intermediate outputs. 65
4.11 Chess model intermediate outputs. 66

5.1 A 'thinking' network trained on 9×9 mazes and their solutions (left) autonomously synthesizes a scalable algorithm. By running this algorithm for longer, it reliably solves problems of size 59×59 (middle), and 201×201 (right) without retraining. Standard architectures, and even existing primitive thinking models, as described in Chapter 4, fail to tolerate this domain shift. 69
5.2 Architecture schematics. Left to right: A feed-forward network, a network containing three recurrent blocks (in green) that share weights, and a recurrent network with recall. 75
5.3 **Prefix sum models trained on 32-bit inputs extrapolate to 512-bit data.** The value of our recall and progressive loss is clear by how quickly and accurately our models solve this very large problem. 83
5.4 **Maze solving models trained on 9×9 inputs extrapolate to 59×59 problems.** Mazes this large cannot be solved without recall, and furthermore progressive loss leads to more accurate models. 83
5.5 **Chess models trained on the first 600K easiest puzzles extrapolate to 600K-700K.** Recall and progressive loss are required to retain accuracy with many iterations. 85
5.6 Chess performance when tested on puzzles with indices from 700k to 800k (above) and from 1M to 1.1M (below). 87
5.7 **Left:** Prefix sum models trained on 32-bits tested on 48-bit inputs. **Right:** Maze models trained on 9×9 mazes tested on 13×13 mazes. In these regimes, we can see that the weaker models can extrapolate to slightly harder problems, but importantly only models with recall avoid the overthinking trap. 88

5.8 **Left:** How long it takes prefix sum models to recover from perturbation – recall keeps this quantity small. **Center:** Test accuracy on 13×13 mazes when features are swapped after 50 iterations. **Right:** The change in the features when solving 13×13 mazes. 90
5.9 Maze solving model performance when Gaussian noise is added to the features after the first iteration. DT models with recall are robust to this perturbation. . . . 92
5.10 Maze solving model performance when the features after 50 iterations are replaced with zeros. DT models with recall are also robust to this perturbation. . . . 92
5.11 Maze solving model performance when the start and end of the maze are changed to be closer together after 50 iterations. DT models with recall can self-correct. . 93
5.12 The change in the norm of the feature maps for maze solvers as function of iteration shows that models with recall seem to move the feature maps less and less with each iteration. 93
5.13 Prefix sum model performance when Gaussian noise is added to the features after the first iteration. DT models with recall are robust to this perturbation. 94
5.14 Prefix sum model performance when the features after the first iteration are replaced with zeros. DT models with recall are also robust to this perturbation. . . . 94
5.15 The changes in the norm of the feature maps for prefix sums as function of iteration show that models with recall seem to move the feature maps less and less with each iteration. 95
5.16 Accuracy as a function of iteration for prefix sum models when generalizing from harder 32-bit strings to easier 24-bit strings. 97
5.17 **Left:** Accuracy on 512-bit strings for different values of m show that $m = 30$ is sufficient. **Right:** Accuracy on 512-bit strings for different values of w. The gain plateaus on prefix sum models at $w = 400$. Models presented here are trained with $\alpha = 1$ and recall. 97

6.1 An example Sudoku problem from the test set. Top left: the original puzzle showing the given information. Top right: the output of a DT-Recall network after one iteration, colored according to correctness and showing the location of the given information. Bottom: the solution after four iterations and the complete solution after 33 iterations. 101

Chapter 1: Introduction

Machine learning systems perform well on pattern matching tasks, but their ability to perform algorithmic or logical reasoning is not well understood. One important reasoning capability is *logical extrapolation*, in which models trained only on small or simple reasoning problems can synthesize complex algorithms that scale up to large or complex problems at test time. Logical extrapolation can be achieved through recurrent systems, which can be iterated many times to solve difficult reasoning problems.

This dissertation includes the description of three datasets that we compiled, packaged, and released for public use as part of the experimental work. These details, as well as preliminary and background information, are presented in Chapter 2.

In Chapter 3, we study recurrence in neural networks. As the primary mechanism we use for logical extrapolation, we first seek to understand when and how recurrence can factor into familiar deep learning pipelines. This inquiry starts by challenging the widely held belief that deep neural networks contain layer specialization, wherein neural networks extract hierarchical features representing edges and patterns in shallow layers and complete objects in deeper layers. Unlike common feed-forward models that have distinct filters at each layer, recurrent networks reuse the same parameters at various depths. We observe that recurrent models exhibit the same hierarchical behaviors and the same performance benefits with depth as feed-forward networks

despite reusing the same filters at every recurrence. By training models of various feed-forward and recurrent architectures on several datasets for image classification as well as maze solving, we show that recurrent networks have the ability to closely emulate the behavior of non-recurrent deep models, often doing so with far fewer parameters.

With an understanding that recurrence can provide neural networks with similar computational power as feed-forward layers and the knowledge that recurrent systems can be made deeper at test time, we apply these networks to algorithmic problems in Chapter 4. We are motivated by humans' ability to extrapolate reasoning strategies learned on simple problems to solve harder examples, often by thinking for longer. For example, a person who has learned to solve small mazes can easily extend the very same search techniques to solve much larger mazes by spending more time working toward the solution. In computers, this behavior is often achieved through the use of algorithms, which scale to arbitrarily hard problem instances at the cost of more computation. In contrast, the sequential computing budget of feed-forward neural networks is limited by their depth, and networks trained on simple problems have no way of extending their reasoning to accommodate harder problems. In Chapter 4, we show that recurrent networks trained to solve simple problems with few recurrent steps can indeed solve much more complex problems simply by performing additional recurrences during inference. We demonstrate this algorithmic behavior of recurrent networks on prefix sum computation, maze solving, and chess puzzles. In all three domains, networks trained on simple problem instances are able to extend their reasoning abilities at test time simply by "thinking for longer."

The approaches described and studied in Chapter 4 constitute a simple effort to build neural networks that extrapolate. Indeed, we observe that this first approach fails to scale to highly complex problems because behavior degenerates when many iterations are applied – an issue we

refer to as "overthinking." In Chapter 5, we propose a recall architecture that keeps an explicit copy of the problem instance in memory so that it cannot be forgotten. We also employ a progressive training routine that prevents the model from learning behaviors that are specific to iteration number and instead pushes it to learn behaviors that can be repeated indefinitely. These innovations prevent the overthinking problem and enable Deep Thinking Systems to solve extremely hard logical extrapolation tasks, some requiring over 100K convolutional layers.

Chapter 2: Preliminaries

In this section, we review the pertinent details of neural networks and we describe the datasets used in the rest of this dissertation.

2.1 Deep Learning Background

2.1.1 Neural Networks

A neural network is a parameterization of a function made by composing *layers*, or operations, typically made of affine transformations and non-linear functions. Formally, let a layer of a neural network be denoted by ℓ. Then, a k-layer network can be defined by a function f that maps an input x to a output y where

$$f(x) = \ell_k \circ \ell_{k-1} \circ \cdots \circ \ell_2 \circ \ell_1(x). \tag{2.1}$$

In this general formulation, each member of the set $\{\ell_i\}_{i=1}^{k}$ can be a different function, including compositions of linear and nonlinear functions. A common example is called a *multi-layer perceptron* (MLP), which has $\ell_i = \sigma(A_i x + b_i)$ where σ is a rectified linear unit (ReLU) and is computed element-wise as $\max(\xi, 0)$. The entries in A_i and b_i are called the *parameters* of the network.

When the affine transformation is defined by a convolutional operator, we call the network a *convolutional neural network* (CNN). Convolutions are linear operations and they can be expressed in matrix form to match Equation (2.1). In this case, the parameters are the entries in the convolutional kernel.

Another important component of the neural networks used in this work are *skip connections* (also referred to as *residual connections*) where the input to one layer is used in computing the output of a later layer [He et al., 2016] For example, a three-layer network with a skip connection could be expressed as follows.

$$f(x) = \ell_3(\ell_2(\ell_1(x)) + x). \tag{2.2}$$

The particular set of functions composed in a given neural network define its *architecture*. Several popular architectures are used throughout this dissertation and are named and cited where appropriate.

The following additional terminology helps facilitate the discussions in the chapters that follow. The non-linear functions are often referred to as *activation* functions and their outputs for a given layer are called the *neurons* of that layer. The number of layers (k in Equation (2.1)) is called the *depth* and the number of neurons is called the *width*.

2.1.2 Training Neural Networks

In order to discuss training a neural network, we need to define to objective. For a classification task, consider a dataset D of samples from some distribution $\mathcal{D}$, where $D = \{(\mathbf{x}_j, y_j)\}_{j=1}^J$, which has J input-label pairs where the inputs are vectors. Then *training* is phrased as an opti-

mization problem in which we seek to minimize the expected loss over the data distribution of a network f_θ with parameters θ. The network parameters are the free variables that are optimized during *training* or *learning* – while solving the following optimization problem.

$$\min_\theta \mathbb{E}_{\mathcal{D}} \mathcal{L}(f_\theta(\mathbf{x}), y) \tag{2.3}$$

The loss $\mathcal{L}$ is task specific, but a common example for classification is cross-entropy. Let the output of f_θ be a vector ϕ, and define *softmax* as

$$s(\phi)_i = \frac{e^{\phi_i}}{\sum_j e^{\phi_j}}. \tag{2.4}$$

Then cross-entropy loss can be written as follows.

$$c(\phi, y) = -\log(s(\phi)_y) \tag{2.5}$$

Training a classification model can be described by a neural network that outputs a vector with the same number of entries as classes in the dataset, a set D of samples drawn from the data distribution, a loss function $\mathcal{L} = c(s(f_\theta(\mathbf{x})))$ that reflects a probability that the data is labelled correctly, and an optimizer to find network parameters that (approximately) minimize the loss.

2.1.3 Recurrence in Neural Networks

Recurrence shows up in several ways across the deep learning literature. Most commonly, it comes in the form of recurrent models that digest sequence data like time series or language

[e.g., Hochreiter and Schmidhuber, 1997]. In our work, we examine recurrence of a slightly different form, that is recurrent layers inside a neural network.

More formally, let r be a function representing a recurrent block, e.g. a composition of several layers, and let r^n denote n recurrences of that function, e.g. $r^2(x) = r(r(x))$. Let ϕ denote a feature map, or an output of r, and let $\phi_n = r^n(x)$. We also consider an initial *projection function* denoted by p, which projects an input instance into feature space, and also a final *output head* denoted by h, which maps features to outputs. A *Deep Thinking* (DT) network with m iterations of the recurrent block can then be expressed as follows.

$$f_\theta(x; m) := h(r^m(p(x))) \tag{2.6}$$

Recall that p, r, and h are special cases of the generic layer ℓ described above, and therefore have their own parameters, which all together comprise the parameters θ of the trainable model f_θ. Importantly, the parameters of r define one call to the function and so they are independent of m and allow us to vary the number of iterations without retraining, fine tuning, or adding parameters to the network. Also note, we refer to r as the *recurrent module* or *recurrent block*.

In Chapter 5, we introduce a modification to DT networks to boost performance, called *recall* – it is a concatenatentive skip connection. To formalize this architectural change, adding recall to the network can be expressed using the notation defined above as follows.

$$\begin{aligned} &f_{\text{recall}}(x; m) := h(r^m_{\text{recall}}(p(x), x)), \text{ where} \\ &r_{\text{recall}}(\phi, x) := r([\phi, x]) \text{ and } r^m_{\text{recall}}(\phi, x) = r_{\text{recall}}(r^{m-1}_{\text{recall}}(\phi, x), x) \end{aligned} \tag{2.7}$$

Whereas the input to r at iteration k is usually ϕ_{k-1}, with recall, the input to r at iteration k is

$[\phi_{k-1}, x]$, or the concatenation of the input with the feature map output by the previous recurrence. We add a single convolutional layer to map the input $[\phi_{k-1}, x]$ to a feature map of the same shape as ϕ_{k-1}. We refer to DT networks with concatenative skip connections as DT-Recall models. Equations 2.6 and (2.7) are repeated and renumbered in Chapter 5 as they originally appeared in the paper that chapter is based on and they provide necessary context to the discussion there.

The *effective depth* of a network (feed-forward or recurrent) is the number of layers, not necessarily unique, composed with one another to form f. Thus, the network in Equation (2.6) has an effective depth equal to the number of layers in p plus the number of layers in h plus m times the depth of r. This point is reiterated in Chapter 3 where we study the relationship between feed-forward and recurrent networks through the lens of effective depth.

2.2 Datasets for Studying Extrapolation from Easy to Hard

Joint work with Eitan Borgnia, Arjun Gupta, Arpit Bansal, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein.

In domains like computer vision, single and multi-agent games, and mathematical reasoning, classically trained models perform well on inputs from the same distribution used for training, but often fail to extrapolate their knowledge to more difficult tasks sampled from a different (but related) distribution. The goal of approaches like deep thinking and algorithm learning is to construct systems that achieve this extrapolation. With this in mind, we detail several datasets intended to motivate and facilitate novel research into systems that generalize from easy training data to harder test examples.

We present three datasets: *Prefix Sums*, *Mazes*, and *Chess Puzzles*. We also provide an easy to install Python package and accompanying documentation to make training and testing on this data accessible.[1]

2.2.1 Prefix Sums

The *Prefix Sums* dataset is meant to provide a simple toy baseline for testing new approaches, as models can be trained and tested rapidly on this dataset. Inputs are lists of binary $(0/1)$ numbers. Each sample comes with a label/target, which is a list of the same length containing the cumulative sums modulo two for the input strings. For 52 different input lengths, we provide sets of 10,000 examples each. The shortest strings available are 16 bits, and we have every length through 64 bits as well as 72, 128, 256, and 512.

The term prefix sums refers to computing cumulative sums of a list a numbers. In general, this can be done for any list, so the input $[21, 43, 18]$ has cumululative sums of $[21, 64, 82]$, but our attention in on binary strings. As such, we look at the cumulative sums modulo two. So the input $[1, 0, 1, 1]$ has cumulative sums of $[1, 1, 2, 3]$, and cumulative sums modulo two of $[1, 1, 0, 1]$. This task – computing prefix sums or cumulative sums modulo two – is a classical problem used for studying algorithms. The process is simple and a basic for-loop can produce the solution. Moreover, for our purposes, this problem can be described with only short examples, but an algorithmic approach (e.g. the for-loop) should scale to solve inputs of any length.

We refer to longer input strings as “harder” examples, which follows from the classical algorithms understanding of work required to compute the prefix sums. By offering so many different lengths, we provide many levels of difficulty.

[1]The source code is available at `http://github.com/aks2203/easy-to-hard-data`.

2.2.1.1 Data Generation

For each choice of the sample length n, we produce 10,000 unique random strings, each containing n binary numbers that represent a fair coin flip. We then compute their cumulative sums modulo two and save the input-output pairs.

2.2.1.2 Examples

Examples from the sets with 16 bit and 28 bit inputs are shown below in Figure 2.1.

Input: $[1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1]$
Target: $[1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0]$

Input: $[1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0]$
Target: $[1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1]$

Figure 2.1: *Prefix Sums* input samples and their corresponding targets/labels. We provide multiple sets, each containing problems of a different length, and intend for users to train on shorter strings and test on longer ones. Examples from the sets of length 16 and 28 are shown above.

2.2.2 Mazes

The *Mazes* dataset contains images of mazes and their solutions. The solutions are represented as segmentation maps (or a per-pixel binary classification) of the input pixels, with one class for pixels that are on the optimal (i.e., shortest) path from start to finish, and another class for those that are not on the optimal path. The inputs are three-channel (RGB) images. The "start" of the maze is marked in red, the "end" of the maze is marked by a green square, and the walls of the maze are black. Every square where the player is allowed to move is white. Note that the "start" and "end" positions of an example can be swapped without changing the label; we do not care about the direction in which the path is walked.

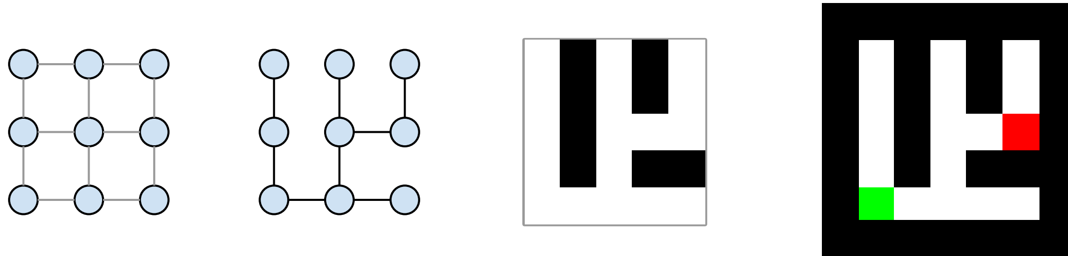

Figure 2.2: The *Mazes* generation process for making a 5×5 maze. We start with a 3×3 grid graph. Each side of the grid graph contains 3 nodes and 2 edges ($3 + 2 = 5$ total elements). We then produce a spanning tree for the graph using a randomized algorithm. The tree encodes the allowed and forbidden paths. This tree is then represented as a 5×5 array of maze cells. Finally, we convert this to an image by representing each cell as a 2×2 array of pixels, and adding a 3-pixel border on each side. This creates an image representation that has $5 \times 2 + 3 + 3 = 16$ pixels on each side. The green and red start and end cells are chosen at random.

2.2.2.1 Data Generation

We generate the mazes as abstract graphs, and then render them as images. We initialize a square grid graph, then use depth first search to find a subset of the edges that form a spanning tree. We randomly assign two nodes in the graph to be the end points of the maze. To render an image from this representation, we represent each node and each edge in the spanning tree with a white cell and each edge from the grid graph that is not in the spanning tree as a black cell. See Figure 2.2 for a depiction of each stage in the maze generation process. The code used was adapted from another maze generation repository [Hill, 2017]. Finally, the solutions are generated with breadth first search, using the image as input. By construction, the generated mazes admit a unique solution, in other words, there is exactly one path connecting the "start" and "end" positions.

A note on the size of mazes: After creating an $n \times n$ maze, we represent the graph using $2 \times 2 = 4$ pixels per cell, and add a 3 pixel wide black border on each side. For this reason, a dataset of $n \times n$ mazes is represented using images of dimension $(2n + 6) \times (2n + 6)$. For

example, the 9×9 *Mazes* dataset contains images of size 24×24 pixels.

2.2.2.2 Examples

Figure 2.3 shows examples of mazes and their solutions from datasets of three different difficulties. An extra large maze is shown in Figure 2.4 and solving this fun maze is left as an excercise to the reader. Figure 2.5 shows an absurdly large maze.

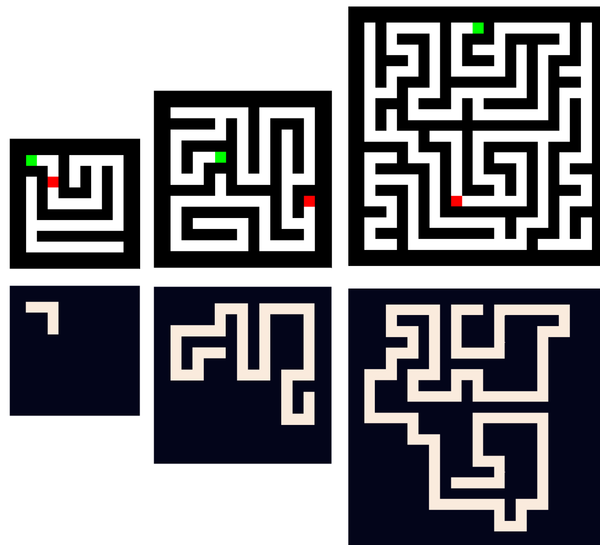

Figure 2.3: Samples from *Mazes*. Mazes (top) and their labels/solutions (bottom) of size 9×9, 13×13, and 21×21.

2.2.3 Chess Puzzles

Training a full-scale chess game system is a complex task that requires large code bases and lots of compute resources. The chess puzzles compiled and provided by Lichess [Lichess, 2021] make for a much more accessible task. In these puzzles, the inputs are mid-game boards and the outputs are the unique best next move as determined by a state of the art chess engine. Here, we provide a *Chess Puzzles* dataset that encodes the complex decision problems required to play

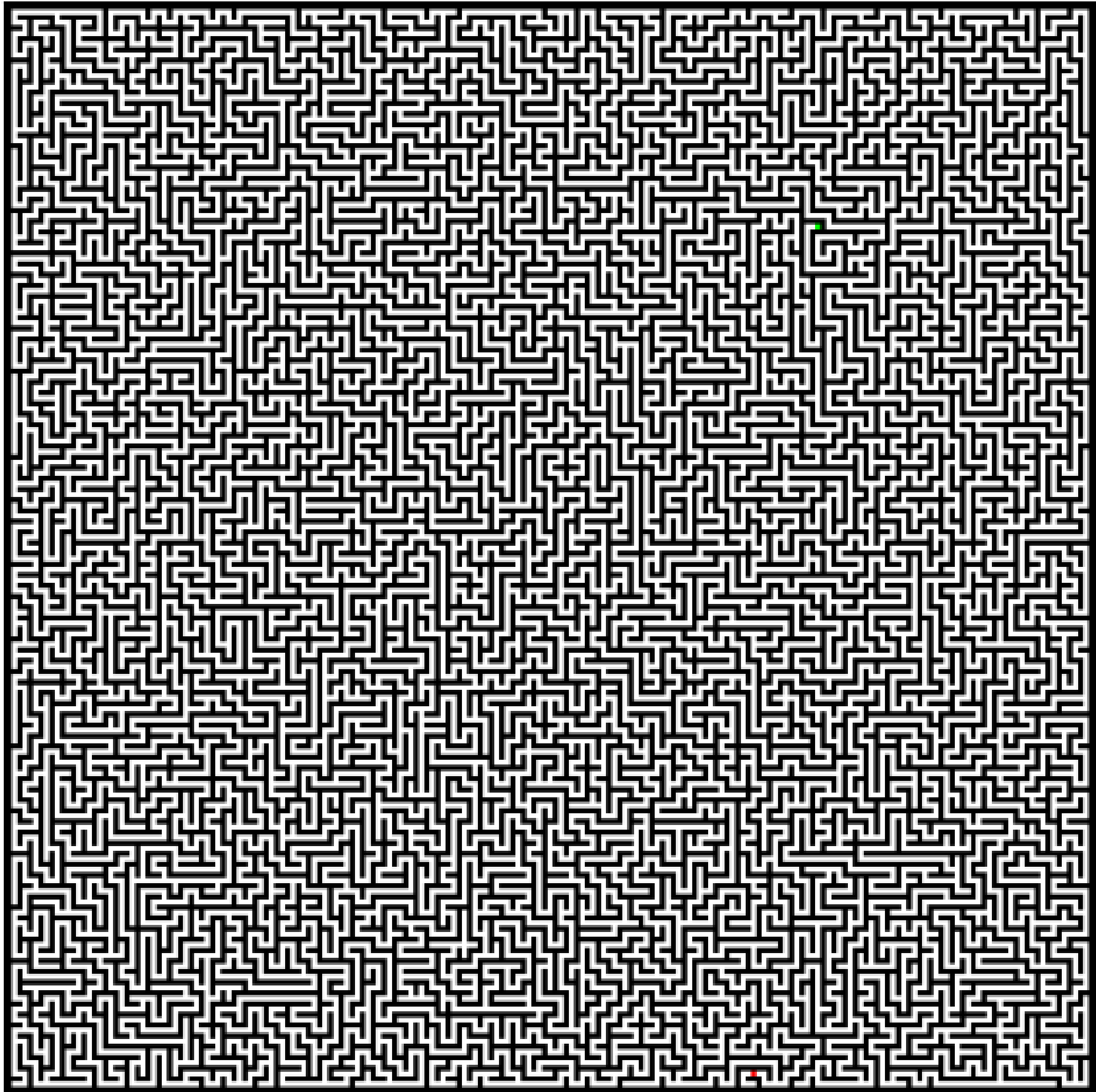

Figure 2.4: Fun for the whole family! We leave solving this 201×201 maze as an exercise for the reader. The green starting dot is near the center of the upper right quadrant. The red ending point is on the second row from the bottom.

chess, but is formatted as a simple pixel-wise classification problem that is easily used for training and testing. Our dataset consists of images representing game states (i.e., board configurations) and labels for each game state, which are 2D maps in which the source and destination positions for the optimal move are marked with a 1, and all other positions are marked with a 0.

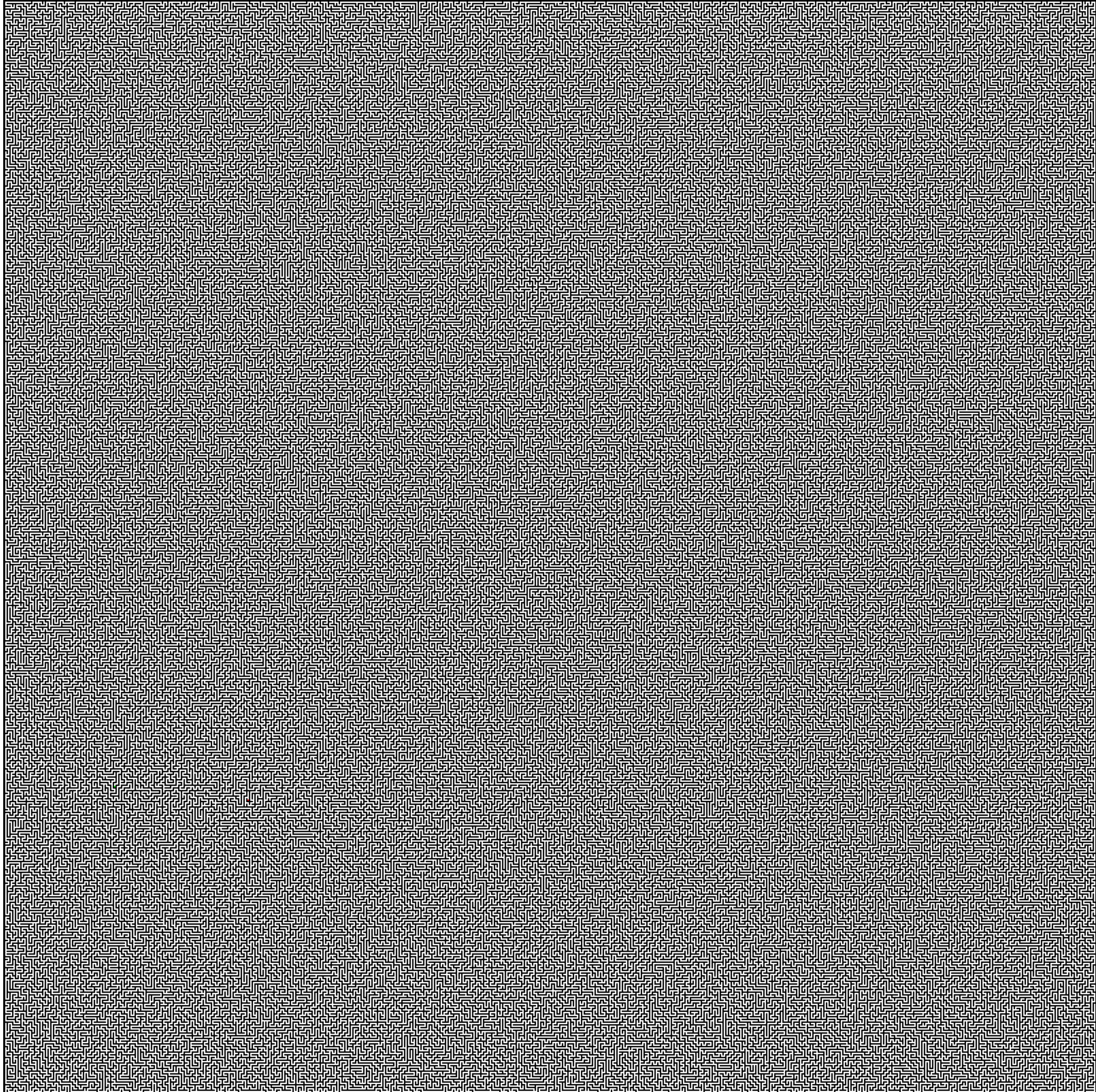

Figure 2.5: An example of a *very, very* fun 801×801 maze. The green starting dot is three quarters of the way down on the left-hand side. A model trained using only 9×9 mazes and 30 iterations is able to solve this extreme maze at test time using 10,000 iterations, which is equivalent to 100,004 convolutional layers.

2.2.3.1 Data Generation

By combing through 200,000,000 user games using the Stockfish 12/13 chess engines, Lichess creates puzzles consisting of sequences of unique best next moves. Once available for

play on Lichess's website, a puzzle's initial Elo rating is then refined by having users with known ratings attempt to defeat the puzzle.[2] Interactions with hundreds/thousands of users eventually leave the puzzle with an equilibrated final rating, which quantifies difficulty. Each puzzle is tagged with the Forsyth-Edwards Notaion (FEN) representation of the board and an automatically generated short descriptor using code made available on the Lichess website. In other words, we can download a string that describes the current board, the best next move, and the difficulty rating for each of these puzzles.

We use FEN information to create a Pytorch tensor that encodes the board state as a multi-channel "image." This representation consists of twelve 8×8 channels, one for each class of white piece and one for each class of black piece. In particular, each channel has ones at the locations of the relevant pieces (e.g., pawns for one channel, rooks for another, etc..) and zeros elsewhere. The first six channels correspond to the player who is to move next. For this reason, a machine learning system that accepts this representation should not need to be explicitly told which player acts next, however a boolean flag for this purpose can be requested by the user when the dataset is constructed. Note, that our formulation of this dataset leaves out information about castling rights and *En passant* – a simplification of the game we make for convenience.

Each board state is labeled with a single-channel 8×8 tensor representing the solution to the problem. In these tensors there are zeros everywhere except on the origin and destination squares for the piece that should be moved. Again, note that the target itself does not possess the explicit information about what type of piece or which color is to move, but that information can be determined with the input and the identification of which color's turn it is.

[2]Elo is a system for rating players; one increases their Elo by besting other players with high Elo, and decreases their Elo by failing against lower rated players. Chess puzzles are rated similarly by presenting them to players with known skill rating.

2.2.3.2 Examples

Examples of puzzles with their solutions are shown in Figure 2.6.

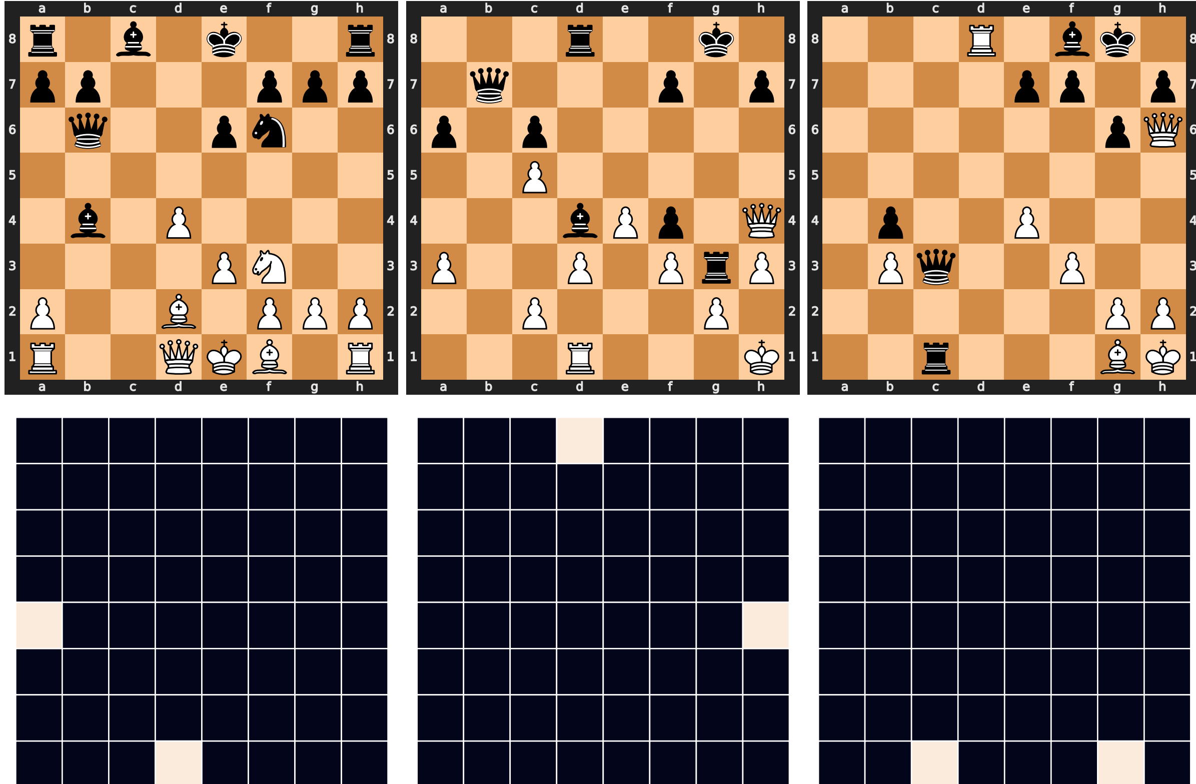

Figure 2.6: Samples from *Chess Puzzles*. "Easy" chess puzzles, each with their solution below. The board state is represented as a $12 \times 8 \times 8$ tensor, where the first 6 maps encode the pieces belonging to the player to act next. The white player is acting in the two leftmost puzzles, while the black player acts in the puzzle on the right. The solution is represented as a $1 \times 8 \times 8$ tensor that marks the start and end position of the piece to be moved.

2.2.4 Python Package

The Python package for these datasets is available through the Package Installer for Python (pip). We provide Python classes for each problem type, and the class constructors accept an argument specifying the desired difficulty for the dataset.[3]

[3]The documentation and code is available at `https://pypi.org/project/easy-to-hard-data/`

When a dataset object is created, the user can point the constructor to the location of the downloaded raw data, or else a flag can be set to perform this download automatically.

In addition to initializing dataset objects, the repository also houses code for generating the data and code to make visualizations for mazes and chess. The dataset objects rely on standard Pytorch features, and should be easy to modify to create variations on the standard benchmark problems.

Chapter 3: The Uncanny Similarity of Recurrence and Depth

Joint work with Arjun Gupta, Amin Ghiasi, Micah Goldblum, and Tom Goldstein. Appeared in ICLR 2022.

It is widely believed that deep neural networks contain layer specialization, wherein neural networks extract hierarchical features representing edges and patterns in shallow layers and complete objects in deeper layers. Unlike common feed-forward models that have distinct filters at each layer, recurrent networks reuse the same parameters at various depths. In this work, we observe that recurrent models exhibit the same hierarchical behaviors and the same performance benefits with depth as feed-forward networks despite reusing the same filters at every recurrence. By training models of various feed-forward and recurrent architectures on several datasets for image classification as well as maze solving, we show that recurrent networks have the ability to closely emulate the behavior of non-recurrent deep models, often doing so with far fewer parameters.

3.1 Introduction

It is well-known that adding depth to neural architectures can often enhance performance on hard problems [He et al., 2016, Huang et al., 2017]. State-of-the-art models contain hundreds

of distinct layers [Tan and Le, 2019, Brock et al., 2021]. However, it is not obvious that recurrent networks can experience performance boosts by conducting additional iterations, since this does not add any parameters. On the other hand, increasing the number of iterations allows the network to engage in additional processing. In this work, we refer to the number of sequential layers (possibly not distinct) as effective depth. While it might seem that the addition of new parameters is a key factor in the behavior of deep architectures, we show that recurrent networks can exhibit improved behavior, closely mirroring that of deep feed-forward models, simply by adding recurrence iterations (to increase the effective depth) and no new parameters at all.

In addition to the success of very deep networks, a number of works propose plausible concrete benefits of deep models. In the generalization literature, the sharp-flat hypothesis submits that networks with more parameters, often those which are very wide or deep, are biased towards flat minima of the loss landscape which are thought to coincide with better generalization [Keskar et al., 2016, Huang et al., 2020, Foret et al., 2020]. Feed-forward networks are also widely believed to have *layer specialization*. That is, the layers in a feed-forward network are thought to have distinct convolutional filters that build on top of one another to sequentially extract hierarchical features [Olah et al., 2017, Geirhos et al., 2018]. For example, the filters in shallow layers of image classifiers may be finely tuned to detect edges and textures while later filters may be precisely tailored for detecting semantic structures such as noses and eyes [Yosinski et al., 2015]. In contrast, the filters in early and deep iterations of recurrent networks have the very same parameters. Despite their lack of layer-wise specialization and despite the fact that increasing the number of recurrences does not increase the parameter count, we find that recurrent networks can emulate both the generalization performance and the hierarchical structure of deep feed-forward networks, and the relationship between depth and performance is similar regardless of whether

recurrence or distinct layers are used.

Our contributions can be summarized as follows:

- We show that image classification accuracy changes with depth in remarkably similar ways for both recurrent and feed-forward networks.

- With optimization based feature visualizations, we show that recurrent and feed-forward networks extract the very same types of features at different depths.

3.1.1 Related Work

Recurrent networks are typically used to handle sequential inputs of arbitrary length, for example in text classification, stock price prediction, and protein structure prediction [Lai et al., 2015, Borovkova and Tsiamas, 2019, Goldblum et al., 2020, Baldi and Pollastri, 2003]. We instead study the use of recurrent layers in place of sequential layers in convolutional networks and on non-sequential inputs, such as images. Prior work on recurrent layers, or equivalently depth-wise weight sharing, also includes studies on image segmentation and classification as well as sequence-based tasks. For example, Pinheiro and Collobert [2014] develop fast and parameter-efficient methods for segmentation using recurrent convolutional layers. Alom et al. [2018] use similar techniques for medical imaging. Also, recurrent layers have been used to improve performance on image classification benchmarks [Liang and Hu, 2015]. Recent work developing transformer models for image classification shows that weight sharing can reduce the size of otherwise large models without incurring a large performance decrease [Jaegle et al., 2021].

Additionally, the architectures we study are related to weight tying. Eigen et al. [2013]

explore very similar dynamics in CNNs, but they do not address MLPs or ResNets, and Boulch [2017] studies weight sharing only in ResNets. More importantly, neither of those works have analysis beyond accuracy metrics, whereas we delve into what exactly is happening at various depths with feature visiualization, linear separability probes, and feature space similarity metrics. Bai et al. [2018] propose similar weight tying for sequences, and they further extend this work to include equilibrium models, where infinite depth networks are found with root-finding techniques [Bai et al., 2018, 2019]. Equilibrium models and neural ODEs make use of the fact that hidden states in a recurrent network can converge to a fixed point, training models with differential equation solvers or root finding algorithms [Chen et al., 2018, Bai et al., 2019].

Prior work towards understanding the relationship between residual and highway networks and recurrent architectures reveals phenomena related to our study. Greff et al. [2016] present a view of highway networks wherein they iteratively refine representations. The recurrent structures we study could easily be thought of in the same way, however they are not addressed in that work. Also, Jastrzebski et al. [2017] primarily focus on deepening the understanding of iterative refinement in ResNets. While our results may strengthen the argument that ResNets are naturally doing something iterative, this is tangential to our broader claims that for several families of models depth can be achieved with distinct layers or with recurrent ones. Similarly, Liao and Poggio [2016] show that deep RNNs can be rephrased as ResNets with weight sharing and even include a study on batch normalization in recurrent layers. Our work builds on theirs to further elucidate the similarity of depth and recurrence. Specifically, we carry out quantitative and qualitative comparisons between the deep features, as well as performance, of recurrent and analogous feed-forward architectures as their depth scales. We analyze a variety of additional models, including multi-layer perceptrons and convolutional architectures which do not include

residual connections.

Our work expands the scope of the aforementioned studies in several key ways. First, we do not focus on model compression, but we instead analyze the relationship between recurrent networks and models whose layers contain distinct parameters. Second, we use a standard neural network training process, and we study a variety of architectures. Finally, we conduct our experiments on image data, as well as a reasoning task, rather than sequences, so that our recurrent models can be viewed as standard fully-connected or residual networks but with additional weight-sharing constraints.

For each family of architectures and each dataset we study, we observe various trends as depth, or equivalently the number of recurrence iterations, is increased that are consistent from recurrent to feed-forward models. These relationships are often not simple, for example classification accuracy does not vary monotonically with depth. However, the trends are consistent whether depth is added via distinct layers or with iterations. We are the first to our knowledge to make qualitative and quantitative observations that recurrence mimics depth.

3.2 Recurrent Architectures

Widely used architectures, such as ResNet and DenseNet, scale up depth in order to boost performance on benchmark image classification tasks [He et al., 2016, Huang et al., 2017]. We show that this trend also holds true in networks where recurrence replaces depth. Typically, models are made deeper by adding layers with new parameters, but recurrent networks can be made deeper by recurring through modules more times without adding any parameters.

To discuss this further, it is useful to define *effective depth*. Let a layer of a neural network

be denoted by ℓ. Then, a p-layer feed-forward network can be defined by a function f that maps an input x to a label y where

$$f(x) = \ell_p \circ \ell_{p-1} \circ \cdots \circ \ell_2 \circ \ell_1(x). \tag{3.1}$$

In this general formulation, each member of the set $\{\ell_i\}_{i=1}^{p}$ can be a different function, including compositions of linear and nonlinear functions. The recurrent networks in this study are of the following form.

$$f(x) = \ell_p \circ \cdots \circ \ell_{k+1} \circ m_r \circ m_r \circ \cdots \circ \ell_k \circ \cdots \circ \ell_1(x) \tag{3.2}$$

We let $m_r(x) = \ell_q^{(m)} \circ \cdots \circ \ell_1^{(m)}(x)$ and call it a q-layer *recurrent module*, which is applied successively between the first and last layers. The *effective depth* of a network (feed-forward or recurrent) is the number of layers, not necessarily unique, composed with one another to form f. Thus, the network in Equation (3.2) has an effective depth of $p + nq$, where n is the number of iterations of the recurrent module. A more concrete example is a feed-forward network with seven layers, which has the same effective depth as a recurrent model with two layers before the recurrent module, two layers after the recurrent module, and a single-layer recurrent module that runs for three iterations. See Figure 3.1 for a graphical representation the effective depth measurement.

We train and test models of varying effective depths on ImageNet, CIFAR-10, EMNIST, and SVHN to study the relationship between depth and recurrence [Russakovsky et al., 2015, Krizhevsky et al., 2009, Cohen et al., 2017, Netzer et al., 2011]. For every architecture style

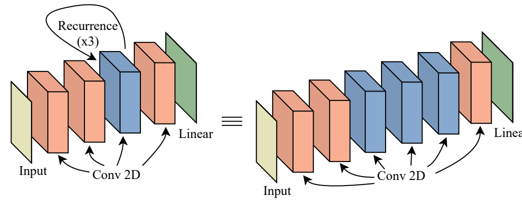

Figure 3.1: Effective depth measures the depth of the unrolled network. For example, a network with 3 iterations of its single-layer recurrent module and 4 additional layers has an effective depth of 7.

and every dataset, we see that depth can be emulated by recurrence. The details of the particular recurrent and feed-forward models we use in these experiments are outlined below.

The specific architectures we study in the context of image classification include families of multi-layer perceptrons (MLPs), ConvNets, and residual networks. Each recurrent model contains an initial set of layers, which serve as a projection from input space to feature space, followed by a module whose inputs and outputs have the same shape. As a result of the latter property, this module can be iteratively applied in a recurrent fashion an arbitrary number of times. The feed-forward analogues of these recurrent models instead stack several unique instances of the internal module on top of each other. In other words, our recurrent models can be considered the same as their feed-forward counterparts but with depth-wise weight sharing between each instance of the internal module.

3.2.1 Multi-layer Perceptrons

The MLPs we study are defined by the width of their internal modules which is specified per dataset below. We examine feed-forward and recurrent MLPs with effective depths from three to 10. The recurrent models have non-recurrent layers at the beginning and end, and they have one layer in between that can be recurred. For example, if a recurrent MLP has effective

depth five, then the model is trained and tested with three iterations of this single recurrent layer. A feed-forward MLP of depth five will have the exact same architecture, but instead with three distinct fully-connected layers between the first and last layers. All MLPs we use have ReLU activation functions between fully-connected layers.

All of the MLPs have a single fully connected layer before the internal module, and a single fully connected layer afterward. Each layer in an MLP can be defined by a matrix dimension, which are outlined in Table 3.1.

Table 3.1: MLPs: The size of each fully connected layer in the MLPs we use for image classification.

Dataset	First Layer	Internal Layers	Last Layer
CIFAR-10	3072×200	200×200	200×10
SVHN	3072×500	500×500	500×10
EMNIST	3072×500	500×500	500×47

3.2.2 Convolutional Networks

Similarly, we study a group of ConvNets which have two convolutional layers before the internal module and one convolutional and one fully connected layer afterward. The internal module has a single convolutional layer with a ReLU activation, which can be repeated any number of times to achieve the desired effective depth. Here too, the output dimension of the second layer defines the width of the model, and this is indicated below for specific models. Each convolutional layer is followed by a ReLU activation, and there are maximum pooling layers before and after the last convolutional layer. For all convolutional layers outside of the internal module, we use 3×3 filters with stride of one and no padding. In order to preserve the exact shape of the inputs, the internal module has convolutional layers with 3×3 kernels, a stride of one

pixel, and padding of one pixel in each direction. We train and test feed-forward and recurrent ConvNets with effective depths from four to eight.

Table 3.2: ConvNets: The number of channels in the output of each layer.

Dataset	First Layer	Second Layer	Internal Layers	Last Conv Layer	Linear Layer
CIFAR-10	32	64	64	128	10
SVHN	32	64	64	128	10
EMNIST	32	64	64	128	47

3.2.3 Residual Networks

We also employ networks with residual blocks like those introduced by He et al. [2016]. The internal modules are residual blocks with four convolutional layers, so recurrent models have a single recurrent residual block with four convolutional layers. There is one convolutional layer before the residual block and one after, which is followed by a linear layer. The number of channels in the output of the first convolutional layer defines the width of these models. Each convolutional layer is followed by a ReLU activation. As is common with residual networks, we use average pooling before the linear classifier. The residual networks we study have effective depths from seven to 19.

The residual networks employ convolutional layers with 3×3 kernels. The first layer has a stride of two pixels and padding of one pixel in every direction. Each layer in the internal modules has striding by one pixel, while the final convolutional layer strides by two pixels. There are ReLU activations after every convolution, and average pooling before the linear layer. All the convolutional layers output 512 channels.

We also train and test residual networks with BatchNorm in order to compare the performance with ResNet-18 models. These residual networks have a unique batch norm layer after

every convolutional layer followed by ReLU activation function. Since the input to the recurrent module at iteration n is the output at iteration $n - 1$ (for $n = 2, 3, 4, ...$), the batch statistics for every iteration is different. Thus, we use separate batch norm layer for each iteration.

For solving mazes, we train and test residual networks with slightly different architectures. These models are fully convolutional and every convolutional layer has 3×3 kernels that stride by one pixel with padding of one pixel in each direction. They have one layer before the internal module, residual blocks of four layers in the internal module, and three convolutional layers afterward. All of the layers before the third to last one have 128 output channels, and in the final three layers the numbers of output channels are 32, eight, and two.

3.3 Matching Image Classification Accuracies

Across all model families, we observe that adding iterations to recurrent models mimics adding layers to their feed-forward counterparts. This result is not intuitive for three reasons. First, the trend in performance as a function of effective depth is not always monotonic and yet the effects of adding iterations to recurrent models closely match the trends when adding unique layers to feed-forward networks. Second, adding layers to feed-forward networks greatly increases the number of parameters, while recurrent models experience these same trends without adding any parameters at all. On the CIFAR-10 dataset, for example, our recurrent residual networks each contain 12M parameters, and yet models with effective depth of 15 perform almost exactly as well as a 15-layer feed-forward residual network with 31M parameters. Third, recurrent models do not have distinct filters that are used in early and deep layers. Therefore, prior intuition that early filters are tuned for edge detection while later filters are tailored to pick up

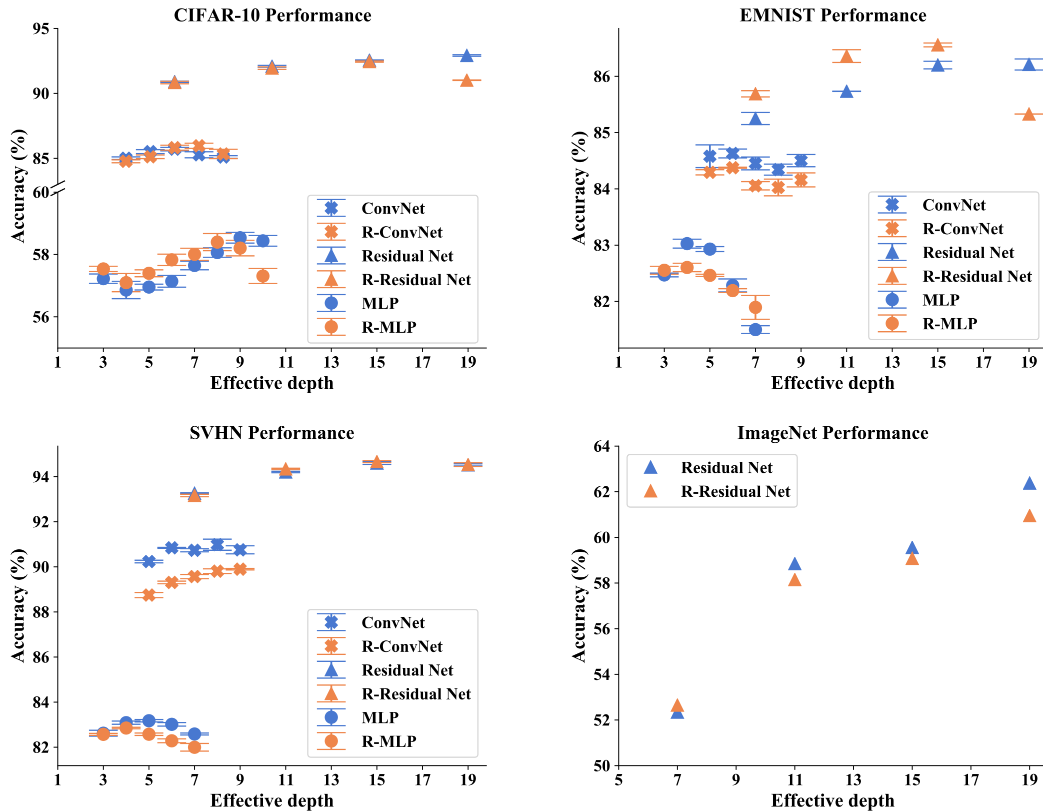

Figure 3.2: Average accuracy of each model. The x-axis represents effective depth and the y-axis measures test accuracy. The marker shapes indicate architectures, and the blue and orange markers indicate feed-forward and recurrent models, respectively. The horizontal bars are $\pm$ one standard error from the averages over several trials. Across all three families of neural networks, recurrence can imitate depth. Note, the y-axes are scaled per dataset to best show each of the trends.

higher order features cannot apply to the recurrent models [Olah et al., 2017, Yosinski et al., 2015].

In Figure 3.2, the striking result is how closely the blue and orange markers follow each other. For example, on SVHN, we see from the triangular markers that residual networks generally perform better as they get deeper and that recurrent residual models with multiple iterations perform as well as their non-recurrent counterparts. On CIFAR-10, we see that MLP performance as a function of depth is not monotonic, yet we also see the same pattern here as we add iterations to recurrent models as we do when we add distinct layers to non-recurrent networks.

The increasing trend in the ImageNet accuracies, which are matched by the recurrent and feed-forward models, suggests that even deeper models might perform even better. In fact, we can train a ResNet-50 and achieve 76.48% accuracy on the testset. While training our recurrent models, which do not have striding or pooling, at that depth is difficult, we can train a ResNet-50 modified so that layers within residual blocks that permit inputs/outputs of the same shape share weights. This model is 75.13% accurate on the testset. These experiments show that the effect of added depth in these settings is consistent whether this depth is achieved by iterating a recurrent module or adding distinct layers.

3.3.1 Separate Batch Normalization Statistics for Each Recurrence

The models described above comprise a range of architectures for which recurrence can mimic depth. However, those models do not employ the modern architectural feature batch normalization [Ioffe and Szegedy, 2015]. One might wonder if the performance of recurrent networks is boosted even further by the tricks that help state-of-the-art models achieve such high performance. We compare a recurrent residual network to the well-known ResNet-18 architecture [He et al., 2016]. There is one complexity that arises when adding batch normalization to recurrent models; despite sharing the same filters, the feature statistics after each iteration of the recurrent module may vary, so we must use a distinct batch normalization layer for each iteration. Although it is not our goal to further the state of the art, Table 3.3 shows that these recurrent models perform nearly as well as ResNet-18, despite containing only a third of the parameters. This recurrent architecture is not identical to the models used above, rather it is designed to be as similar to ResNet-18 as possible (as described above in Section 3.2).

Table 3.3: **Models with batch normalization.** Performance shown for three different classification datasets. We report averages ($\pm$ one standard error) from three trials.

Dataset	Model	Params	Acc (%)
CIFAR-10	Ours	3.56M	93.99 ± 0.10
	ResNet-18	11.2M	94.69 ± 0.03
SVHN	Ours	3.56M	96.04 ± 0.08
	ResNet-18	11.2M	95.48 ± 0.11
EMNIST	Ours	3.57M	87.42 ± 0.19
	ResNet-18	11.2M	87.71 ± 0.09

3.4 The Maze Problem

To delve deeper into the capabilities of recurrent networks, we shift our attention from image classification and introduce the task of solving mazes.[1] In this dataset, the mazes are represented by three-channel images where the permissible regions are white, the walls are black, and the start and end points are green and red squares, respectively. We generate 180,000 mazes in total, 60,000 examples each of small (9x9), medium (11x11), and large mazes (13x13). We split each set into 50,000 training samples and 10,000 testing samples. The solutions are represented as a single-channel output of the same spatial dimensions as the input maze, with 1's on the optimal path from the green square to the red square and 0's elsewhere. For a model to "solve" a maze, it must output a binary segmentation map – each pixel in the input is either on the optimal path or not. We only count a solution as correct if it matches labels every pixel correctly. See Chapter 2 for images of example mazes and more details of the dataset.

[1]The Mazes dataset described in Chapter 2 was first introduced as a part of the publication that this chapter is from. While the datatset is not new at this point in this dissertation, it is reviewed here for context.

3.4.1 Maze Solving Networks

We extend our inquiry into recurrence and depth to neural networks that solve mazes. Specifically, we look at residual networks' performance on the maze dataset. The networks take a maze image as input and output a binary classification for each pixel in the image. These models have one convolutional layer before the internal module and three convolutional layers after. Similar to classification networks, the internal modules are residual blocks and a single block is repeated in recurrent models while feed-forward networks have a stack of distinct blocks. We employ residual networks with effective depths from eight to 44 layers.

3.4.2 Solving Mazes Requires Depth (or Recurrence)

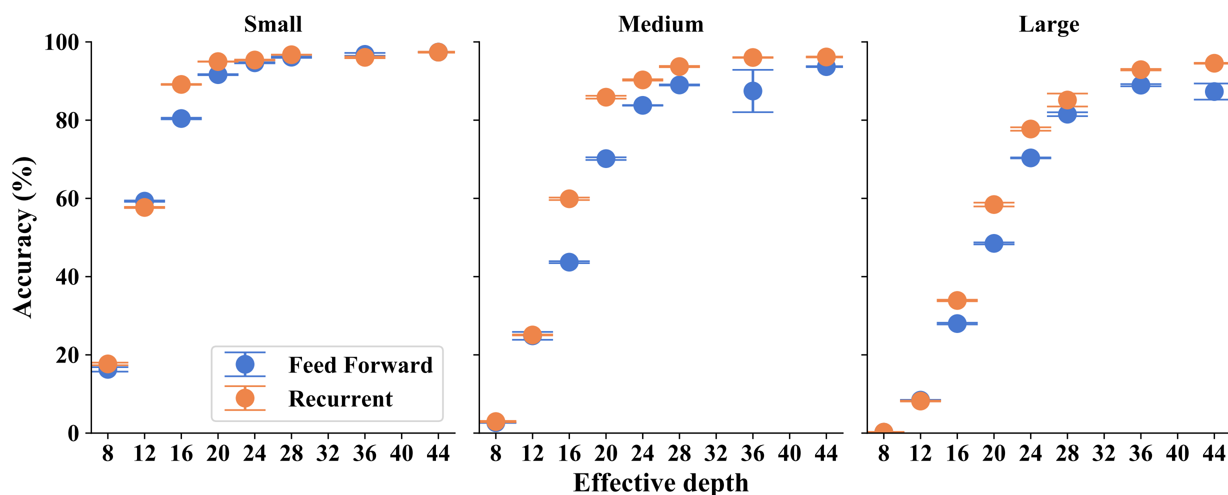

Figure 3.3: Recurrent models perform similarly to feed-forward models of the same effective depth. All models here are residual models as described in Section 3.4.

We train several recurrent and non-recurrent models of effective depths ranging from eight to 44 on each of the three maze datasets, and we see that deeper models perform better in general. Figure 3.3 shows these results. More importantly, we observe that recurrent models perform

almost exactly as well as feed-forward models with equal effective depth. Another crucial finding confirmed by these plots is that larger mazes are indeed harder to solve. We see that with a fixed effective depth, models perform the best on small mazes, then medium mazes, and the worst on large mazes. This trend matches the intuition that larger mazes require more computation to solve, and it confirms that these datasets do, in fact, constitute problems of varying complexity.

One candidate explanation for these trends is the *receptive field*, or the range of pixels in the input that determine the output at a given pixel. Intuitively, solving mazes requires global information rather than just local information which suggests that the receptive field of successful models must be large. While the receptive field certainly grows with depth, this does not fully account for the behavior we see. By computing the receptive field for each effective depth, we can determine whether the accuracy improves with even more depth beyond what is needed for a complete receptive field. For the small mazes, the receptive field covers the entire maze after eight 3×3 convolutions, and for the large mazes, this takes 12 such convolutions. In Figure 3.3, we see performance rise in all three plots even after effective depth grows beyond 20. This indicates that receptive field does not fully explain the benefits of effective depth – there is further improvement brought about by adding depth, whether by adding unique parameters or reusing filters in the recurrent models.

We can further test this by training identical architectures to those described above, except that the 3×3 convolutional filters are dilated (this increases the receptive field without adding parameters or depth). In fact, these models are better at solving mazes, but most importantly, we also see exactly the same relationship between recurrence and depth. This confirms that recurrence has a strong benefit beyond merely increasing the receptive field. See Table 3.4 for numerical results.

Table 3.4: **Dilated Filters.** The average accuracy of models with dilated filters trained and tested on small mazes.

	Effective Depth			
Model	8	12	16	20
Recurrent	75.07	90.41	94.70	95.19
Feed-forward	75.59	90.58	93.84	95.00

3.5 Recurrent Models Reuse Features

Test accuracy is only one point of comparison, and the remarkable similarities in Figure 3.2 raise the question: What are the recurrent models doing? Are they, in fact, recycling filters, or are some filters only active on a specific iteration? In order to answer these questions, we look at the number of positive activations in each channel of the feature map after each recurrence iteration for a batch of 1,000 randomly selected CIFAR-10 test set images. For each image-filter pair, we divide the number of activations at each iteration by the number of activations at the most active iteration (for that pair). If this measurement is positive on iterations other than the maximum, then the filter is being reused on multiple iterations. In Figure 3.7, it is clear that a large portion of the filters have activation patterns on their least active iteration with at least 20% of the activity of their most active iteration. This observation leads us to conclude that these recurrent networks are indeed reusing filters, further supporting the idea that the very same filters can be used in shallow layers as in deep layers without a loss in performance.

We can similiarly depict the activity in a ConvNet trained on SVHN and in a residual network trained on CIFAR-10. See Figures 3.5 and 3.6. Although these plots show distinct behavior from Figure 3.7, the conclusions are consistent. In recurrent models, features are being reused. This is clear since every filter is being used on more than one iteration, and for many

filters there is significant activity in most iterations.

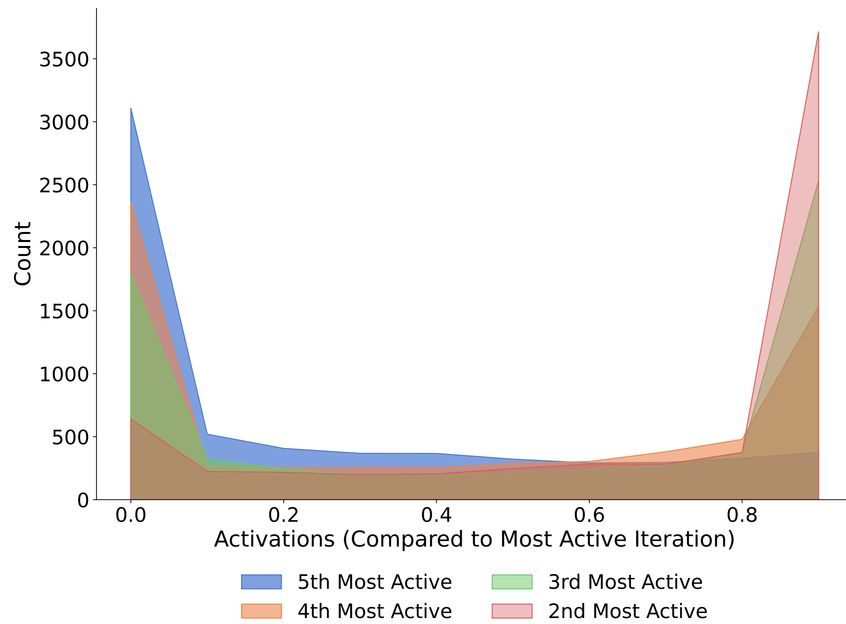

Figure 3.4: Relative activations of the recurrent layer in an untrained CNN. This plot confirms that the distributions in the other similar plots from trained models are significant.

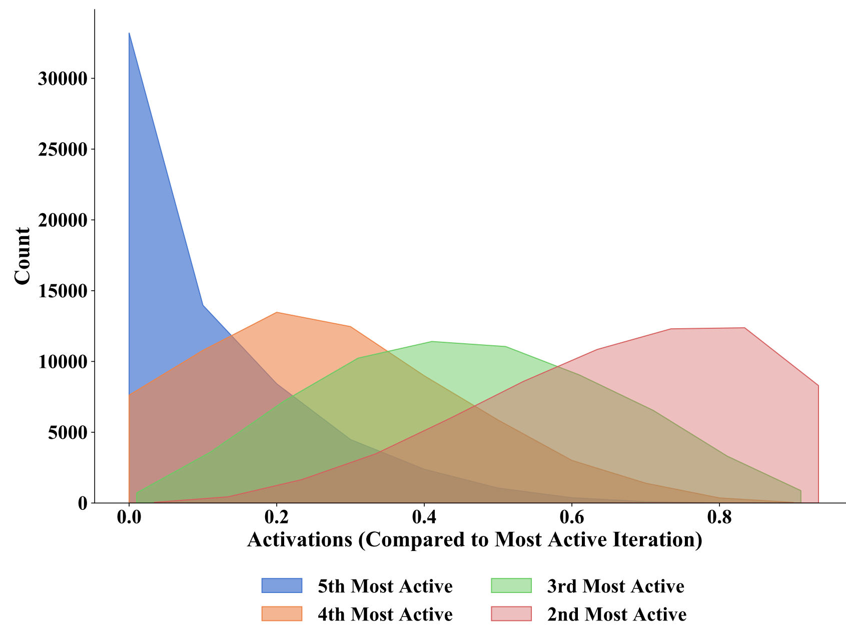

Figure 3.5: Relative activations of the recurrent layer of a CNN trained on SVHN.

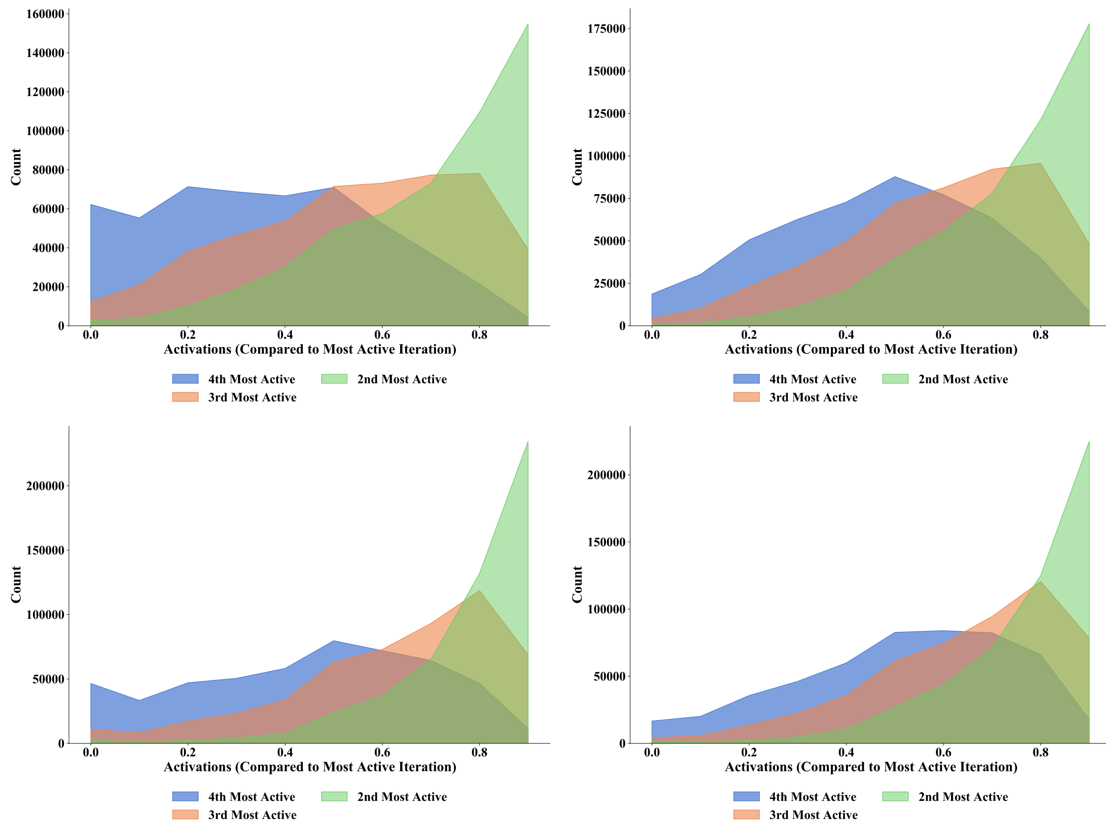

Figure 3.6: Relative activations at each of the four layers in the recurrent module of a residual network (with a four-layer residual block as the recurrent module) trained on CIFAR-10. The top row is from the first and second layers and the bottom row shows the third and fourth layers.

3.5.1 Recurrent and Feed-forward Models Similarly Disentangle Classes

In order to compare the features that image classifiers extract, we train linear classifiers on intermediate features from various layers of the network. In trying to classify feature maps with linear models, we can analyze a network's progress towards disentangling the features of different classes. For this experiment, we examine the features of CIFAR-10 images output by both feed-forward and recurrent residual networks. Similar to the method used by Kaya et al. [2019], we use the features output by each residual block to train a linear model to predict the labels based on these feature vectors.

Table 3.5: Classifiability of features. CIFAR-10 classification accuracy of linear classifiers trained on intermediate features output by each residual block (or each iteration) in networks with 19-layer effective depths. Each cell reports the percentage of training/testing images correctly labeled.

	Block #1	Block #2	Block #3	Block #4
Feed-forward	84.7 / 81.6	94.1 / 86.9	99.3 / 89.8	100.0 / 90.6
Recurrent	86.3 / 82.3	95.9 / 87.5	99.5 / 89.4	99.9 / 90.3

Using residual networks with effective depth of 19 (as this is the best performing feed-forward depth), we show that the deeper features output by each residual block, or iteration in the recurrent model, are increasingly separable. More importantly, as shown in Table 3.5, the feed-forward and recurrent models are separating features very similarly at corresponding points in the networks.

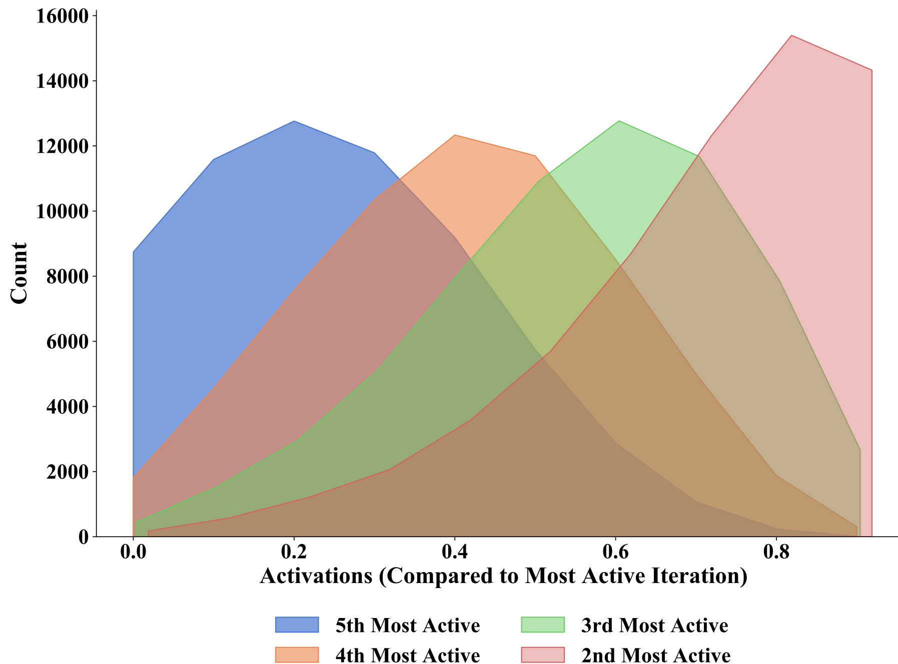

Figure 3.7: The number of active neurons after each iteration of a recurrent ConvNet with 5 iterations. Note that the most active iteration is left off this plot since we are normalizing by the number activations at that iteration.

Additional experiments on the linear classifiability of features in recurrent models shows

that other architectures are also separating features with each iteration. We show these results in Table 3.6.

Table 3.6: Classifiability of features in ConvNets and MLPs. CIFAR-10 classification accuracy of linear classifiers trained on intermediate features output by each iteration. Each cell reports the percentage of training/testing images correctly labeled.

	Layer #1	Layer #2	Layer #3	Layer #4	Layer #5	Layer #6
R-MLP	46.52/44.28	50.67/47.22	57.54/52.07	59.43/54.08	61.43/55.19	61.54/55.39
R-CNN	58.74/68.67	59.68/66.99	65.39/70.74	67.73/73.84		

Another tool for quantifying relationships between deep neural networks is feature space similarity metrics. We employ the techniques proposed by Kornblith et al. [2019] to compare models trained on CIFAR-10 and obtain a similarity score between zero and one. When we compare feed-forward ConvNets with their recurrent counterparts, the results indicate that models with the same effective depths are indeed very simlar. Specifically, the features extracted by an eight-layer feed-forward ConvNet are more similar to those extracted by its recurrent counterpart with the same effective depth (CKA = 0.843) than they are to those of feed-forward models of different depths (e.g. CKA = 0.819 for a ConvNet with 4 layers). That is, feed-forward and recurrent networks of the same effective depth are more similar to each other than they are to models of different depths.

3.5.2 Visualizing the Roles of Recurrent and Feed-forward Layers

In previous sections, we performed quantitative analysis of the classification performance of recurrent and feed-forward models. In this section, we examine the visual behavior that transpires inside these networks qualitatively. It is widely believed that deep networks contain a hierarchical structure in which early layers specialize in detecting simple structures such as edges and

textures, while late layers are tuned for detecting semantically complex structures such as flowers or faces [Yosinski et al., 2015]. In contrast, early and late layers of recurrent models share exactly the same parameters and therefore are not tuned specifically for extracting individual simple or complex visual features. Despite recurrent models applying the same filters over and over again, we find that these same filters detect simple features such as edges and textures in early layers and more abstract features in later layers. To this end, we visualize the roles of individual filters by constructing images which maximize a filter's activations at a particular layer or equivalently at a particular recurrence iteration.

We follow the approach proposed by Yosinski et al. [2015]. We also employ a regularization technique involving batch-norm statistics for fine-tuning the color distribution of our visualizations [Yin et al., 2020]. We study recurrent and feed-forward networks trained on ImageNet and CIFAR-10.

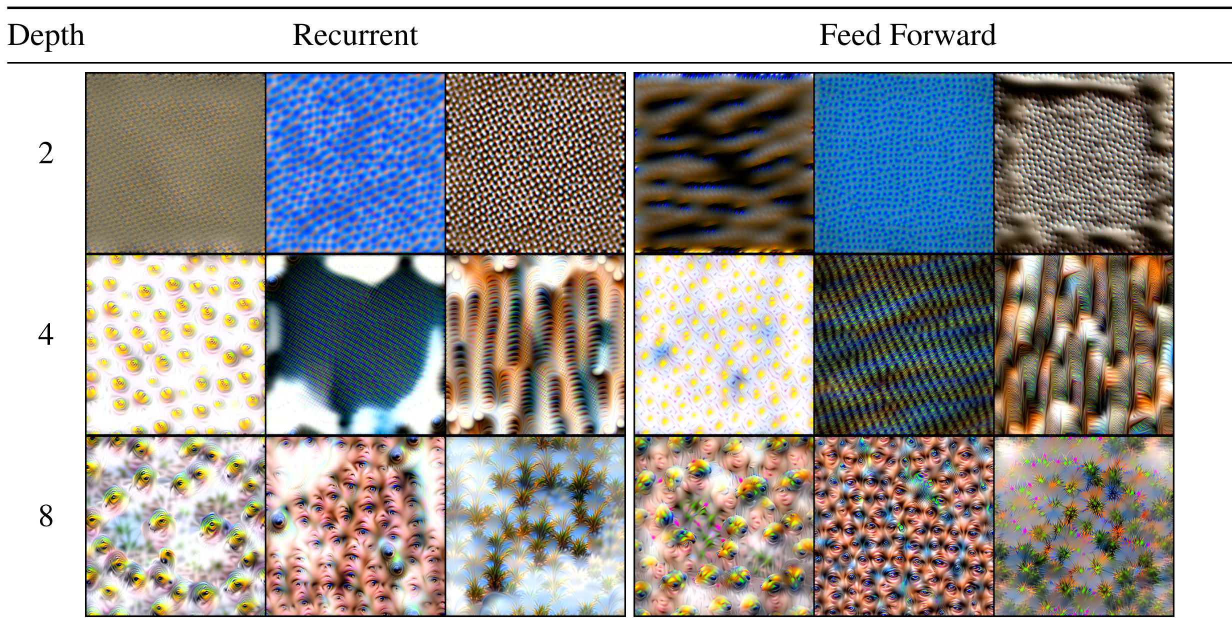

Figure 3.8: ImageNet filter visualization. Filters from recurrent model iterations alongside corresponding feed-forward layers.

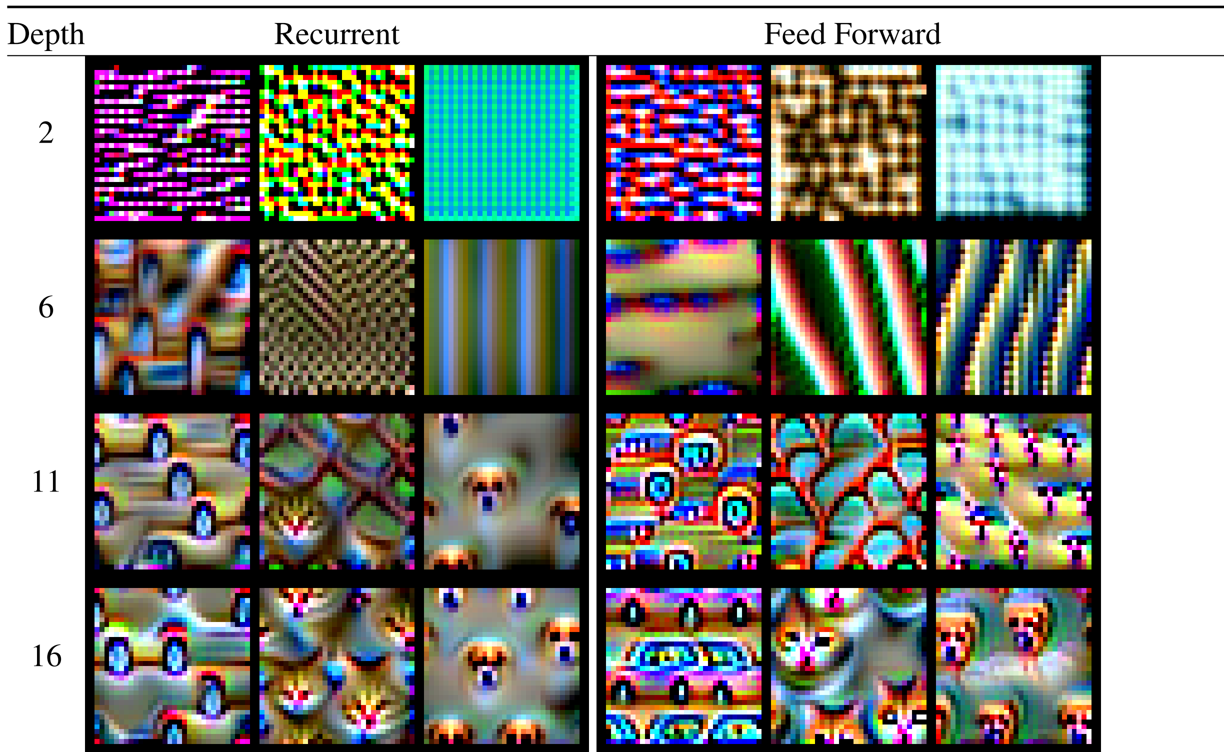

Figure 3.9: CIFAR-10 filter visualization. Filters from recurrent model iterations and corresponding feed-forward layers.

Figures 3.8 and 3.9 show the results for ImageNet and CIFAR-10 datasets, respectively. Note that in these figures, the column of visualizations from recurrent models show the input-space image that activates output for the same filter but at different iterations of the recurrent module. These experiments show qualitatively that there are no inherent differences between the filters in the shallow layers and deep layers. In other words, the same filter is capable of detecting both simple and complex patterns when used in shallow and deep layers. We show the results of more visualizations in Figures 3.10 and 3.11. We note that the first and final layers are not part of the recurrent modules.

This visualization technique is similar to prior work [Yin et al., 2020]. We also use pixel jitter as an augmentation and total variation as a regularizer. The pixel jitter moves the input image both horizontally and vertically by selecting two random numbers in $(-32, +32)$, one

for each axis. The portion of image that moves out of the viewpoint fills up the empty space on the opposite side. For total variation regularization, we use the anisotropic version in four directions: horizontal, vertical, and two diagonal directions. We use the ℓ_2-norm for computing the total variation in all directions. We find that the batch statistics from the first convolutional layer provide a strong enough signal for improving the quality through regularization. And for CIFAR-10, we optimize with a mini-batch of 64 samples of 32×32 images for 1000 iterations. For ImageNet, we use a mini-batch of 32 examples of 224×224 images for 1000 iterations.

3.6 Discussion

With the qualitative and quantitative results presented in this chapter, it is clear that the behaviour of deep networks is not the direct result of having distinct filters at each layer. Nor is it simply related to the number of parameters. If the specific job of each layer were encoded in the filters themselves, recurrent networks would not be able to mimic their feed-forward counterparts. Likewise, if the trends we see as we scale common network architectures were due to the expressiveness and overparameterization, recurrent networks should not experience these very same trends. This work calls into question the role of expressiveness in deep learning and whether such theories present an incomplete picture of the ability of neural networks to understand complex data.

In addition to comparing performance metrics, we show with visualizations that features extracted at early and late iterations by recurrent models are extremely similar to those extracted at shallow and deep layers by feed-forward networks. This observation confirms that the hierarchical features used to process images can be extracted, not only by unique specialized filters,

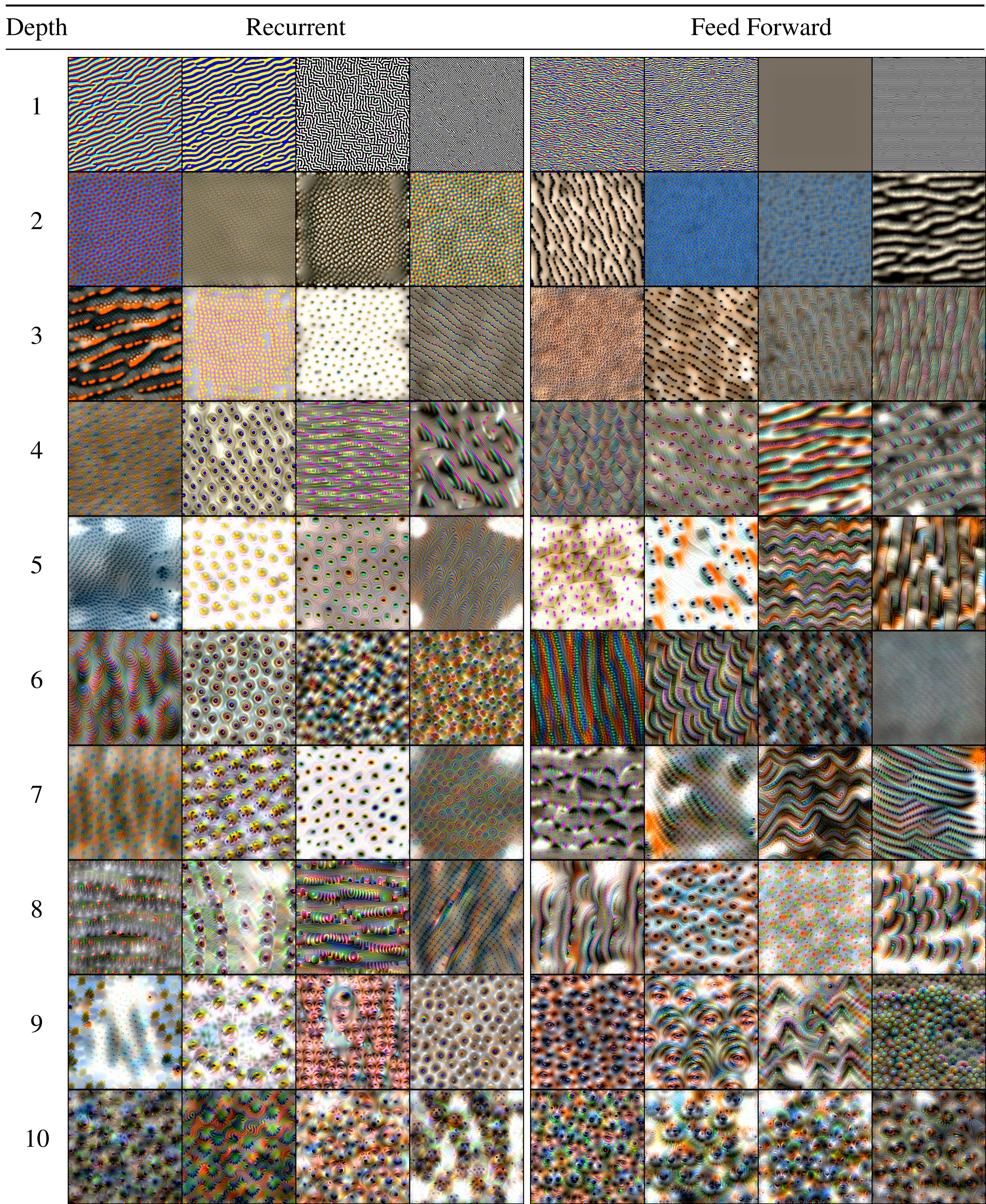

Figure 3.10: ImageNet filter visualization. Filters from recurrent model iterations alongside corresponding feed-forward layers.

but also by applying the same filters over and over again.

The limitations of our work come from our choice of models and datasets. First, we cannot necessarily generalize our findings to any dataset. Even given the variety of image data used in this project, the conclusions we make may not transfer to other image domains, like medical images, or even to other tasks, like detection. With the range of models we study, we are able to show that our findings about recurrence hold for some families of network architectures, however there are regimes where this may not be the case. We are limited in making claims about architectures that are very different in nature from the set that we study, and even possibly in addressing the same models with widths and depths other than those we use in our experiments.

Our findings together with the above limitations motivate future exploration into recurrence in other settings. Analysis of the expressivity and the generalization of networks with weight sharing would deepen our grasp on when and how these techniques work. More empirical investigations covering new datasets and additional tasks would also help to broaden our understanding of recurrence. In conclusion, we show that recurrence can mimic depth in several ways within the bounds of our work, i.e. the range of settings in which we test, and ultimately confirm our hypothesis.

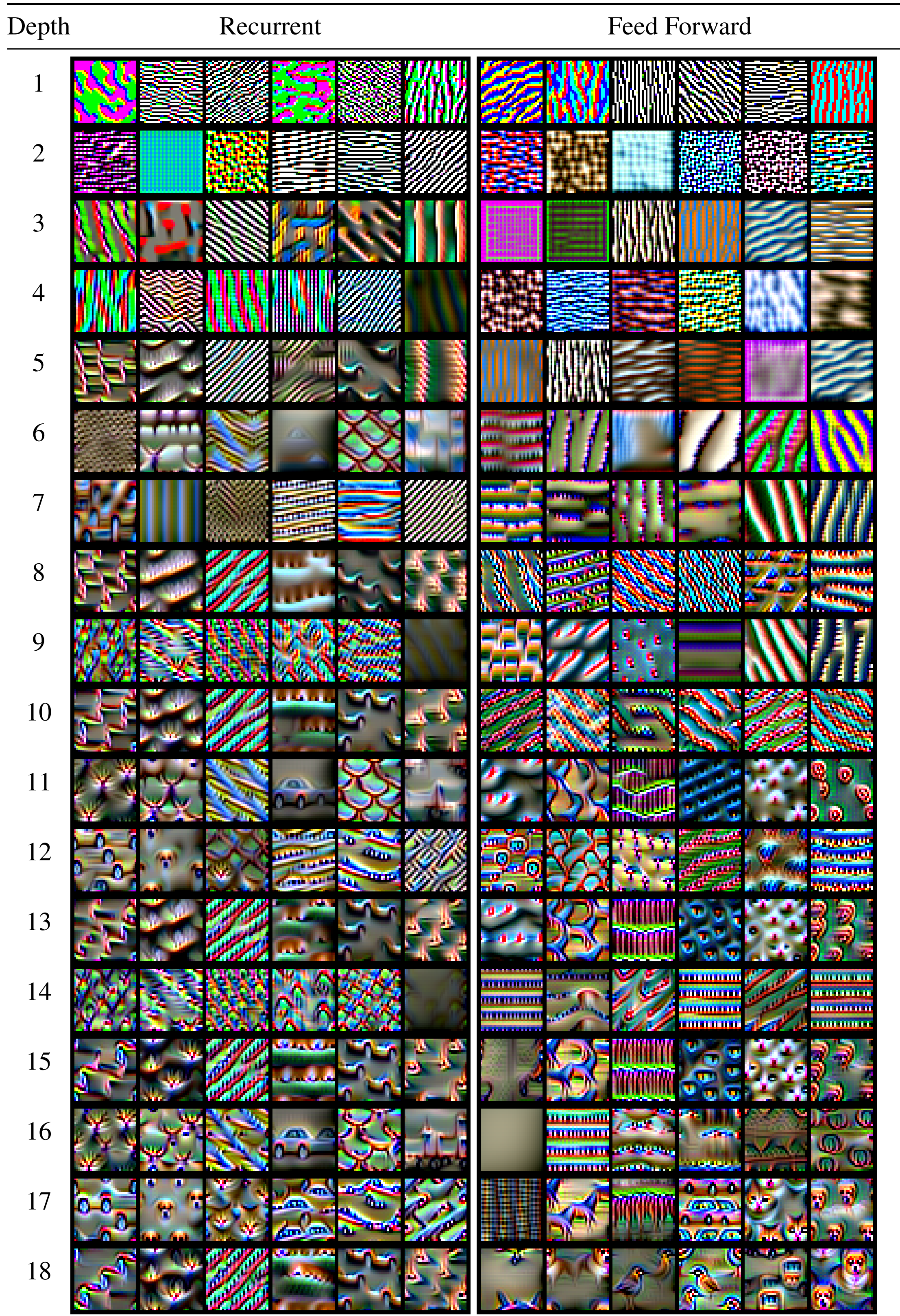

Figure 3.11: CIFAR-10 filter visualization. Filters from recurrent model iterations alongside corresponding feed-forward layers.

Chapter 4: Can You Learn an Algorithm?

Joint work with Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. Appear in NeurIPS 2021.

With our the understanding that recurrence can, in some contexts, replace depth in neural networks that is established in Chapter 3, we turn our attention to utilize recurrence for algorithmic learning. Specifically, we aim to build neural networks with recurrent modules (layers or blocks of layers) that can be iterated more to process harder examples. We know that handcrafted classical algorithms can scale to larger and more complex examples, and they do so with correspondingly more computation. In this chapter, we show that recurrent neural networks can learn scalable behaviors from data that endow them with algorithmic reasoning capabilities. We use all three datasets described in Chapter 2 as test beds to demonstrate the extrapolative power of recurrent networks.

4.1 Introduction

In computational theories of mind, an analytical problem is tackled by embedding it in "working memory" and then iteratively applying transformations to the representation until the problem is solved [Baddeley, 2012, Baddeley and Hitch, 1974]. Iterative processes underlie the

human ability to solve sequential reasoning problems, such as complex question answering, proof writing, and even object classification [Liao and Poggio, 2016, Kar et al., 2019]. They also enable humans to extrapolate their knowledge to solve problems of potentially unbounded complexity, including harder problems than they have seen before, by thinking for longer.

This work examines whether recurrent neural networks trained on easy problems can extrapolate their knowledge to solve hard problems. We find that recurrent networks can indeed generalize to harder problems simply by increasing their test time iteration budget (i.e., thinking for longer than they did at train time). Moreover, we find that the performance of recurrent models improves as they recur for more iterations, even without adding parameters or re-training in the new, more challenging problem domain. This ability is specific to recurrent networks, as standard feed-forward networks rely on layer-specific behaviors that cannot be repeated to extend their reasoning power.

The behavior we observe in recurrent networks falls outside the classical notions of generalization in which models are trained and tested on the same distribution. Because we train and test on problems of different sizes/difficulties, our training and test distributions are disjoint, and systems must extrapolate to solve problems from the test distribution. Outside the field of machine learning, computers achieve a functionally similar extrapolation ability through the use of algorithms, which encode the process required to solve a class of problems, and can therefore scale to problems of arbitrary size, albeit with longer runtime.

By training networks to solve problems iteratively, we hope to find models that encode a scalable *method* for solving problems rather than *memorizing* a mapping between input features and outputs. In short, the goal is to create recurrent architectures that are capable of *learning an algorithm*.

Our focus is on three reasoning problems that are classically solved using hand-crafted algorithms: computing prefix sums, solving mazes, and playing chess. Sequential reasoning tasks like these are ideal for our study because one can directly quantify the difficulty of a problem instance. In the case of mazes, for example, we can easily swap to a more challenging domain by increasing the size of the search space.

For each class of problems, recurrent networks are trained on a set of "easy" problems using a fixed number of iterations of the recurrent module on the forward pass. After training is complete, we assess whether our models exhibit logical extrapolation behaviors by testing them on "hard" problems, with varying numbers of additional iterations. Remarkably, models trained on easy examples exhibit little extrapolative behavior until their iteration budget is increased — generalizing to harder problems *requires* thinking deeper. Moreover, we find recurrent models tested with a sufficient number of extra iterations outperform the inflexible feed-forward models of comparable depth, often by a wide margin. Finally, we visualize the iterative behavior of the recurrent module to gain insights into the problem solving process they discover.

4.1.1 Related Work

Our investigation into generalizing from easy to hard examples builds on several bodies of work. Logical extrapolation encompasses a special kind of distributional shift. A number of existing works on domain generalization instead explore shifts, such as re-stylization and image corruptions that do not represent an increase in scale or computational complexity [Arjovsky et al., 2019, Shu et al., 2020]. Also, the basic neural architectures we use are not new and build upon prior studies of weight sharing and recurrence [Pinheiro and Collobert, 2014, Liang and Hu,

2015, Alom et al., 2018, Bai et al., 2018, 2019, Lan et al., 2020, Jaegle et al., 2021]. Networks with variable numbers of test time iterations/layers have also been studied, including variable depth networks [Graves, 2016, Huang et al., 2016, Kaya et al., 2019, Eyzaguirre and Soto, 2020].

Existing work on algorithm learning involves recurrent neural network (RNN) based approaches. For example, neural Turing machines and neural GPUs can learn simple algorithms for tasks such as binary addition and multiplication [Graves et al., 2014, Kaiser and Sutskever, 2015]. Like most RNNs, the compute budget for these methods is inextricably tied to input length. Motivated by the fact that input sequence length is not necessarily correlated with the computational burden required to solve a problem, Graves [2016] develops a method for RNNs to adaptively select a compute time limit. This work considers only sequence inputs and shows the benefits of decoupling compute budget from input length. A differentiable extension of the technique can also be applied to visual question answering [Eyzaguirre and Soto, 2020].

The above works leverage neural networks with adaptive computation budgets to speed up and strengthen inference when learning on stationary distributions. In contrast, our work studies the logical extrapolation behaviors that recurrent networks possess when both computation budgets and problem difficulties are extended beyond the train-time regime. In the domain of constraint satisfiability problems (CSPs), Selsam et al. [2018] show message passing neural networks trained on small CSPs can generalize to larger problems if more messages are passed at test-time – a similar type of extrapolation to our methods, but for a very specific problem formulation.

The particular problems we use to study this type of extrapolation include prefix sum computation and maze solving, two problems analyzed in the classical algorithms literature. For example, there are many ways to solve mazes, both classical (e.g. breadth first search) and

learned (e.g. value iteration networks), but our goal is not to develop the best solver, rather to use mazes as a test bed for logical extrapolation [Tamar et al., 2016]. Our third case study is the game of chess, which has also been the focus of much artificial intelligence work [Romstad et al., 2010-2022, Biswas and Regan, 2015, Silver et al., 2017, McIlroy-Young et al., 2020]. However, those efforts to play chess rely heavily on hand-crafted search algorithms, often paired with neural networks or opening books, and aim to play games from start to finish, both methods and goals that diverge from ours. As opposed to hard-coded algorithms, which scale by design, we are interested in studying whether learned processes can generalize from the data on which they are trained to even harder problems.

4.2 Model Architectures and Training

In all of the experiments in this chapter, we employ network architectures based on ResNets. Our feed-forward networks are slight deviations from the most commonly used ResNets, in that the width does not change except at the first layer and after the last residual block, and we do not use batch normalization. This is done so that the recurrent models, whose internal recurrent module has the same input and output dimensions, can be as similar as possible to the feed-forward ResNets. In fact, the only difference during training between a feed-forward and recurrent model of the same effective depth is that the weights are shared between the residual blocks in the recurrent models. We refer to the recurrent portion of the network as the *recurrent module*. Also, our models are fully convolutional with no fully connected heads. For solving prefix sums, which involves one-dimensional strings, we further deviate from classical ResNets by using one-dimensional convolutions.

The feed-forward prefix sum models are fully convolutional models that take in $n \times 1$ arrays. The first layer is a one-dimensional convolution with a three entry wide kernel that strides by one entry with padding by one on either end on the input. The output of this first convolution has 120 channels of the same shape as the input. The next parts of the networks are residual blocks made up of four layers with skip connections every two layers. After the residual blocks, there are three similar convolutional layers that output 60, 30, and two channels, respectively. For a network of depth d, there are $(d-4)/4$ residual blocks. The recurrent models are identical, except that all residual blocks share weights.

The feed-forward maze solving models are fully convolutional models that take in $n \times n \times 3$ arrays. The first layer is a two-dimensional convolution with a 3×3 kernel that strides by one entry and pads by one unit in each direction. The output of this first convolution has 128 channels of the same shape as the input. As above, the next parts of the networks are residual blocks made up of four layers that are identical to the first layer with skip connections every two layers. After the residual blocks, there are three similar convolutional layers that output 32, 8, and two channels, respectively. For a network of depth d, there are $(d-4)/4$ residual blocks. The recurrent models are identical, except that all residual blocks share weights.

For experiments with dilated filters, the only changes made are to the dilation of the convolutional filters and to the padding of every convolution and the values are set to maintain the output dimenssion of each layer. The chess playing models are the same as the maze models except that the first layer takes $8 \times 8 \times 12$ inputs and outputs 512 channels. None of the models used in this chapter have batch normalization or bias terms.

We measure recurrent models in terms of *iterations* and *effective depth*. An iteration is a repetition of the recurrent residual block, which contains four layers in all of the models in this

chapter. Therefore, the effective depth is equal to four times the number of iterations, plus the non-recurrent projection and head layers that sandwich the recurrent module. For example, the models used for computing prefix sums have one convolutional projection layer, followed by the recurrent block and then by a three layer convolutional head. In this case, a 10-iteration model has effective depth $1 + 10 \times 4 + 3 = 44$ layers.

For each training sample we consider, the label is an array of binary classification variables, one per input pixel. The training loss is simply the mean cross-entropy loss across output values. In general, hyperparameters were determined with the goal of finding models that train to convergence. For data augmentation, binary string inputs to prefix sum networks are approximately normalized by subtracting 0.5 from every element in the string in order to aid in training stability. Also, mazes are padded to be 32×32 pixels. All prefix sum networks are trained using the Adam optimizer with a weight decay factor of 0.0002. Because of training instability, we also apply gradient clipping at magnitude 1.0. The maze solving networks are trained with stochastic gradient descent with a weight decay factor of 0.0002 and momentum coefficient of 0.9.

Chess networks are also trained using stochastic gradient descent with a weight decay factor of 0.0002 and similarly have a momentum coefficient of 0.9. Prefix sum networks are trained to convergence with 500 epochs, maze models are trained for 200 epochs, and chess networks are all trained to convergence with 140 epochs. All prefix sum networks are trained using an exponential warm-up learning rate schedule applied over the first 10 epochs. The initial learning rate (post warm-up) is set at 0.001 and is subsequently halved at epochs 100, 200, and 300. Maze solving networks also use warm-up with a period of five epochs after which the learning rate is 0.001. The learning rate further decays by a factor of ten at epoch 175. Chess networks are trained with an exponential warm-up schedule applied over 3 epochs with an initial learning rate (post warm-

up) of 0.1 and dropped by a factor of ten at epochs 100 and 110. Finally, prefix sum models are trained with batches of 150 binary strings, maze networks with batches of 50 mazes, and chess models with batches of 300 puzzles.

4.3 Recurrent Networks Can Generalize from Easy to Hard Problems

We explore the ability of recurrent neural networks to generalize to more difficult problems simply by thinking deeper. To this end, we train models of varying effective depth on easy training examples and test them on harder problems. We find that recurrent models are even better at generalizing from easy to hard than their feed-forward counterparts. While there is only one way to test the feed-forward models, we take a closer look at what happens when the recurrent models are allowed to think deeper about the harder problems. Formally, we use more iterations of the recurrent module within the recurrent models when performing inference on test data. Across all three problem types, we find that the confidence of the model is a good surrogate for correctness. Therefore, when evaluating recurrent models, we use the output from the iteration to which the network assigns the highest confidence.

4.3.1 Prefix Sums

The first task on which we demonstrate the ability of recurrent neural networks to learn an algorithm is one from the classical algorithms literature, namely computing prefix sums. Specifically, we study the problem of computing the prefix sums modulo two of binary input strings.

When computing prefix sums, we employ models with effective depths from 40 to 68 layers. We train models on easy data consisting of 32-bit input strings and test on harder 40-

bit and 44-bit strings. In Table 4.1, it is clear that even when holding the depth constant at test time, recurrent models generalize from easy to hard better than feed-forward networks.

Table 4.1: **Extrapolating to longer input strings.** Shown here are the average accuracies of models trained on 32-bit inputs and tested on 40-bit inputs. The effective depths listed below correspond to 9, 10, and 11 iterations in recurrent models. We report average accuracy $\pm$ one standard error.

	Effective Depth (Layers)		
	40	44	48
Recurrent	24.96 ± 2.96	31.02 ± 2.56	35.22 ± 3.34
Feed-forward	22.17 ± 0.85	24.78 ± 1.65	22.79 ± 1.32

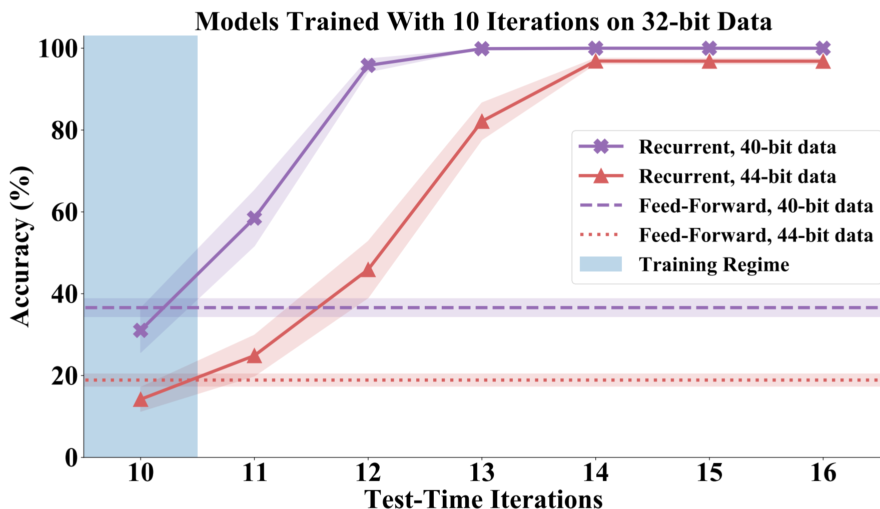

Figure 4.1: Generalizing from easy to hard prefix sums. The ability of networks to compute prefix sums on two test sets with longer input strings than were used for training (accuracy on 40-bit inputs in purple and on 44-bit inputs in red). We compare recurrent models to the best feed-forward models of comparable effective depth. The markers are at average values from several trials and the shaded regions indicate $\pm$ one standard error.

When the thought budget, or number of iterations, is increased, we see that recurrent models can get upwards of 90% of the harder testing examples correct. In Figure 4.1, we observe this large boost in the recurrent models' performance and a vast difference in the accuracy of recurrent models (with added iterations at test time) and feed-forward networks. Note that the dotted lines represent the average accuracy of the deepest feed-forward networks considered. That depth is

68 layers, or the effective depth of recurrent models with 16 iterations, and we use this baseline in the plot specifically because in the range of depths corresponding to the numbers of iterations shown, these feed-forward models achieve the highest accuracy. In other words, recurrent models trained with relatively few iterations generalize well to harder data while similar and even much larger feed-forward networks fail to generalize in the same scenario.

The generalization we see in Figure 4.1 indicates that these recurrent models learn processes that can be extended to harder problems by running for more iterations. In particular, the recurrence is both the machinery that allows for varying the depth at test time, as well as a force at training to push the model to find parameters that make progress toward a solution with each reuse.

When these results are viewed through the lens of algorithm design, one might wonder how the *receptive field*, or the number of entries in the input that determine a single entry in the output, affects these models. The feed-forward models, whose accuracies are shown in Figure 4.1, have the same receptive field as the recurrent models when tested with 16 iterations. This makes it clear that the increase in accuracy of recurrent models does not simply occur because the receptive field grows with added iterations, rather it occurs because they have learned a process that can extrapolate beyond the training distribution.

4.3.1.1 Iterative Outputs

One way to dissect the learned process and compare it to known algorithms is to plot the confidence of the model at each iteration. In Figure 4.2, we show a representative example of a network's confidence that each bit in the output is a one. Two striking observations can be made

from this figure. The first is that the model is progressing to the solution with each iteration. The second observation is that it is resolving the prefix sum in the earlier bits first and moving down the string, settling on the final bits only in the last iteration. This is remarkably similar to a naive algorithm one might implement for this task that marches from the first index until the end of the string computing prefix sums in order.

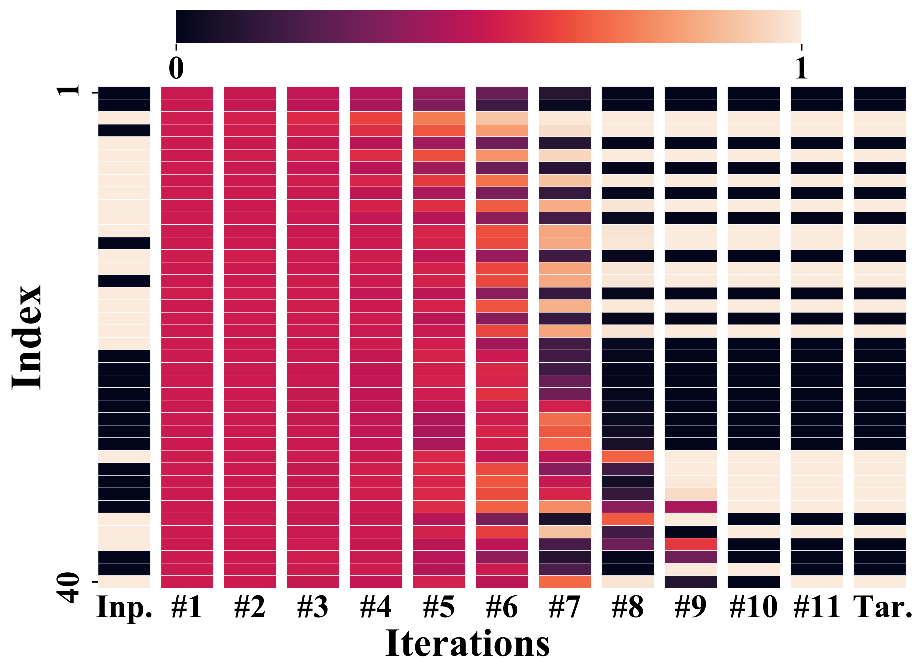

Figure 4.2: A recurrent model's output from each of 11 iterations on a 40-bit input string. Shown here is the confidence that there is a 1 at each index of the output. The first index is at the top for all vectors, the input is in the left-most column and the target is in the right-most column. The model used to produce this plot was trained with fewer iterations (10) on shorter input strings (32-bit).

4.3.2 Mazes

For maze solving, we train models on a training set composed of the easier small mazes, and we investigate the ability of networks to make the leap to larger, or harder, mazes at test time. In line with the findings above, we show two important behaviors. First, the recurrent models

make the leap from small to large mazes better than feed-forward models. Second, when allowed to think deeper, the recurrent models exhibit even higher performance.

The networks employed here are fully convolutional and have 512 channels in the internal layers. In assessing the confidence of a given output, we average each pixel's classification confidence.

Table 4.2: **The average accuracy (%) of models trained on small mazes and tested on large ones.** Over a range of effective depths, we see that recurrent models generalize to the harder mazes better than their feed-forward counterparts. Figures reflect averages over several trials $\pm$ one standard error.

	Effective Depth				
	20	24	28	36	44
Recurrent	12.66 ± 0.44	14.02 ± 0.39	19.95 ± 0.31	22.96 ± 1.03	29.72 ± 1.22
Feed-forward	7.94 ± 0.36	12.43 ± 0.50	14.67 ± 0.54	17.71 ± 0.36	22.53 ± 1.14

Table 4.2 shows that for a fixed effective depth, recurrent models always generalize to the hard mazes better than their feed-forward counterparts. Inspired by the upward trend in Table 4.2, we shift focus to deeper models. In Figure 4.3, we show that recurrent models can extrapolate to harder problems better than feed forward models. When trained on small mazes with 20 iterations (effective depth of 84 layers), these networks can solve about half of large mazes. However, when allowed to think for longer, the recurrent models can correctly solve an even higher proportion of large mazes. In fact, models trained with 20 iterations can achieve upward of 70% accuracy on large mazes using five additional iterations at test time.

Maze solving is a task for which global information is needed. We investigate how dilated filters affect model performance. Dilation is a way of changing the receptive field without adding new parameters or more depth. Table 4.3 shows that dilations lead to slight improvements, however, the benefits of recurrence are still abundantly clear. Indeed, the difference in performance

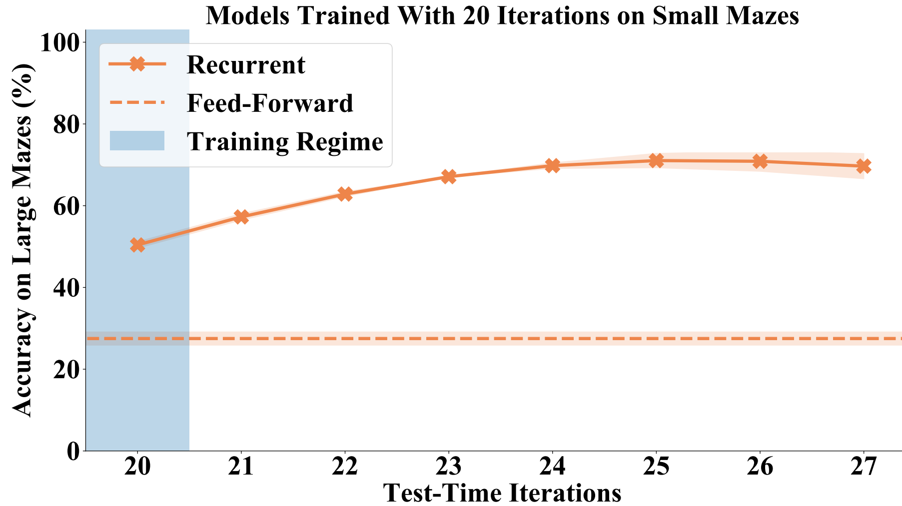

Figure 4.3: Generalizing from easy to hard mazes. We compare recurrent models to the best feed-forward models. The markers are at average values from several trials and the shaded regions indicate ± one standard error.

is even larger.

Table 4.3: **The average accuracy (%) of models with dilated filters trained on small mazes and tested on large ones.** Figures reflect averages over several trials ± one standard deviation.

	Effective Depth				
	20	24	28	32	36
Recurrent	33.60 ± 1.06	40.49 ± 1.63	33.91 ± 1.99	40.56 ± 4.34	50.50 ± 7.97
Feed-forward	19.73 ± 0.72	21.59 ± 0.22	24.18 ± 1.33	25.82 ± 0.10	26.54 ± 2.13

4.3.2.1 Iterative Outputs

The recurrent maze solving networks also produce output at every iteration. Examining this output again leads to a remarkable conclusion: these recurrent models are narrowing in on the answer with each successive iteration. In Figure 4.4, it is clear from the output on iteration four that the network has found two routes emanating from the red square. Moving through the iterations, the model refines the output, increasing the confidence for pixels on the path and decreasing the others until finally at Iteration #7, the output matches the target. It is also interesting

to observe here that the output is consistently correct for two iterations, after which a few pixels flip (Iteration #9).

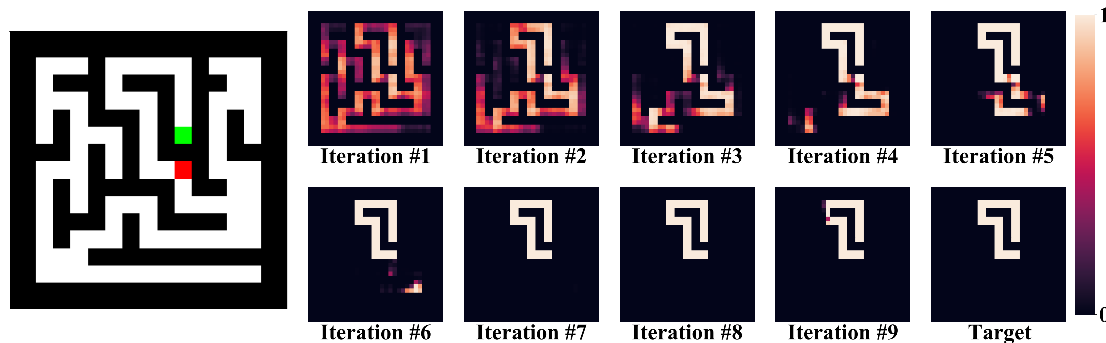

Figure 4.4: Input, target, and outputs from different iterations are shown to highlight the model's ability to think sequentially about mazes. We plot the model's confidence that each pixel belongs to the optimal path. This is a representative example from a model trained to solve small mazes in six iterations.

4.3.3 Chess Puzzles

The third dataset comprises chess puzzles, or mid-game chess boards for which we seek the best next move. Unlike the other two datasets, the state of the art for chess playing algorithms is complex and has components that use algorithms like Monte Carlo tree search as well as neural network based elements for evaluating positions [Romstad et al., 2010-2022, Silver et al., 2017]. The complicated nature of these systems provides some context for how difficult these puzzles are. What makes these puzzles particularly useful for us is that there is a predetermined best next move. A move is defined as an origin square, or the current location of the piece to be moved, and a destination square.[1] In order to generate target outputs for our models, we define a move as an 8×8 array with zeros everywhere except at the entries corresponding to the origin and destination squares which are ones.

[1]There are some cases, pawn promotions, where this information does not uniquely identify the move, and they are overlooked in this project.

We compare recurrent and feed-forward networks of effective depths from 84 layers to 100 layers that take $8 \times 8 \times 12$ arrays as input. These fully convolutional networks have 512 channels in the internal layers, and the output is $8 \times 8 \times 2$, corresponding to binary classification at each input pixel. During training, we use an average of cross-entropy losses at every pixel. When evaluating these models, however, we define the predicted move by the locations of the two highest confidence scores.

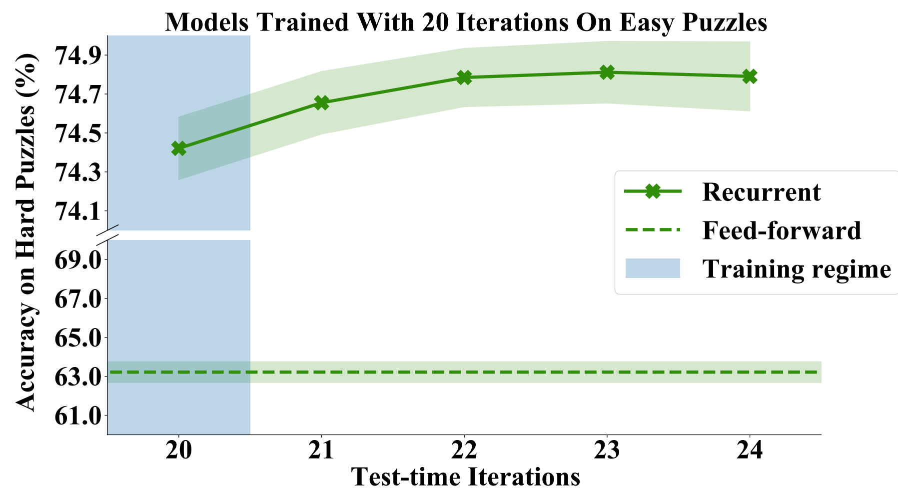

Figure 4.5: Generalizing from easy to hard chess puzzles. The ability of networks to solve harder puzzles than were used for training. We compare recurrent models to the best feed-forward models of comparable effective depth. The markers are at average values from several trials and the shaded region indicate $\pm$ one standard error.

Once again, we see that recurrent models can solve more chess puzzles than their feed forward counterparts. Furthermore, by thinking deeper at test time, recurrent models can perform even better. While the gains shown in Figure 4.5 are modest in comparison to the other two problem settings, the trend is clear – recurrent models can solve more puzzles with more iterations.

4.3.3.1 Iterative outputs

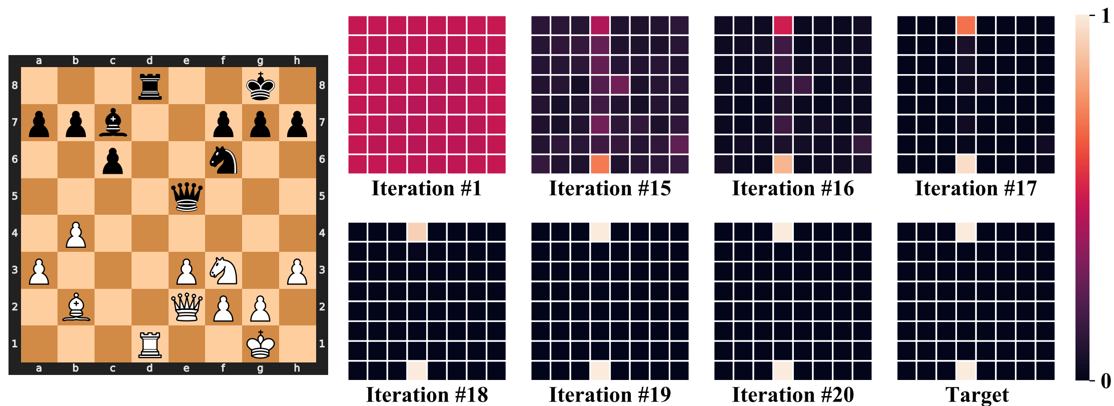

Figure 4.6: Input, target, and outputs form different iterations are shown to highlight the model's ability to think about the next move. We plot the model's confidence that each pixel is one of the two that define a move. In this example, black is to move next. For space consideration, iterations 2-14, which look like the first iteration, are left out of this plot.

While the outputs from the other two datasets show the similarity between the learned iterative process and known sum and search algorithms, extracting that insight on chess puzzles is much more difficult. Partly, this is because exploiting known search algorithms on such a large search space is hard to visualize. Nonetheless, the network's output, shown in Figure 4.6, tells a fascinating story. First, the Iteration #1 plot shows that after one iteration, we observe equal confidence at every location on the board. In the next frame, Iteration #15, it is clear that the model is considering moving the D8 rook to multiple squares, and it also considers the intuitive idea of using the E5 queen to place the white king in check on H2. With each successive iteration, the network becomes more confident – first that the rook is the correct piece to move, and then where to move it to.

4.4 Further Experimentation

4.4.1 Dilated Filters

The receptive field can be increased without adding parameters or depth by dilating the convolutional filters. When we use dilated filters to compute prefix sums, we find that we can fit the training data with N-bit sequences with fewer than N layers – a behaviour that is not possible with non-dilated convolutions. This suggests that the algorithm learned by our models is informed by the receptive field. In other words, since the final entry in a prefix sum does require global information (and therefore, in order for a neural network to compute the final entry it needs a complete receptive field), the use of extra iterations on longer sequences meshes with the algorithmic analysis. When we contextualize our models by analyzing the way they scale, it is reasonable to find that networks with non-dilated convolutions scale linearly with the input length. This also motivates future work to study whether neural networks can be designed to learn an algorithm that scales with the square root of the problem size, or even logarithmically.

4.4.2 Deeper Feed-forward Models

Significantly deeper feed forward models cannot generalize from easy to hard as well as recurrent ones. For example, we extend the experiments in Figure 4.1 to feed-forward models with depths 132, 264, and 528. These models all fit the training data (32-bits) well, but the best performance on 40-bit data is 53% and on 44-bit data is 27% – neither is attained by the deepest models lending confidence that the performance would not increase with even more depth.

4.4.3 In-distribution Tests

For in-distribution test accuracy (on small/easy cases) on prefix sums, both classes of models achieve >99% and on mazes, both achieve >97% (using the same number of iterations as used during training). When we apply more iterations, we see only slight drops for in-distribution test accuracy (tenths of a percent), but no improvement. On chess puzzles, the in-distribution test accuracies are 92% for the recurrent models and 84% for the feed-forward networks (though there is a gap in performance, it is smaller than the gap observed for out-of-distribution testing).

We also train and test on the harder datasets to measure in-distribution accuracy. Here, we find there is little difference between recurrent and feed-forward models. As examples, both classes of models for prefix sums and mazes achieve essentially 100% on unseen data. Specifically, both recurrent and feed-forward models achieve >99% on prefix sums and >97% on mazes.

Finally, we train models on larger instances and test them on smaller cases, and we find that they can solve smaller examples in fewer iterations than were used at training. For example, when models are trained to compute prefix sums on 44-bit data in 10 iterations, they can get 100% of the 16-bit test examples correct after only four iterations.

4.4.4 Prefix Sum Experiments on Other Datasets

4.4.5 Even Harder Chess Puzzles

4.5 More Visualizations

Additional visualizations of intermediate outputs, along with input and target examples from all three datasets are presented below.

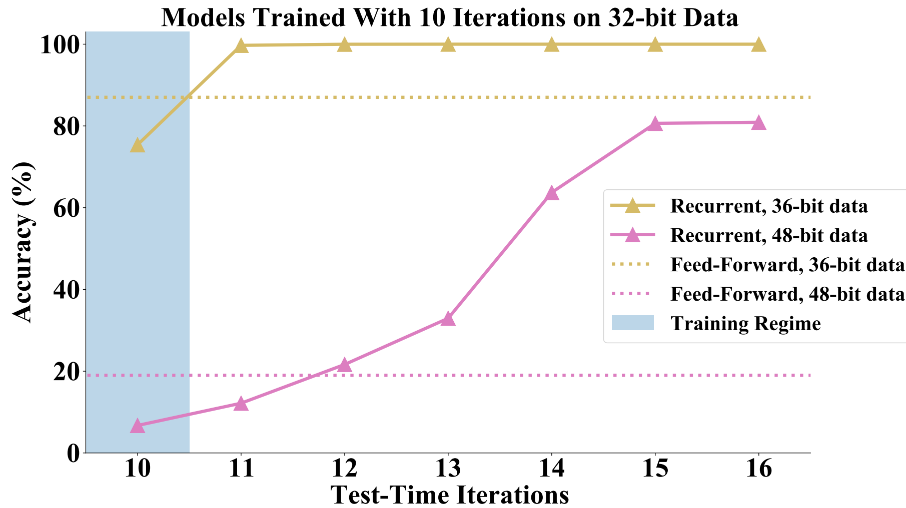

Figure 4.7: Generalizing from easy to hard prefix sums. The ability of networks to compute prefix sums on two additional test sets with longer input strings than were used for training (accuracy on 36-bit inputs in yellow and on 48-bit inputs in pink). We compare recurrent models to the best feed-forward models of comparable effective depth.

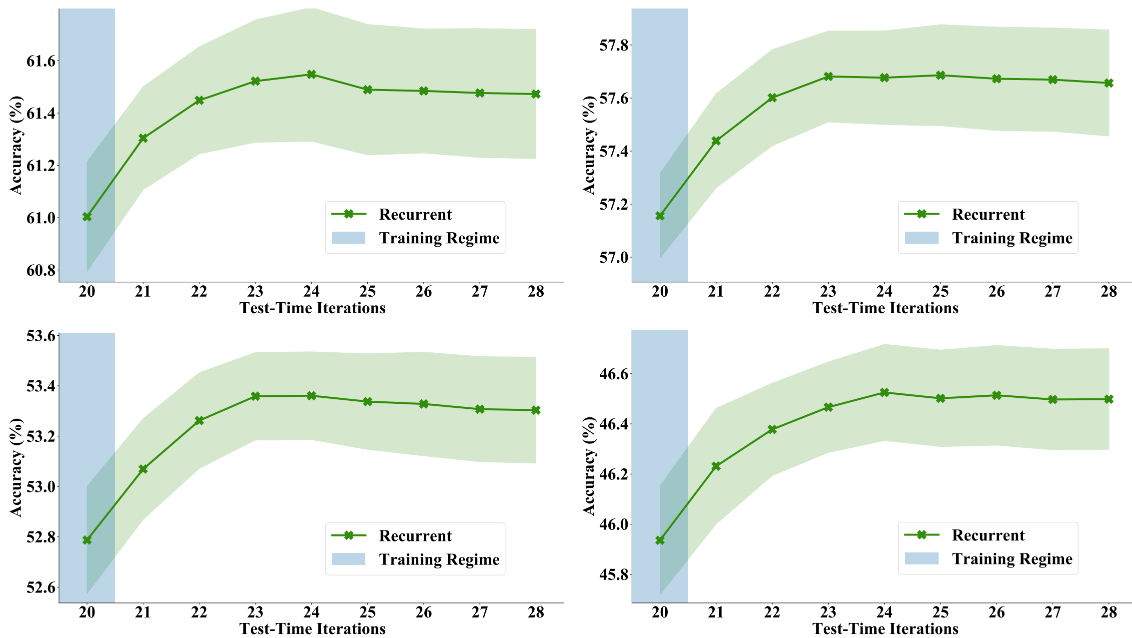

Figure 4.8: Generalizing from easy to hard chess puzzles. The ability of networks to solve harder puzzles than were used for training. We trained models on the first 600,000 puzzles and we show their performance with extra iterations on puzzles with index 800,000 to 850,000 (top left), 850,000 to 900,000 (top right), 900,000 to 950,000 (bottom left), and 100,000 to 150,000 (bottom right).

4.5.1 Prefix Sums

We show a recurrent model's output from each of 11 iterations on 40-bit input strings. Shown below is the confidence that there is a 1 at each index of the output. The first index is at the top for all vectors, the input is in the left-most column and the target is in the right-most column. The model used to produce these plots was trained with fewer iterations (10) on shorter input strings (32-bit).

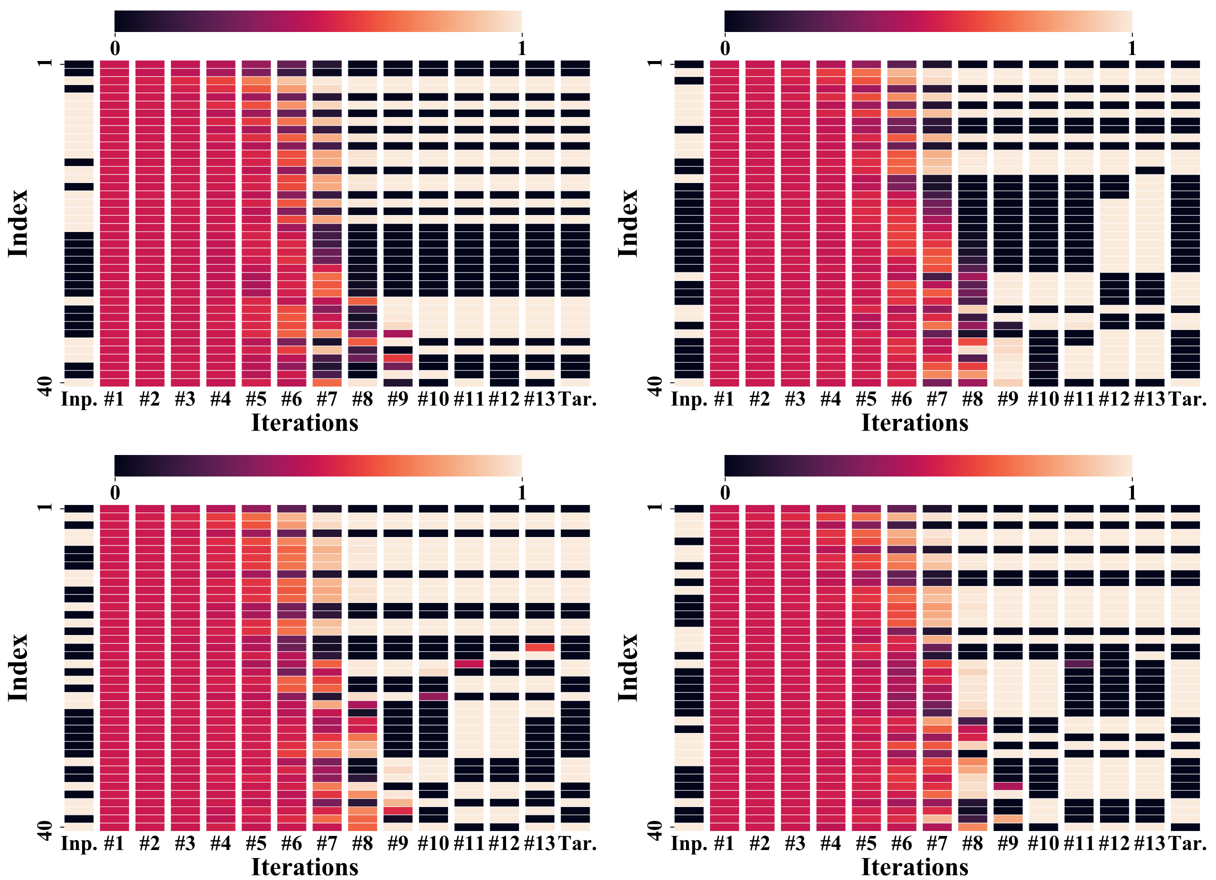

Figure 4.9: Prefix sum intermediate outputs.

4.5.2 Mazes

We show inputs, targets, and outputs form different iterations to highlight the model's ability to think sequentially about mazes. We plot the model's confidence that each pixel belongs to the optimal path. Below are several representative examples from a model trained to solve small mazes in six iterations.

4.5.3 Chess Puzzles

We show inputs, targets, and outputs from different iterations to highlight the model's ability to think about the next move. Below, we plot the model's confidence that each pixel is one of the two that define a move.

4.6 Discussion

More than an answer, the results and conclusions in this chapter are posing a question: Can learned models behave like classical algorithms in the way they generalize to harder or larger problems?

Our discussion of this question and our observations begins with delineating the limitations of our work. The first major limitation is that we do not propose a definitive answer. Rather, using representative cases, we demonstrate that recurrence can help neural models make the leap from easy training data to hard testing examples. A more subtle limitation of our work lies in how we split the data by difficulty. For prefix sum computation and for mazes, the classical algorithms approach to measuring problem complexity is tied to problem size, so in those settings we make intuitive easy/hard splits. With chess however, the issue is much more complex. Should puzzles

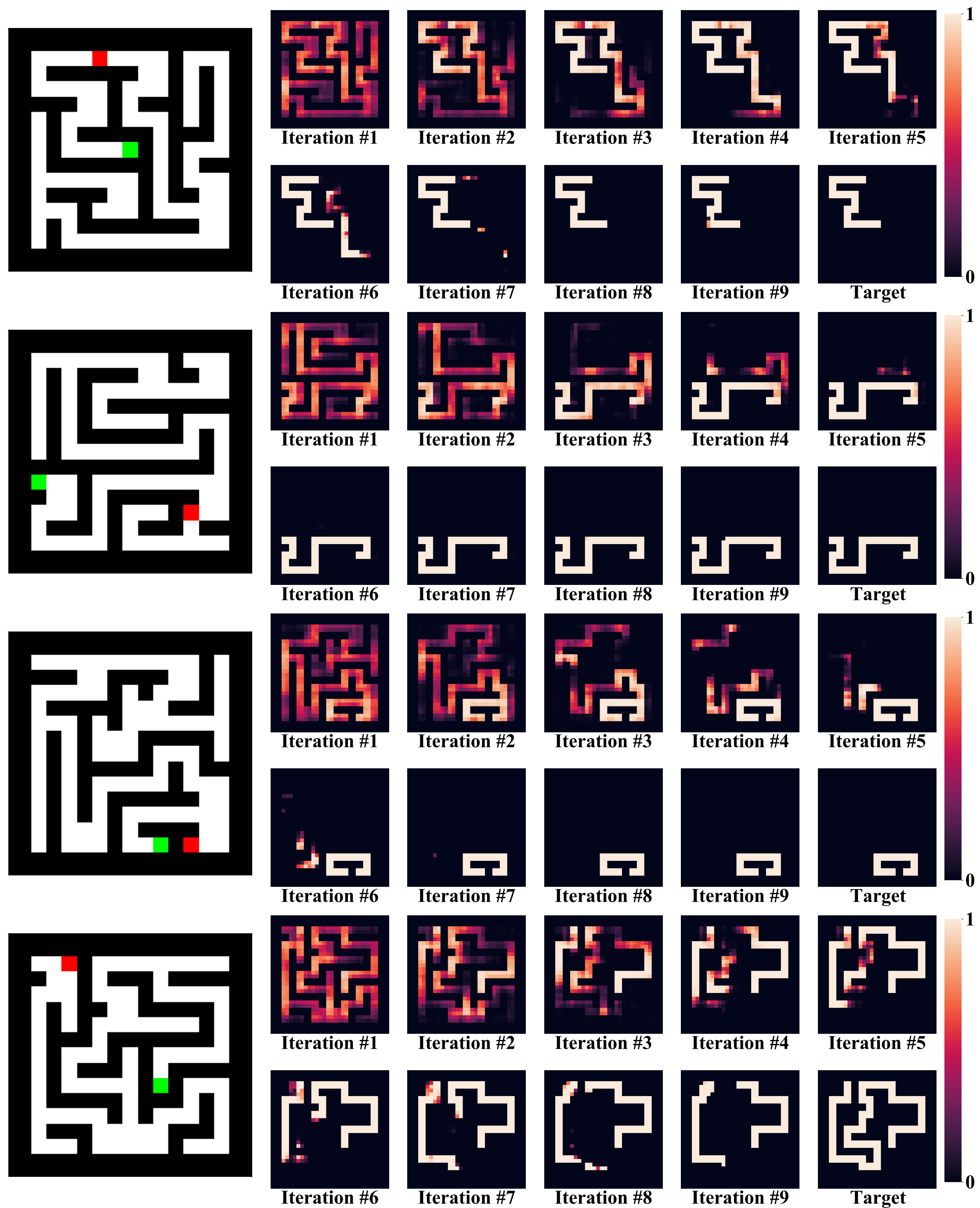

Figure 4.10: Maze model intermediate outputs.

with higher ratings require more memory or more computation? We carry out our work assuming the answer is yes, but we remain open minded to the possibility that the ratings assigned by Lichess may be weak surrogates for algorithmic complexity of each puzzle. In short, chess is an

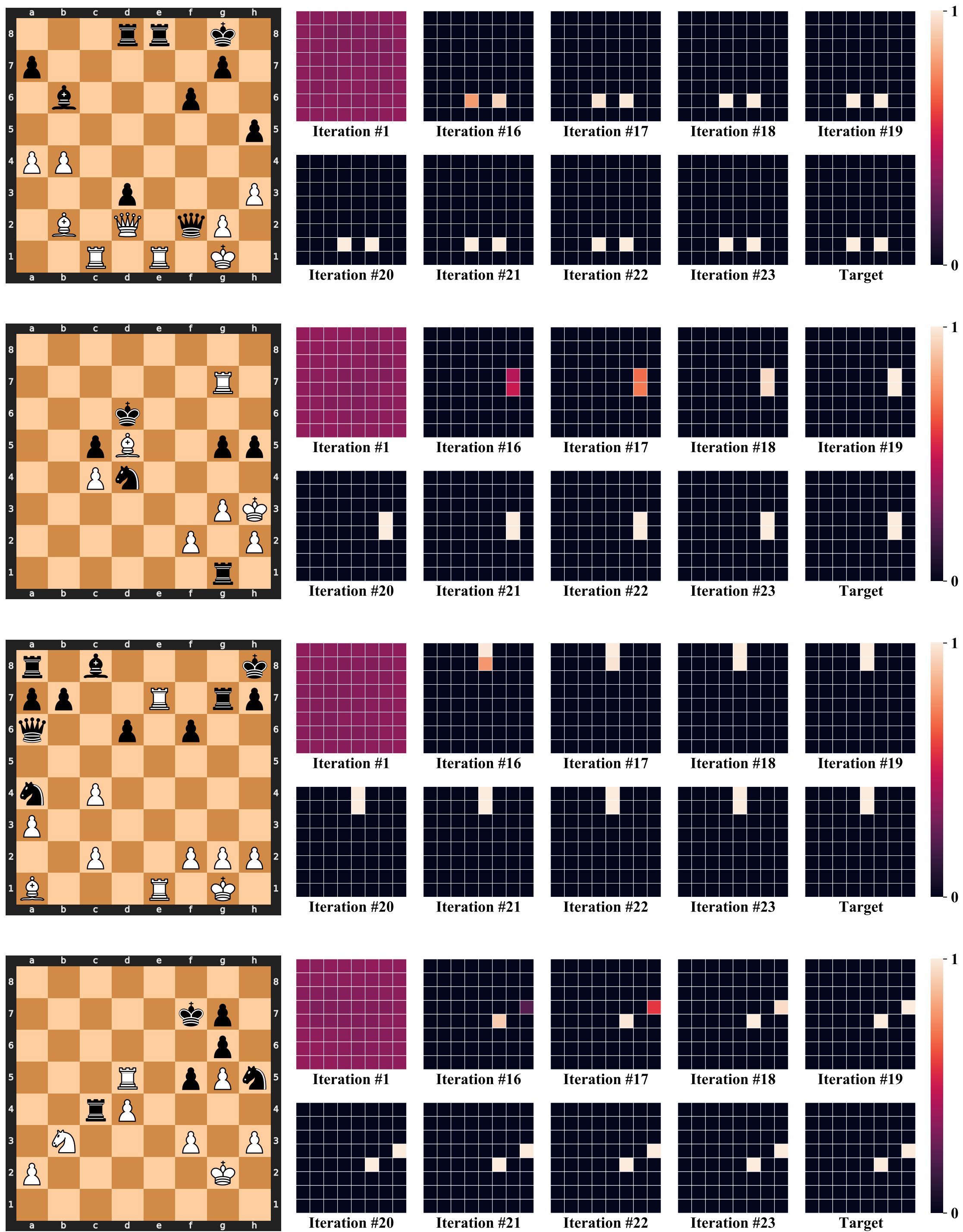

Figure 4.11: Chess model intermediate outputs.

extremely difficult domain to analyze.

The observations above indicate that iterative models can learn processes that generalize

beyond the training distribution with more iterations. One exciting use case is when the testing distribution is inaccessible, a setting where training on the harder distribution itself would be impossible. Many real life scenarios demand exactly this type of problem solving, from robots deployed in the real world after training in simulation to humans who spend a lot of time practicing on easy math problems only to spend years on difficult unsolved questions.

On a conceptual level, the recurrent model behavior we show is analogous to the human behavior of manipulating representations in working memory; in this analogy, the recurrent block performs the transformations and the activations it generates are the memory. It is not the goal of the experiments here to suggest that iterative models use mechanisms similar to those in a human brain. Nonetheless, it is exciting to see, even in a proof of concept setting, neural models that appear to deliberate on a problem until it is solved and can extend their abilities by thinking for longer.

This raises a question that motivates the experiments in the following chapter. Can we build neural networks that can think for even longer? Is it feasible to have models whose performance only increases with added compute time? Humans who are given more than enough time to solve a maze, will not suddenly get it wrong after arriving at the right answer, perhaps this awareness of when to stop thinking can be built into networks like the ones we study here.

4.7 Conclusion

In this chapter, we demonstrate that neural networks are capable of solving sequential reasoning tasks and then extrapolating this knowledge to solve problems of greater complexity than they were trained on. These recurrent models are largely inspired by the classical theory of

mind, in which the brain iteratively applies primitive strategies to solve complex problems over time [Baddeley, 2012]. Our models are recurrent versions of popular architectures and we acknowledge that variations to the model architecture may be helpful. Thus, we leave an in depth investigation into other neural network designs for future work.

Interestingly, the resulting models excel at solving problems that are classically solved by hand-crafted algorithms; prefix sums are computed using reduction trees, mazes are classically solved by depth/breadth first search, and chess is solved by Monte-Carlo tree search. Even with the advances in machine learning that we have today, hand-crafted algorithms still play a role in state-of-the-art reasoning systems. A prominent example is AlphaZero, which plays board games using Monte-Carlo tree search algorithms assisted by a learned pruning function [Silver et al., 2017]. While moving away from this paradigm that includes hand-crafted elements is highly ambitious, this work suggests that it may be possible to train gameplay systems without building them on top of a hand-crafted tree search engine. In other words, it may be possible to machine-learn these algorithmic behaviors end-to-end.

Chapter 5: Logical Extrapolation Without Overthinking

Joint work with Arpit Bansal, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. Appeared in NeurIPS 2022.

In Chapter 4, we showed how recurrence can be employed to extrapolate from easy training data to hard test problems. The techniques described in that chapter have a major short coming: their performance deteriorates with large numbers of iterations. In this chapter we introduce novel techniques to c scalable processing behavior in neural networks without overthinking.

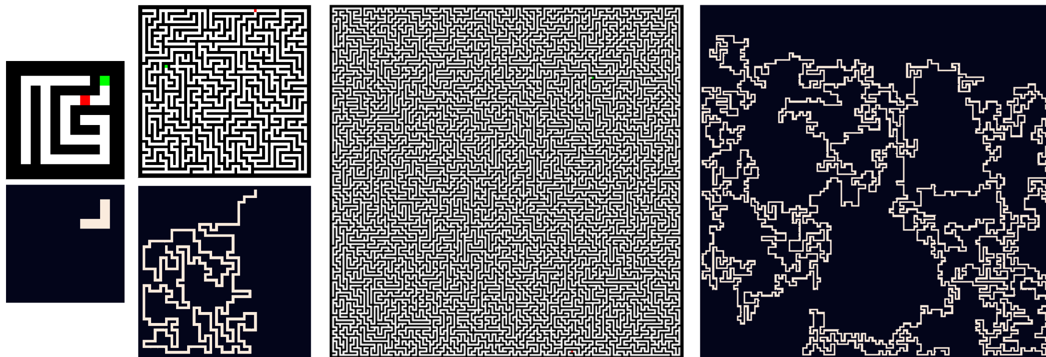

Figure 5.1: A 'thinking' network trained on 9×9 mazes and their solutions (left) autonomously synthesizes a scalable algorithm. By running this algorithm for longer, it reliably solves problems of size 59×59 (middle), and 201×201 (right) without retraining. Standard architectures, and even existing primitive thinking models, as described in Chapter 4, fail to tolerate this domain shift.

5.1 Introduction

Humans solve complex logical reasoning problems through *logical extrapolation* – they assemble simple logical primitives into complex strategies. For example, a person taught to prove simple lemmas can in turn prove more complex theorems simply by expending more cognitive effort.

Neural networks have achieved great success at pattern matching tasks, often exceeding human performance, but they struggle to solve complex reasoning tasks in a scalable, algorithmic way. Recently, Deep Thinking systems have been proposed as a way to represent and learn scalable reasoning processes using recurrent neural networks [Schwarzschild et al., 2021b]. The word 'thinking' in this context refers to sequential processing to solve discrete/logical problems. These systems train recurrent models (networks that recycle parameters between layers) to solve reasoning problems. Unlike traditional feed-forward models, which are limited in the complexity of problems they can solve by their finite depth, the effective depth of recurrent models can be expanded after training simply by iterating the recurrent unit for longer.

When trained properly, thinking systems learn scalable algorithms for solving classes of problems. After training to solve small/easy problem instances with few recurrent iterations, the algorithm is then extended to run for more iterations at test time. In doing so, the system can achieve algorithmic extrapolation, solving problems of greater difficulty than those in the training set.

Until this section, the level of algorithmic extrapolation observed in thinking systems has been quite modest. For example it has been demonstrated that a system trained on 9×9 mazes can extrapolate to solve a 13×13 maze. These systems fail to achieve greater extrapolation because

of a problem we call *overthinking*; recurrent systems, when extended too far outside their training regime, often deteriorate and fail to produce interpretable outputs.

In this chapter, we design purpose-built neural architectures and specialized training loops to make it possible to train systems that do not suffer from overthinking and instead converge to a fixed point when iterated for thousands of iterations. By doing so, we are able to build thinking systems that exhibit extreme algorithmic extrapolation behaviors, and leap from solving small/simple training problems with tens of iterations to solving large and complex problem instances at test time using thousands of iterations.

We experiment on benchmark problems for measuring extrapolation behavior. These tasks include computing prefix sums, finding optimal solutions for two-dimensional mazes, and solving chess puzzles [Schwarzschild et al., 2021a]. Our architectures and training routines significantly outperform existing methods on all tasks in the benchmark suite. Additionally, we demonstrate that iterative methods for these reasoning problems are susceptible to overthinking, a problem that is overcome by our new architectures and training loops.

Our contributions can be summarized as follows:

- We provide a recurrent architecture for algorithmic extrapolation, in which the problem input is concatenated directly to the feature stack of certain layers in the recurrent thinking module. This prevents the problem instance from being forgotten if deep features become noisy, corrupted, or lossy.
- We develop a new training routine that incentivizes recurrent networks to make incremental improvements towards a solution, improving the feature representation after each iteration. This training process removes information about how many times the recurrent module

has been applied, preventing the network from learning iteration-specific behaviors and instead allowing models to learn scalable behaviors that can be iterated indefinitely for extrapolation.

- We analyze the overthinking problem and show that our models overcome this phenomenon. In some cases, the algorithms learned by the proposed networks appear to be capable of solving problems of arbitrary size, despite training only on very small problem instances.

Our improvements in performance on the easy-to-hard benchmark datasets can be categorized in several ways. First, our models yield uniformly higher accuracy across the most difficult tasks used for testing in previous work. Second, we show that our models can extrapolate to much harder/larger examples than are considered in previous work, where the prior methods generalize poorly, if at all. Lastly, we show that our models do not forget solutions in settings where previous models overthink.

5.2 Related Work

Preliminary work [Schwarzschild et al., 2021b] (and in Chapter 4 of this dissertation) shows that simple recurrent architectures, when trained to solve various reasoning problems, can exhibit algorithmic extrapolation while their feed-forward counterparts cannot. In this section, we contextualize this approach amongst prior work on algorithm learning, adaptive neural models, and logical extrapolation.

Algorithm learning describes models that learn scalable processes from data. Early works on this topic study the ability of recurrent neural networks (RNNs) to process input strings of arbitrary lengths [Gers and Schmidhuber, 2001, Schmidhuber et al., 2007]. More recent work by

Graves et al. [2014] introduces neural Turing machines designed to mimic programmable computers, and Kaiser and Sutskever [2015] propose a parallel version inspired by massively parallel graphical processing units. These methods, and various improvements to them, show promising results on bit string to bit string tasks, including copying inputs and adding integers, and even demonstrate the ability to generalize from shorter training strings to longer ones at testing [Graves et al., 2014, Kaiser and Sutskever, 2015, Freivalds and Liepins, 2017]. Since they are based on classical RNNs, however, the amount of computation they perform is directly linked to the length of the input string, which prevents them from executing more or less computation independently of the input size (or corresponding to the difficulty of the problem). Moreover, classical RNNs are often trained incrementally to produce one bit at a time, rather than synthesizing an algorithm for solving an entire problem end-to-end. This makes it difficult to apply them to problems where the solution cannot be decomposed into incremental parts (e.g., chess).

Constraint satisfiability problem (CSP) solving networks disentangle the amount of work from the input size [Selsam et al., 2018]. Specifically, message passing neural networks can execute more passes to solve harder CSPs [Selsam et al., 2018]. These systems are specific to the problem of constraint satisfaction for boolean expressions, but they are an early demonstration of scalable algorithmic behavior.

Adaptive neural networks are designed to expend varying amounts of computation on different inputs, thus overcoming the limitation of classical RNNs. Self delimiting neural networks use one neuron to determine when to stop updating the hidden state in RNNs, and in doing so, they perform more or less computation for each token in an input sequence [Schmidhuber, 2012]. Adaptive compute time (ACT) is an algorithm that provides RNNs with a halting unit, which estimates the probability that computation should continue. This algorithm penal-

izes ‘ponder time’ during training to encourage the network to solve problems quickly [Graves, 2016]. Eyzaguirre and Soto [2020] exhibit strong performance on visual question answering by introducing a differentiable version of ACT. Adaptive transformer-based language models also exist, notably Universal Transformers and Depth-Adaptive Transformers, which utilize ACT to determine the work required for each input [Dehghani et al., 2018, Press et al., 2021]. Similarly, iterative residual networks like NAIS-Nets repeat blocks in stages and perform well on image classification [Ciccone et al., 2018]. All of these works test their methods *in-distribution*, i.e. where the training and testing data are sampled from the same distribution; logical extrapolation outside the training domain is not considered.

Logical extrapolation describes the task of generalizing to test sets which comprise more computationally complex samples than the training data. Nuamah [2021] claims that “neural network models with end-to-end training pipelines ... cannot, on their own, successfully perform algorithmic reasoning,” and instead proposes a hybrid hand-crafted and learned approach. Similarly, Palm et al. [2018] propose recurrent relational networks, which operate on graphs by iteratively passing messages. They also claim that classical architectures lack the inductive bias to reason about relationships between objects. Several recent works call these claims into question. Schwarzschild et al. [2021b] employ recurrent networks based on weight-sharing architectures, which can be made deeper at test time independent of the input size. These systems exhibit logical extrapolation behavior in several domains. More details on these methods are discussed in Section 5.3 as our algorithms build on these directly. Banino et al. [2021] reformulate the halting unit in ACT leading to probabilistic RNN models with improved performance called PonderNet, and importantly their method outperforms ACT on logical extrapolation tasks for prefix sums.

The ‘thinking’ systems proposed by Schwarzschild et al. [2021b] depart from classical re-

current networks for text, which learn from step-by-step supervision to produce output tokens one at a time. In contrast, 'thinking' systems autonomously synthesize a scalable algorithm end-to-end with no supervision over what each algorithmic step should do. Even more importantly, they can be applied to solve complex problems that are difficult or impossible to decompose by hand. In other words, sequence-to-sequence models, even adaptive RNNs, have a severe limitation in that the problem needs to be represented as a sequential input and that each token in the output is generated by the same function. 'Thinking' systems, on the other hand, provide a mechanism for domains where this type of decomposition is unnatural, difficult, or even impossible.

5.3 Methods

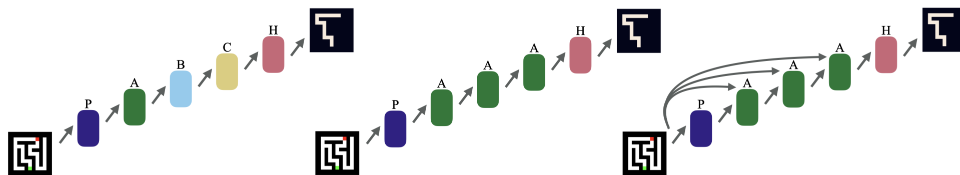

Figure 5.2: Architecture schematics. Left to right: A feed-forward network, a network containing three recurrent blocks (in green) that share weights, and a recurrent network with recall.

We begin with some terms and definitions. We study networks that share weights across blocks of layers during training. For example, instead of three distinct residual blocks, a single residual block is repeated three times (see Figure 5.2 for a graphical depiction). At test time, networks trained this way can be made "deeper" to extend their compute budget simply by repeating the block more times. We refer to the number of layers applied in a recurrent network as its "depth," and this quantity grows as the number of recurrent iterations increases. The number of feature maps produced by each layer (or the number of filters in a convolutional layer) is referred to as its "width."

More formally, let r be a function representing a recurrent block, e.g. a ResNet block [He et al., 2016], and let r^n denote n recurrences of that function, e.g. $r^2(x) = r(r(x))$. Let ϕ denote a feature map, or an output of r, and let $\phi_n = r^n(x)$. We also consider an initial "embedding function" denoted by p, which projects an input instance into feature space, and also a final "output head" denoted by h, which maps features to outputs. A Deep Thinking (DT) network with m iterations of the recurrent block can then be expressed as follows.

$$f(x; m) := h(r^m(p(x))) \tag{5.1}$$

In our systems p comprises a convolutional layer followed by a ReLU, r is a single four-layer residual block, and h is a set of three convolutional layers with ReLUs after the first and second. We fix m during training and compute gradients for optimization by backpropagating through the unrolled network. Then, m can be increased for testing, allowing these networks to increase their processing power and solve larger and harder problems. Below, we consider two new approaches for achieving improved extrapolation: recall architectures and a modified optimization process.

5.3.1 Recall Architectures

When humans think for a long time to solve a problem, we often stop to reread the question or review the task at hand. We improve DT architectures to periodically recall the input exactly. We incorporate this capability into architectures by concatenating the input problem to the features output from each instance of the recurrent block.

Popular architectures in computer vision typically incorporate skip connections that similarly pass information from earlier layers forward. In fact, empirical evidence suggests that skip

connections, for example in highway networks, ResNets, and DenseNets, stabilize training [Srivastava et al., 2015, He et al., 2016, Huang et al., 2017]. Our architectural modification is driven by the intuition that a noisy training process creates thinking networks that are imperfect and may leak or distort information over time as features are iteratively fed back through the recurrent unit thousands of times. *Recall* allows the system to reproduce any missing or damaged features, and makes it impossible to "forget" the problem being solved. To formalize this architectural change, adding *recall* to the network can be expressed using the notation defined above as follows.

$$
\begin{aligned}
&f_{\text{recall}}(x; m) := h(r^m_{\text{recall}}(p(x), x)), \text{ where} \\
&r_{\text{recall}}(\phi, x) := r([\phi, x]) \text{ and } r^m_{\text{recall}}(\phi, x) = r_{\text{recall}}(r^{m-1}_{\text{recall}}(\phi, x), x)
\end{aligned}
\tag{5.2}
$$

Whereas the input to r at iteration k is usually ϕ_{k-1}, with recall, the input to r at iteration k is $[\phi_{k-1}, x]$, or the concatenation of the input with the feature map output by the previous recurrence. We add a single convolutional layer to map the input $[\phi_{k-1}, x]$ to a feature map of the same shape as ϕ_{k-1}. We refer to DT networks with concatenating skip connections as DT-Recall models.

The particular architectures in this section are tuned versions of the recurrent models described previously. In all the models, the first layer p is a convolutional layer with 3×3 filters (or filters of length three in the 1D case) followed by a ReLU non-linearity, which projects the input into feature space. Each filter strides by one, and inputs are padded with one unit in each direction. The number of filters is specified per dataset in Table 5.1 as the "width" of the network. The internal blocks are standard residual blocks with four convolutional layers (followed by ReLUs) and skip connections every two layers [He et al., 2016]. These blocks share weights in recurrent networks and are simply repeated with distinct parameters in feed-forward models. Models with

recall have additional convolutional layers that map the concatenated inputs and features $[\phi, x]$ to the shape of the feature maps ϕ. The feature maps have the same spacial dimension as the input and w channels, where w denotes the width of the model. The final block h is composed of three convolutional layers with decreasing widths (specific numbers are in the Table 5.1), and ReLUs after the first two. The third and final convolutional layer in h has two channel outputs used for binary pixel classification.

Table 5.1: Model architectures for all main experiments. Note, we perform ablations where we change the width or the maximum number of iterations and those parameters are indicated where appropriate.

Dataset	Width	# Channels in h layers
Prefix Sums	400	400, 200, 2
Mazes	128	32, 8, 2
Chess	512	32, 8, 2

5.3.2 Promoting Forward Progress Through Optimization

We propose a training objective to encourage the system to incrementally make progress from any starting point. We do this by inputting a problem instance and running the recurrent module for some random number of iterations. We then take the output of this process, restart the recurrence in the network with these features as if iterations had just begun (discarding gradients from the initial iterations), and train the model to produce the solution after a random number of additional iterations.

This incremental training process has two benefits. First, it trains the network to continue improving the quality of partial solutions, even when they contain errors or distortions that creep in from running many iterations. Second, by choosing features from a random iteration to serve as the initial state for the training step, we discourage the network from internally counting iterations

and learning iteration-specific behaviors, such as behaviors that get executed only on iteration five, for example. Rather, the network is encouraged to learn iteration-agnostic behaviors that are effective at any stage of the problem solving process.

Algorithm 1 Incremental Progress Training Algorithm

Input: parameter vector θ, integer m, weight α
for batch_idx = 1, 2, ... **do**
  Choose $n \sim U\{0, m-1\}$ and $k \sim U\{1, m-n\}$
  Compute ϕ_n with n iterations w/o tracking gradients
  Compute $\hat{y}_{\text{prog}}$ with additional k iterations
  Compute $\hat{y}_m$ with new forward pass of m iterations
  Compute $\mathcal{L}_{\text{max_iters}}$ with $\hat{y}_m$ and $\mathcal{L}_{\text{progressive}}$ with $\hat{y}_{\text{prog}}$.
  Compute $\mathcal{L} = (1-\alpha) \cdot \mathcal{L}_{\text{max_iters}} + \alpha \cdot \mathcal{L}_{\text{progressive}}$
  Compute $\nabla_\theta \mathcal{L}$ and update θ
end for

In our implementation, we randomly sample the number of iterations used to generate a partial solution, n, and the number of training iterations, k, budgeted for the network to improve this partial solution. We then update the network's parameters to minimize loss after $n + k$ total iterations when it starts with the partial solution. This is done by detaching the recurrent module's output after n iterations from the computation graph before computing the gradient of the loss at iteration $n + k$. The process above is an analog of truncated backpropagation through time [Jaeger, 2002], with a random starting and end point. During training, we ensure that the sum of n and k is less than a fixed maximum number of iteration m, which we call the *training regime*. The incremental loss described above is added to the standard loss computed with a full forward and backward pass through the unrolled m-iteration network.

The training loop and computation of the loss are given in Algorithm 1. The incremental progress term is referred to as $\mathcal{L}_{\text{progressive}}$ (or *progressive loss*), and the contribution to the loss from the fully unrolled network is denoted by $\mathcal{L}_{\text{max_iters}}$.

Prefix sum problems with n bits are input to the model as a vector $x \in \{0,1\}^n$, and the output is denoted by $\hat{y} \in \mathbb{R}^{n\times 2}$. The target output is $y \in \{0,1\}^n$. We compute the loss as follows.

$$\ell(\hat{y}, y) = -\frac{1}{n}\sum_{i=0}^{n-1} \log \frac{e^{\hat{y}[i,y_i]}}{e^{\hat{y}[i,0]} + e^{\hat{y}[i,1]}} \tag{5.3}$$

where $[\cdot,\cdot]$ indexes the output array. Therefore, for a batch of B instances, the total loss is

$$\mathcal{L} = \frac{1}{B}\sum_{b=0}^{B-1} \ell(\hat{y}_b, y_b). \tag{5.4}$$

This loss applies to all three problem types we consider. Note that a maze represented by an $n \times n \times 3$ input has n^2 pixels and the loss can be averaged over those pixels. The same applies to chess, where there are always 64 pixels.

5.3.3 Datasets

We evaluate our methods on the benchmark problems available in the Python package `easy-to-hard-data` (and Chapter 2 of this dissertation), which generates reasoning problems of various difficulties. The three problems considered are computing prefix sums, finding the optimal path in a two-dimensional maze, and solving chess puzzles. We briefly review the input and output structures for each problem and refer the reader to Schwarzschild et al. [2021a] (or Chapter 2) for more detail, including the data generation process. Note that the architectures we consider are fully convolutional, and produce outputs of the same dimension as their inputs. Furthermore, the training and testing datasets we consider have labels of the same dimension as their inputs. Therefore, a network trained on inputs of one size can then trivially be applied to

inputs of a different size.

We begin with the toy problem of computing prefix sums modulo two. The inputs and targets for this problem consist of bit strings. The ith bit of the target is the mod two sum of all bits prior to and including the ith bit of the input. We control the "difficulty" of the problem by changing the length of the bit string. Note that computing prefix sums of greater length is known to require a greater number of sequential operations [Ladner and Fischer, 1980]. All of our training is done on 32-bit strings and we explore the behavior of our models on longer strings, even showing strong performance on 512-bits.

The mazes we consider are two-dimensional square images where the walls are black, the permissible paths are white, and the start and end positions are denoted with red and green squares. The targets for this problem are maps of the same dimension as the input maze, but with ones on the optimal path and zeros elsewhere. We make more challenging datasets by increasing the size of the mazes. All training in our work is done on 9×9 mazes. Despite this small training size, we benchmark extrapolation performance on mazes of size 59×59, and observe models solving mazes as large as 201×201 with high success rates.

The chess puzzles we consider are mid-game boards represented by twelve 8×8 planes indicating the positions of all 12 types of pieces (there are six distinct piece classes and two colors on a chess board). The goal of this task is to find the best next move, and each target encodes this information in an 8×8 array with zeros everywhere except the origin and destination of the piece to be moved, which are populated with ones. The chess dataset is sorted by difficulty rating, as determined by Elo scores computed via human trials on Lichess.org. Problems are sorted by difficulty and we train on the first 600K easiest puzzles and test our models on indices ranging from 700K to 1.1M.

Our experiments have two categories: (I) Testing extrapolation at extreme leaps in problem difficulty/size and (II) Testing scenarios that emphasize the overthinking trap and how we avoid it. The following two sections describe each of these experiments, respectively.

5.4 Extreme Extrapolation

In each problem domain, we show that our models with recall and trained with progressive loss perform best.[1] We compare our models to a baseline of DT models trained with $m = 30$ and problem-specific widths (detailed below), which are both wider and deeper than the models previously studied. However, to provide a fair comparison, we use consistent values for these parameters across our experiments since we find that all architectures benefit from additional width and depth. We also compare to feed-forward models without weight sharing of the same effective depth as a 30-iteration DT network, which are ResNets [He et al., 2016]. We include ablation studies lending credence to our claim that both the architecture and the loss modifications are instrumental in boosting extrapolation power.

Prefix Sums Though prior DT systems show extrapolation behavior on strings with up to 48 bits, we use a 512-bit testing set to illustrate the extent of our performance improvements over prior work. We train our models on 32-bit strings with a maximum of only $m = 30$ recurrent iterations (indicated in the figures as the shaded *train regime* section), 400 convolutional filters per layer, and incremental loss weight $\alpha = 1$ when progressive loss is used.

In Figure 5.3, we show that without our techniques, DT networks are unable to solve very long binary strings, achieving 0% accuracy on 512-bit data, while DT-Recall models with pro-

[1]Note that the DT models presented in this section use a maximum confidence exit rule for fair comparison. This exit rule marginally mitigates overthinking in the regime initially tested by Schwarzschild et al. [2021b]. Models with our new architecture and/or loss do not require this rule and are presented with default outputs.

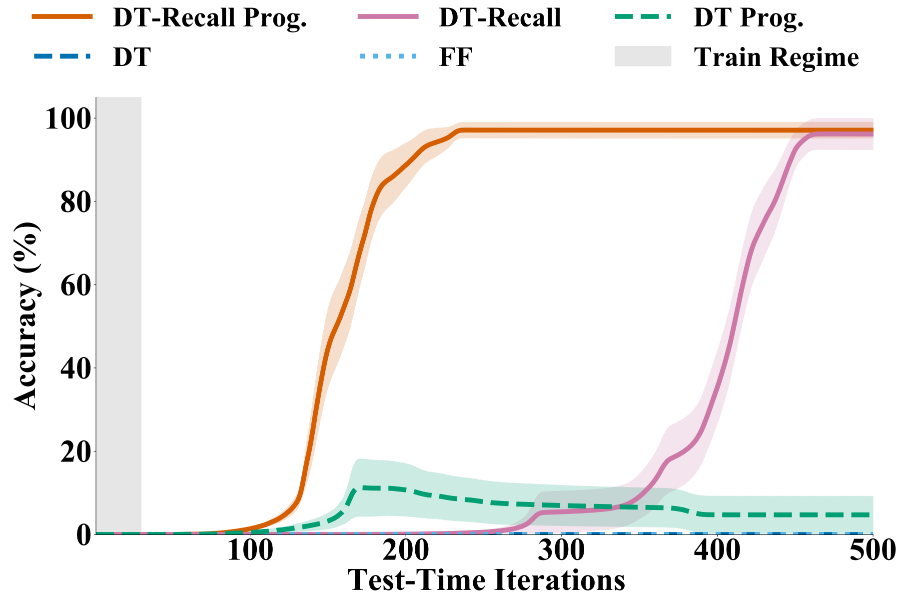

Figure 5.3: **Prefix sum models trained on 32-bit inputs extrapolate to 512-bit data.** The value of our recall and progressive loss is clear by how quickly and accurately our models solve this very large problem.

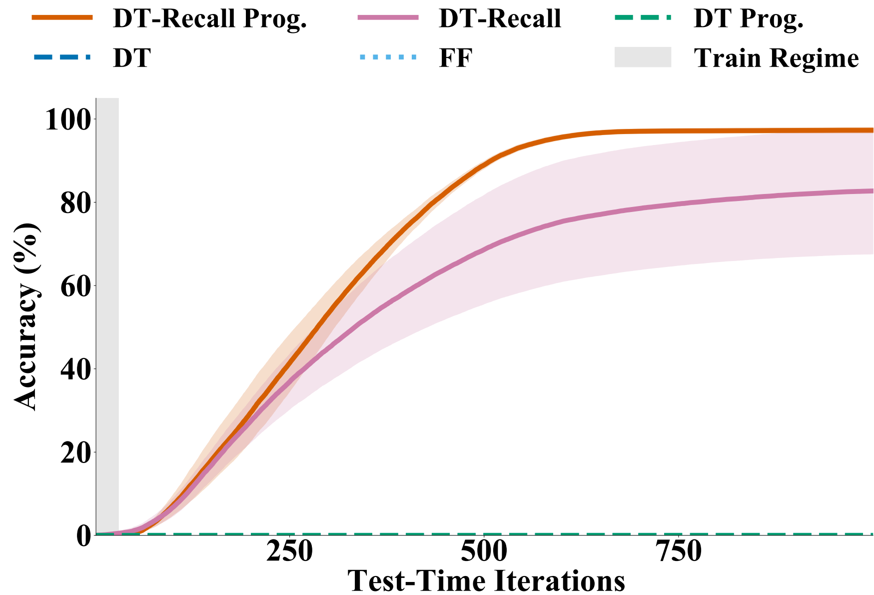

Figure 5.4: **Maze solving models trained on 9×9 inputs extrapolate to 59×59 problems.** Mazes this large cannot be solved without recall, and furthermore progressive loss leads to more accurate models.

gressive loss can solve more than 97% after approximately 200 iterations.

To better understand the individual effects of our proposed approach, we perform an abla-

tion study. Figure 5.3 also makes clear that DT-Recall networks trained *without* progressive loss achieve the same final accuracy as models trained *with* it; but they require approximately twice the number of iterations to get there. Because our proposed objective succeeds in making models iteration agnostic, solutions are often found sooner than DT models that are trained to solve 32-bit problems in 30 iterations specifically. DT models trained with progressive loss outperform vanilla DT networks, however neither of these models (without recall) can solve more than a handful of 512-bit testing examples.

One other study in this domain reveals the importance of randomly setting n in Algorithm 1. To show this, we modify the algorithm slightly, to always use $n = 0$, rather than randomly choosing that value. While this change seems minor, and indeed makes a small impact on the results, it does lower the accuracy. Specifically, DT networks with recall on average solve 90.27% of 512-bit test samples when trained with the $n = 0$ loss, and with randomly sampled values of n, as described in Algorithm 1, they achieve 97.12% accuracy. From this ablation experiment, we conclude that randomly sampling these values is, in fact, beneficial.

Mazes As with the prefix sum problem, we can improve performance on hard mazes by combining incremental training and recall architectures. In particular, while Figure 5.3 may not convince the reader that our proposed loss is critical since DT-Recall models without it perform very well, with more complex data a drastic difference emerges.

We show in Figure 5.4 that on the significantly harder test set of 59×59 mazes, our models exhibit strong algorithmic extrapolation, while previous methods, both feed-forward and DT systems, completely fail. Not only do our models achieve a higher peak accuracy, but they do not overthink, as can be seen by the flat spans in the curves in Figure 5.4. In fact, we can push these systems to their limits (and the limits of our hardware) and find that our models can solve 74%

of the 201×201 mazes.[2] See Section 2.2.2 for examples of 201×201 and 801×801 mazes, which our models solve using 2,000 and 20,000 iterations, respectively (the equivalent of 10,004 and 100,004 convolutional layers in depth).

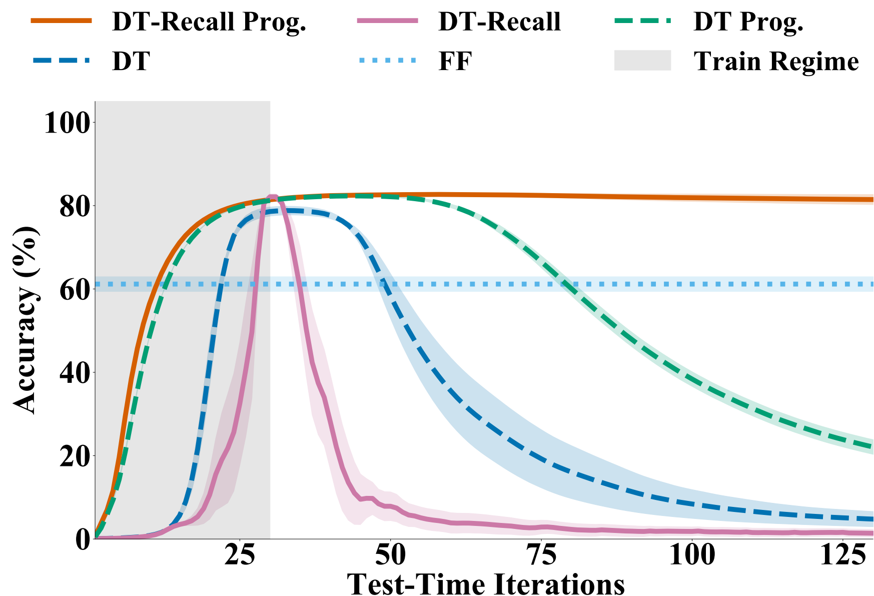

Figure 5.5: **Chess models trained on the first 600K easiest puzzles extrapolate to 600K-700K.** Recall and progressive loss are required to retain accuracy with many iterations.

The best models in Figure 5.4 are DT-Recall models trained with the maximum iteration value set to 30 and a weighting in the loss of $\alpha = 0.01$. In order to see how critical recall is in overcoming the overthinking problem, we show in the same figure that without recall all models fail to extrapolate. The benefit of progressive loss is also highlighted by the fact that DT-Recall models with progressive loss achieve on average 97% accuracy while DT-Recall models without progressive loss only reach an average of 83%. Interestingly, in this domain, we find that the best model has $\alpha = 0.01$ in the loss, a much smaller weight than we use for prefix sums.

Chess Puzzles We further find that using recall and progressive loss yields notable im-

[2] The seemingly modest performance of 74% accuracy should be understood in context. Of the 100 mazes in this test set, 20 have one pixel (out of 166,464) wrong and no single maze has more than seven mistakes.

provement on chess puzzles. In Figure 5.5, we see that our techniques lead to a 4% accuracy improvement compared to networks without them. We show that either removing recall or training without the incremental progress loss will hurt performance. Our best models are DT-Recall networks with 512 convolutional filters in each layer trained with a maximum of 30 iterations and a weight of $\alpha = 0.5$. Moreover, their accuracy is preserved as the number of iterations increases, while that of the other DT networks decays seriously after about 70 iterations – the sign of overthinking and the subject of the following section.

Tables 5.2, 5.3, and 5.4 show the peak accuracy of each of the curves presented in the plots in Section 5.4.

Table 5.2: The peak accuracy and corresponding test-time iteration number for prefix sum solving model performance curves in Figures 5.7 and 5.3.

Tested on 48-bit Strings			
Model	α	Peak Iter.	Peak Acc. (%)
DT	0.0	42	94.61 ± 1.19
DT	1.0	27	97.73 ± 1.80
DT-Recall	0.0	46	99.97 ± 0.01
DT-Recall	1.0	26	99.96 ± 0.02
FF	0.0	30	27.15 ± 2.56

Tested on 512-bit Strings			
Model	α	Peak Iter.	Peak Acc. (%)
DT	0.0	-	0.00 ± 0.00
DT	1.0	171	11.26 ± 6.90
DT-Recall	0.0	466	96.19 ± 3.73
DT-Recall	1.0	237	97.12 ± 1.88
FF	0.0	30	0.00 ± 0.00

Table 5.3: The peak accuracy and corresponding test-time iteration number for maze solving model performance curves in Figures 5.7 and 5.4.

Tested on 13×13 Mazes			
Model	α	Peak Iter.	Peak Acc. (%)
DT	0.00	40	85.59 ± 2.81
DT	0.01	38	86.08 ± 3.96
DT-Recall	0.00	94	99.94 ± 0.03
DT-Recall	0.01	70	99.88 ± 0.05
FF	0.00	0	38.22 ± 5.28

Tested on 59×59 Mazes			
Model	α	Peak Iter.	Peak Acc. (%)
DT	0.00	-	0.00 ± 0.00
DT	0.01	-	0.00 ± 0.00
DT-Recall	0.00	999	82.72 ± 15.14
DT-Recall	0.01	984	97.30 ± 0.68
FF	0.00	-	0.00 ± 0.00

Figure 5.6 shows the test performance of chess models on even harder test sets. It is inter-

Table 5.4: Peak accuracy and iteration number for chess puzzle performance curves in Figure 5.5.

Tested on puzzles 600K-700K			
Model	α	Peak Iter.	Peak Acc. (%)
DT	0.0	32	78.81 ± 1.04
DT	0.5	43	82.32 ± 0.10
DT-Recall	0.0	29	82.12 ± 0.69
DT-Recall	0.5	57	82.69 ± 0.27
FF	0.0	30	61.17 ± 1.76

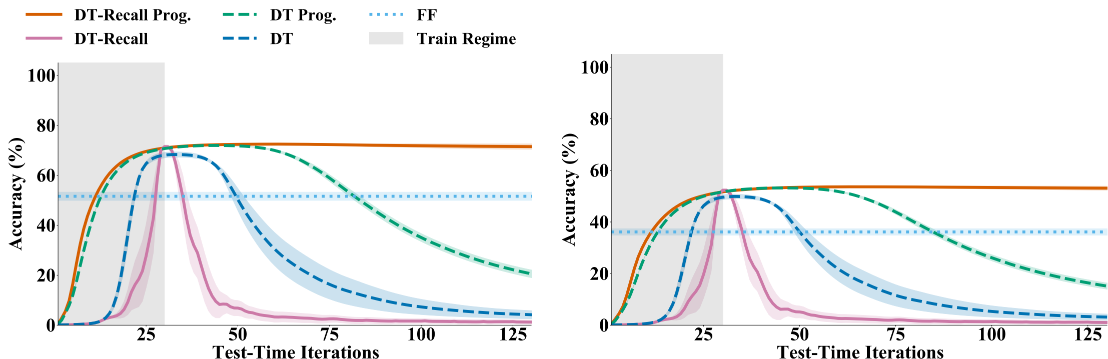

Figure 5.6: Chess performance when tested on puzzles with indices from 700k to 800k (above) and from 1M to 1.1M (below).

esting to note that recall and our loss both still help dramatically, but there is an apparent ceiling on generalizing from the easy puzzles to much harder ones. We leave investigation into this phenomenon for future work.

5.5 The Overthinking Problem

Deep Thinking networks boast impressive capabilities to solve harder problems by thinking deeper, but they are prone to overthinking. In particular, by testing models on data closer in complexity to the training data we can better compare our methods to prior work. Figure 5.7 shows that some recurrent networks that can perform algorithmic extrapolation may collapse entirely when performing too many iterations.

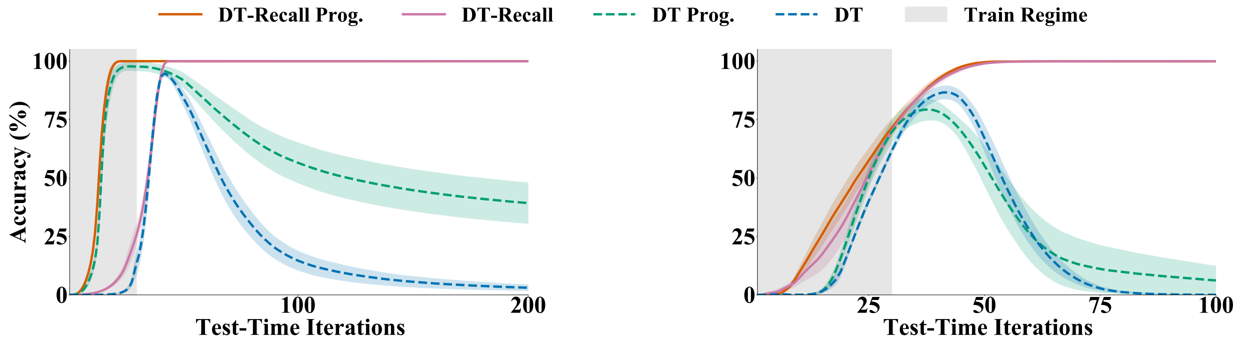

Figure 5.7: **Left:** Prefix sum models trained on 32-bits tested on 48-bit inputs. **Right:** Maze models trained on 9×9 mazes tested on 13×13 mazes. In these regimes, we can see that the weaker models can extrapolate to slightly harder problems, but importantly only models with recall avoid the overthinking trap.

DT-Recall models do not suffer from the sharp decline in accuracy as the number of iterations increases. In Figure 5.7, the decay in each dashed curve and the stability with high iteration counts of the solid curves shows that recall is critical to avoid overthinking. See Figure 5.5 for an example where progressive loss is needed too. The monotonic increase in accuracy with added iterations in our models is practically useful, as it allows for pre-defining a large iteration number at which to terminate the recurrence rather than having to carefully choose a stopping iteration to maximize performance while avoiding degradation.

One observation that can be made from these results is that the overthinking problem often seems to disappear when skip connections are added to provide networks with an uninterrupted view of the input. Another way to describe this is that our models seem to converge to a fixed-point solution as they iterate rather than becoming unstable. This property is desirable and using representative models trained to solve mazes, we explore how robust models are when we manipulate the features during the thinking process. In Section 5.5.4, we present similar findings for prefix sums.

5.5.1 Manipulating Feature Maps

First, we investigate sensitivity to adding noise in the feature maps before concatenating the inputs. We examine model behavior when we add Gaussian noise (with mean zero and standard deviation one) to the features after one iteration of maze solving. Models with recall can still solve 13×13 mazes even when we perturb the features, but models without recall cannot.

Next, we ask whether the initial feature maps (after the projection layer p) carry any important information. We test this by replacing the feature maps with zeros after 50 iterations – at this point the model has solved the maze and by annihilating the features we remove all information from p. In this case, our DT-recall models naturally regenerate their features and recover to solve the problem again, indicating that the learned algorithm is able to find a solution using the input without the initial projection. See Figure 5.10.

5.5.2 Manipulating Inputs

With the notion that our models may embody a convergent process, we turn our attention to investigating how the networks determine when to stop manipulating the representation. Perhaps there is something in the feature maps output by late iterations that tells the network to stop working. We can test this in two ways. We perturb the input (which we concatenate onto the features) after some number of iterations – first subtly, then by swapping it with an entirely different example.

To start, we flip single bits in the input string for prefix sums. We explore the response of the model to flipping bits at different indices after 50 iterations (when the models have already solved the initial 48-bit problem). The left panel of Figure 5.8 shows recovery time as a function

of index of the flipped bit. We observe that our models can indeed recover from single bit flips, and recovery time decays linearly with index. Higher indices are closer to the end and affect fewer bits in the output than lower ones.

Similarly, with mazes, we change the input by moving the end of the maze two steps closer to the start. We use this new input concatenated with features generated after 50 iterations of solving the original maze. In this case, we see in Figure 5.11 that models with recall self-correct and solve the new problem in very few iterations.

The last way we test the hypothesis that networks are continually comparing their solution to the problem instance is by partly solving Maze "A," and then swapping the features with those obtained from 50 iterations of trying to solve Maze "B." In other words: if we concatenate the input problem (A) with the features from iteration 50 corresponding to a different maze (B), which maze will the network solve? Clearly, a system without recall will solve maze B. However, with recall, networks will recover and pull the features back to representing a solution for maze A. The center panel of Figure 5.8 shows the effect of swapping feature maps.

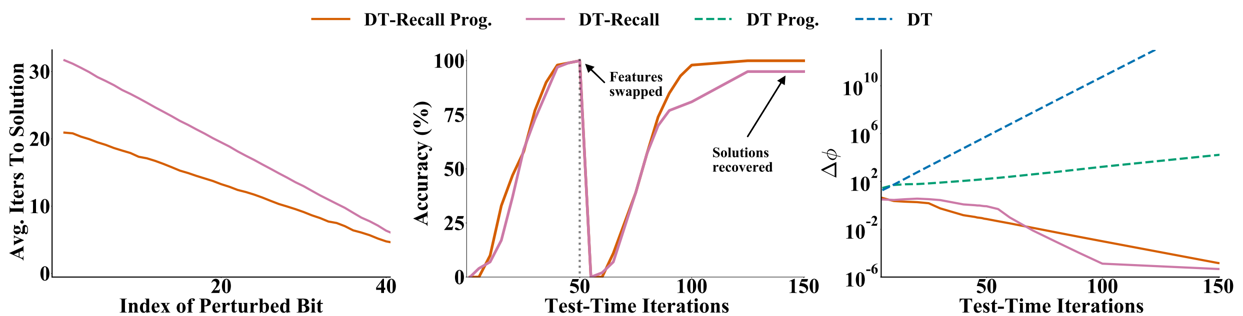

Figure 5.8: **Left:** How long it takes prefix sum models to recover from perturbation – recall keeps this quantity small. **Center:** Test accuracy on 13×13 mazes when features are swapped after 50 iterations. **Right:** The change in the features when solving 13×13 mazes.

5.5.3 Converging to a Fixed Point

We can also study the convergence by measuring the change in the output at each iteration. A decreasing change in the feature maps with each additional iteration suggests that the network manipulates the representation, moving it closer to one that solves the problem, and that it will hold onto this representation (or stay nearby) once it is reached. Moreover, we seek to qualitatively categorize each model type (architecture/loss pair) as convergent or non-convergent. To do we measure the change in the solution at each iteration with ℓ_2-norm of the difference. The right-hand panel of Figure 5.8 shows $\Delta\phi(n) := \|\phi_n - \phi_{n-1}\|_{\ell_2}$. We see that our models appear to converge, while DT networks without recall explode, providing another view into overthinking.

5.5.4 More on the Overthinking Problem

We present additional plots and tables to give a more complete picture of how some models overcome the overthinking problem presented in Section 5.5. Table 5.5 shows the results from several experiments. The first column indicates the average number of iterations to solve a maze and the best accuracy (over all the test iterations considered). Then, we examine the behavior when the features are perturbed after the first iteration by adding mean zero and variance one Gaussian noise ('Noise' column) and we find that this completely destroys the models without recall while only slightly slowing down those with recall. Next, we see the same trend when we replace the features at the first iteration with arrays of zeros ('Zeros' column). Finally, we swap the features after 50 iterations with those obtained from solving an entirely different maze, and again the models without recall cannot recover from this, but those with recall achieve near perfect accuracy. Figures 5.9-5.11 shows accuracy curves for these experiments. Figure 5.12

shows the norm of the difference in features when we input random noise instead of actual mazes. This plot shows that the stable behavior of our models is not limited to the portion of input space occupied by real mazes, rather the process they learn exhibits convergent behavior even on random inputs.

Similar experiments on prefix sum computation reveal that recurrent models with recall seem to converge as well.

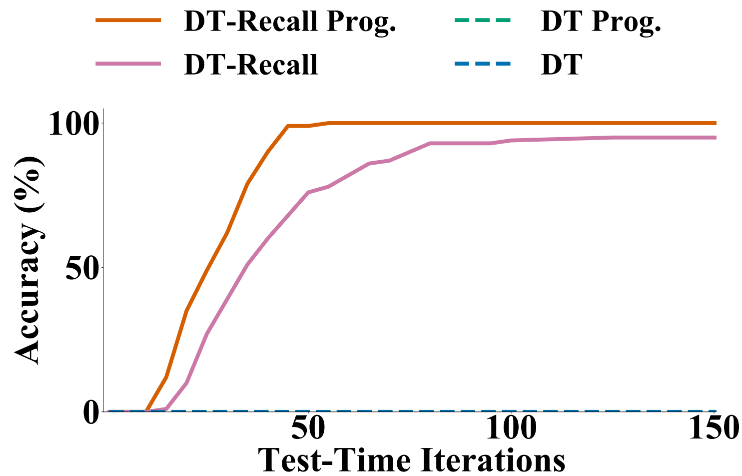

Figure 5.9: Maze solving model performance when Gaussian noise is added to the features after the first iteration. DT models with recall are robust to this perturbation.

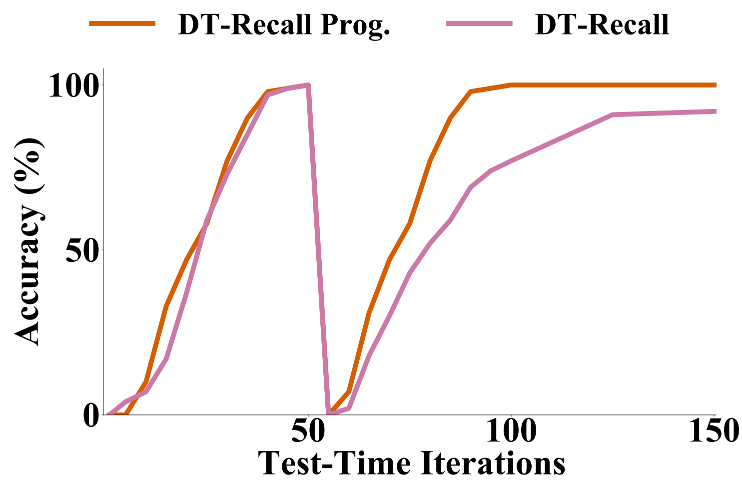

Figure 5.10: Maze solving model performance when the features after 50 iterations are replaced with zeros. DT models with recall are also robust to this perturbation.

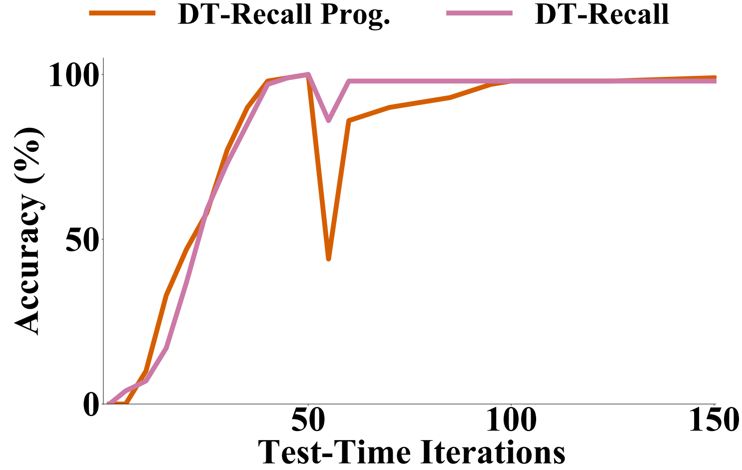

Figure 5.11: Maze solving model performance when the start and end of the maze are changed to be closer together after 50 iterations. DT models with recall can self-correct.

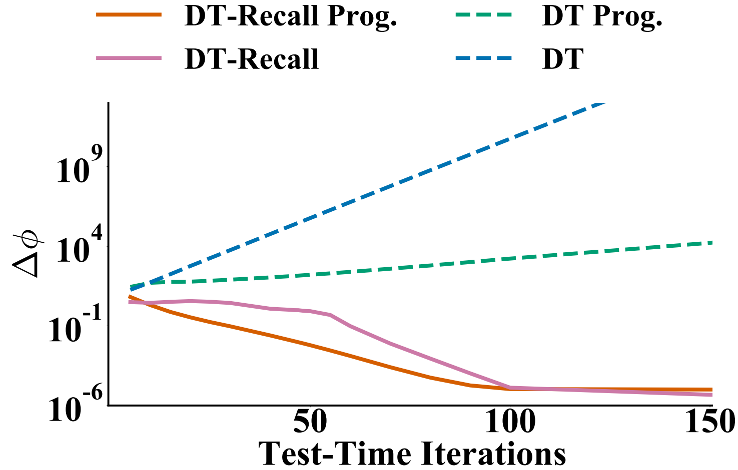

Figure 5.12: The change in the norm of the feature maps for maze solvers as function of iteration shows that models with recall seem to move the feature maps less and less with each iteration.

Table 5.5: Average iterations to solution, with the peak accuracy in parentheses. We evaluate maze solving models on 13×13 mazes, where we intervene in the solving process in different ways.

Model	Clean	Noise	Zeros	Swapped
DT	27.68 (87 %)	- (0%)	- (0%)	- (0%)
DT w/ New Loss	16.87 (95 %)	- (0%)	- (0%)	- (0%)
DT-Recall	23.54 (100 %)	35.70(100 %)	35.04 (96 %)	33.92 (97 %)
DT-Recall w/ New Loss	22.10 (100 %)	25.10(100 %)	22.48 (100 %)	28.79 (100 %)

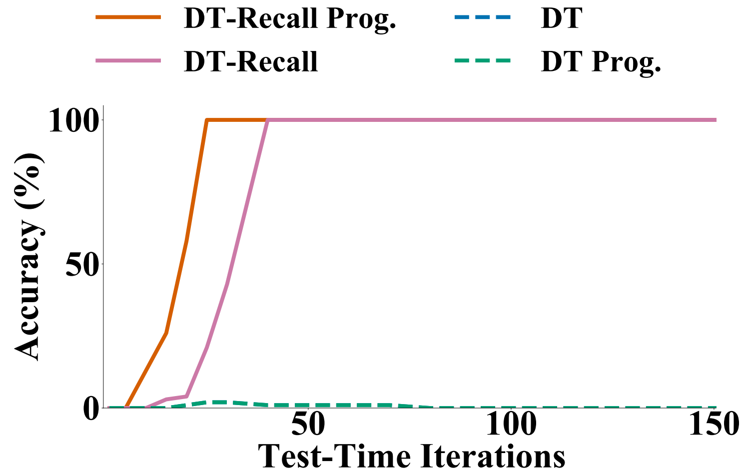

Figure 5.13: Prefix sum model performance when Gaussian noise is added to the features after the first iteration. DT models with recall are robust to this perturbation.

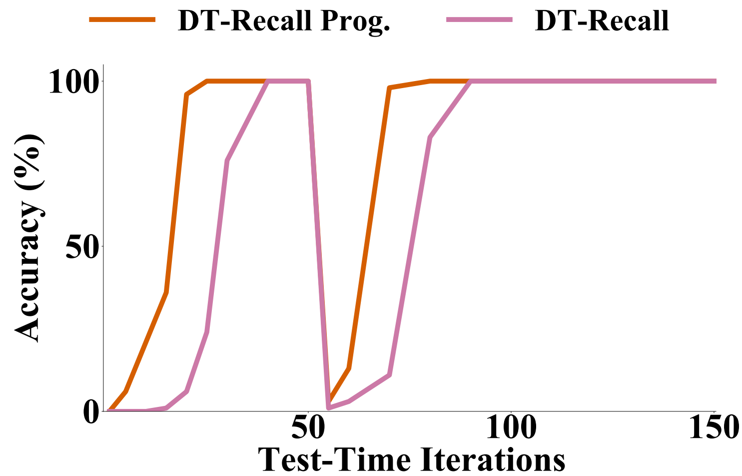

Figure 5.14: Prefix sum model performance when the features after the first iteration are replaced with zeros. DT models with recall are also robust to this perturbation.

5.6 Further Insights

5.6.1 In-distribution Results

Information on how models perform in distribution reveals that each class of model fits the training data and generalizes in the classical sense. Specifically, we report the accuracy on a held-out validation set from the same problem difficulty as the training data. For recurrent mod-

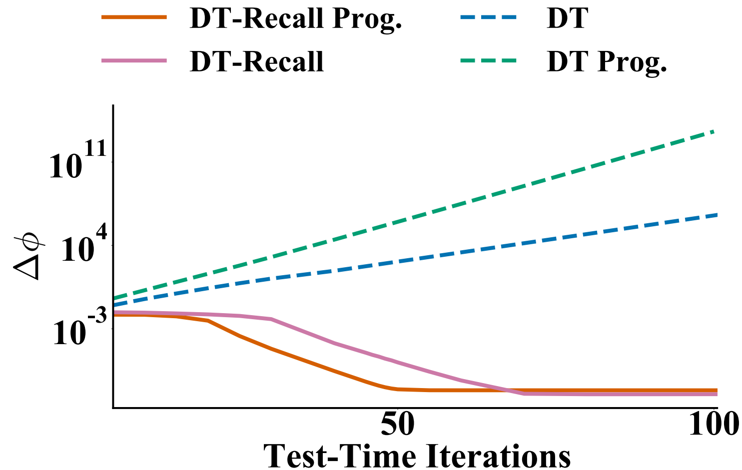

Figure 5.15: The changes in the norm of the feature maps for prefix sums as function of iteration show that models with recall seem to move the feature maps less and less with each iteration.

els, we evaluate the performance using the maximum number of iterations used during training, irrespective of the training objective. See Table 5.6.

5.6.2 Hard to Easy

One natural query about our models is how well they perform on test sets that are easier/smaller than the data used for training. Figure 5.16 shows that recurrent models can generally solve easier/smaller problems in fewer iterations. Even though they solve the easier problems in fewer than the maximum number of iterations used in training, we still see that models without recall suffer from overthinking.

5.6.3 Hyperparameter Selection

All prefix sum models are optimized using Adam [Kingma and Ba, 2014] and gradient clipping. Maze models are also trained with Adam, but no clipping is used. Chess models train stably with SGD without gradient clipping. All training is done with a weight decay coefficient

Table 5.6: Training accuracy and in-distribution validation accuracy for models presented in Figures 5.3 and 5.7.

Trained on 32-bit Prefix Sum Problems			
Model	α	Train Acc. (%)	Valid Acc. (%)
DT	0.00	99.98 ± 0.01	100.00 ± 0.00
DT	1.00	99.57 ± 0.38	97.73 ± 1.80
DT-Recall	0.00	99.95 ± 0.02	100.00 ± 0.00
DT-Recall	1.00	99.98 ± 0.01	100.00 ± 0.00
FF	0.00	99.76 ± 0.14	99.76 ± 0.14
Trained on 9×9 Mazes			
Model	α	Train Acc. (%)	Valid Acc. (%)
DT	0.00	100.00 ± 0.00	100.00 ± 0.00
DT	0.01	99.98 ± 0.01	99.98 ± 0.01
DT-Recall	0.00	99.91 ± 0.07	99.92 ± 0.06
DT-Recall	0.01	99.77 ± 0.15	99.74 ± 0.17
FF	0.00	99.94 ± 0.01	99.93 ± 0.05
Trained on Chess Puzzles 0-600K			
Model	α	Train Acc. (%)	Valid Acc. (%)
DT	0.00	99.98 ± 0.02	92.94 ± 0.34
DT	0.50	99.90 ± 0.01	94.16 ± 0.02
DT-Recall	0.00	99.77 ± 0.02	94.16 ± 0.03
DT-Recall	0.50	99.34 ± 0.02	94.18 ± 0.07
FF	0.00	96.89 ± 0.58	83.24 ± 1.22

of 0.0002 and training with SGD uses a momentum coefficient of 0.9.

In each training run, we hold out 20% of the training data to use as a validation set, and we save for testing the checkpoint with the highest validation accuracy.

Coarse experimentation with learning rates and decay schedules is used to determine optimal choices of these hyperparameters for reproducibility and speed of training. Models were trained with exponential warm-up for a small number of epochs before the main training routine.

In Table 5.7, we present the training hyperparameters shared among models presented in Section 5.4 for each problem instance.

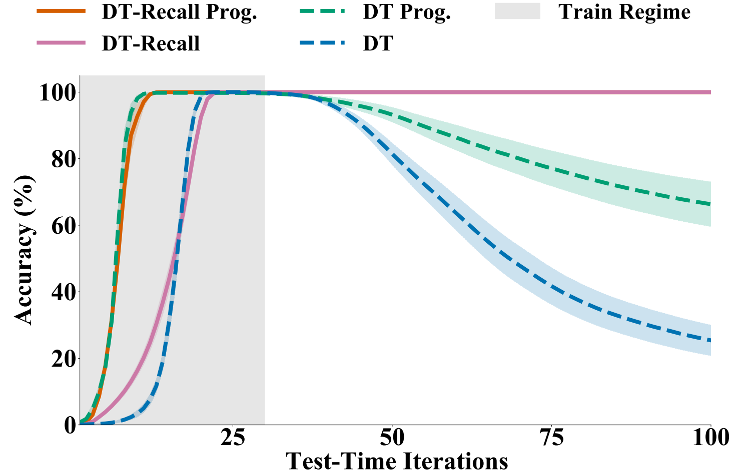

Figure 5.16: Accuracy as a function of iteration for prefix sum models when generalizing from harder 32-bit strings to easier 24-bit strings.

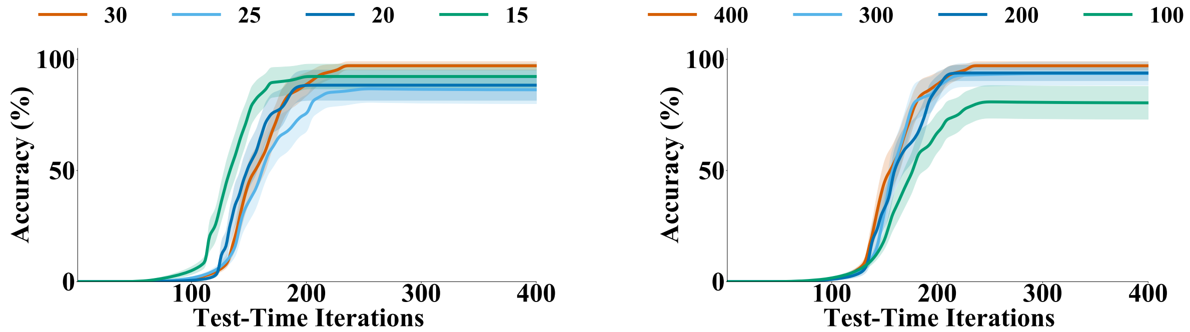

Figure 5.17: **Left:** Accuracy on 512-bit strings for different values of m show that $m = 30$ is sufficient. **Right:** Accuracy on 512-bit strings for different values of w. The gain plateaus on prefix sum models at $w = 400$. Models presented here are trained with $\alpha = 1$ and recall.

Table 5.7: Training hyperparameters. Dashes indicate that we did not utiltize those options. LR denotes learning rate, Decay column lists the epochs after which the learning rate is multiplied by the decay factor.

Task	Optim.	LR	Decay	Decay Factor	Warm-Up	Epochs	Clip Bound
Prefix Sums	Adam	0.0010	$[60, 100]$	0.01	10	150	1.0
Mazes	Adam	0.0010	-	-	10	50	-
Mazes (FF)	Adam	0.0001	$[175]$	0.10	10	200	-
Chess Puzzles	SGD	0.0100	$[100, 110]$	0.01	3	120	-

We perform width and depth experiments to demonstrate that our best prefix sum models are roughly on a plateau in hyperparameter space with respect to performance on 512-bit data.

We see from Figure 5.17 that performance increases steadily with depth, but no additional performance is achieved by increasing m beyond 30. From Figure 5.17, we also see performance steadily increases with network width. We choose to use a width of 400 filters per layer and $m = 30$ because this adequately solves 512-bit data.

These experiments are performed on prefix sums (since they are realtively cheap to run) to exemplify the general correlation between depth, width, and performance observed on all tasks. The choice of depth and width of maze and chess models is also made to be only as large as necessary to avoid excessive memory and computational costs.

We determine the optimal weight to be used it the combination of the loss terms using a coarse grid search. For prefix sums we test models with α values in $[0, 0.25, 0.5, 0.75, 1]$, for mazes we test values $[0, 0.01, 0.1, 0.5, 1]$, and for chess we test values $[0, 0.5, 1]$.

5.7 Conclusion

In this chapter, we improve the algorithmic extrapolation power of neural networks. We propose an architecture and a loss that lead to a gigantic generalization leap from easy training data to much more complex testing examples. Furthermore, we show that our models avoid the overthinking trap. We test algorithmic extrapolation with chess puzzles where the spacial dimension is consistent across difficulties and with maze solving and prefix sum computation where our models can extrapolate to larger problems as well. In fact, our models use more than 2,000 iterations, the equivalent of more than 100,000 convolutional layers, to solve the largest mazes we consider.

Existing methods deteriorate at such large numbers of iterations creating the need for stop-

ping mechanisms. Even for small numbers of iterations, if the halting decision is sub-optimal, then the output may also be sub-optimal. Because our models have the desirable property (for an algorithm) that they converge to fixed points, they run no such risk, require no learned or hand-crafted stopping rules, and can perform well at huge numbers of iterations.

Learning scalable processes and performing logical extrapolation are difficult tasks for most neural models, but with our architecture and loss, we demonstrate that huge leaps in complexity from training to testing data can be handled without overthinking. Importantly, our neural networks learn end-to-end how to perform algorithmic extrapolation. Our findings are limited to the domains we consider and further exploration into how our methods will perform in other settings is needed.

The application of machine learning systems to real world data demands effective extrapolation, even though this is often overlooked. Curated research benchmarks designed for measuring generalization (e.g. ImageNet [Deng et al., 2009]) omit a critical feature of real data in the field: there is often reason to think that data encountered after training could embody more difficult examples than those in the training set. Consider that a practitioner cannot always know *a priori* the range of complexity that exists in their domain. If we are to employ models that cannot extrapolate, we are to miss out on using our machine learning tools to solve problems harder than those for which we already have answers.

Chapter 6: Ongoing and Future Directions

Two ongoing projects are described in this chapter. These ideas are being worked on in collaboration with Arpit Bansal, Alex Stein, Xiyao Wang, Furong Huang, and Tom Goldstein.

6.1 Sudoku Puzzles

In addition to the three logical reasoning problems described at length in Chapter 2 and used in experiments throughout this dissertation, another problem domain we are developing Deep Thinking models for is Sudoku puzzles.

This is well suited for the architectures described above as it is yet another problem that can easily be framed as per-pixel classification. The input is a ten channel 9×9 array where the channels correspond to the ten possible values for each cell – the digits 1 - 9 and blank. The targets are nine channel 9×9 arrays where each of the cells is classified as having one of the nine digits.

Furthermore, Sudoku puzzles have a difficulty gauge, which is the number of non-empty cells in the puzzle. This makes it very easy to split the data into an easy training set and a more difficult test set to study extrapolation. Part of the ongoing work is to determine a significant difficulty leap. So far, we have trained DT-Recall models on 20,000 easy puzzles and they can solve harder puzzles.

Figure 6.1 shows a representative example of the learned Sudoku solving process. Interestingly, our models get some entries correct after only one iterations (marked "Iteration 0" in the top right hand pane of the figure), but some of the given cells are incorrect.

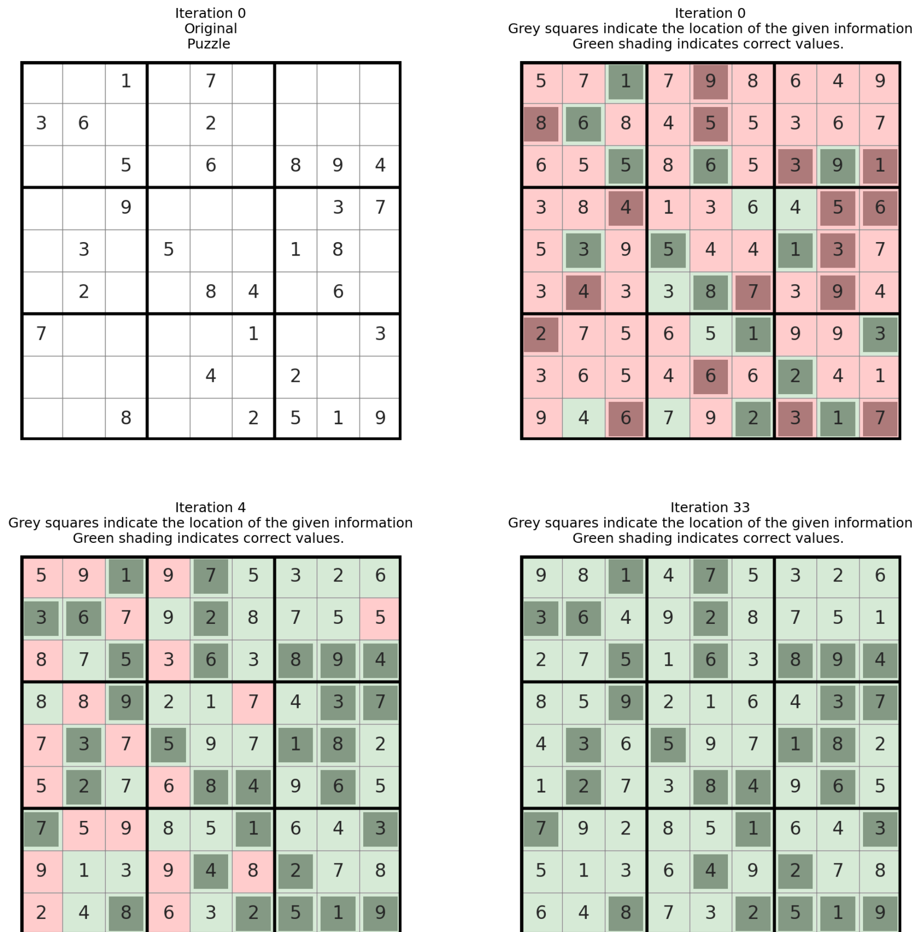

Figure 6.1: An example Sudoku problem from the test set. Top left: the original puzzle showing the given information. Top right: the output of a DT-Recall network after one iteration, colored according to correctness and showing the location of the given information. Bottom: the solution after four iterations and the complete solution after 33 iterations.

One interesting observation about the maze solving models is that they seem to execute an algorithm very similar to dead end filling. We can see this by examining the output at every iteration (a visualization that does not fit in this dissertation). With Sudoku, we are also interested in trying to discern which algorithm our models learn – or more generally whether or not they

learn an already known algorithm.

The larger goal with this direction, which might include even more types of problems, is to harness Deep Thinking networks to learn algorithms in settings where we do not have classical approaches. We might be able not only to learn scalable processes in new domains, but to extract those processes to better understand how to approach these problems.

6.2 Reinforcement Learning

Another domain in which we are exploring the application of Deep Thinking models is reinforcement learning. The chess puzzles we analyze above are solved with supervised learning, a method that does not scale to learning to play entire games of chess. Most state-of-the-art automated chess-playing systems are trained at least in part with reinforcement learning [e.g., Silver et al., 2017]. One common tactic is called *self-play* where a single policy or agent is trained by playing both sides of the board from start to finish. If we can train a Deep Thinking network via self-play, would we see it perform better with more iterations at test time?

This question is particularly interesting in multiplayer games, like chess, where the difficulty of the game is not changed by modifying the rules, dynamics, or environment, but rather by playing a stronger opponent. This is because we could take the very same model – with the same learned parameters – and play it against itself where only one agent is given the latitude to run for more iterations. This experiment is underway and it could have interesting extensions.

One possible direction is to look at speeding up training. For example, if we observe deep thinking behavior, i.e. that extra iterations makes for a stronger agent, could we bootstrap training? To do this, we could run several steps of training with self-play and then increase the

number of iterations in the network before continuing to train. Perhaps this can speed up self-play based training – a notoriously slow process.

Bibliography

Md Zahangir Alom, Mahmudul Hasan, Chris Yakopcic, Tarek M Taha, and Vijayan K Asari. Recurrent residual convolutional neural network based on u-net (r2u-net) for medical image segmentation. *arXiv preprint arXiv:1802.06955*, 2018.

Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant risk minimization. *arXiv preprint arXiv:1907.02893*, 2019.

Alan Baddeley. Working memory: theories, models, and controversies. *Annual review of psychology*, 63:1–29, 2012.

Alan D Baddeley and Graham Hitch. Working memory. In *Psychology of learning and motivation*, volume 8, pages 47–89. Elsevier, 1974.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Trellis networks for sequence modeling. In *International Conference on Learning Representations*, 2018.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. *Advances in Neural Information Processing Systems*, 32:690–701, 2019.

Pierre Baldi and Gianluca Pollastri. The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem. *The Journal of Machine Learning Research*, 4:575–602, 2003.

Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

Tamal Biswas and Kenneth Regan. Measuring level-k reasoning, satisficing, and human error in game-play data. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 941–947. IEEE, 2015.

Svetlana Borovkova and Ioannis Tsiamas. An ensemble of lstm neural networks for high-frequency stock market classification. *Journal of Forecasting*, 38(6):600–619, 2019.

Alexandre Boulch. Sharesnet: reducing residual network parameter number by sharing weights. *arXiv preprint arXiv:1702.08782*, 2017.

Andrew Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. *arXiv preprint arXiv:2102.06171*, 2021.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 6572–6583, 2018.

Marco Ciccone, Marco Gallieri, Jonathan Masci, Christian Osendorfer, and Faustino Gomez. Nais-net: Stable deep networks from non-autonomous differential equations. *Advances in Neural Information Processing Systems*, 31, 2018.

Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. Emnist: an extension of mnist to handwritten letters, 2017.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2018.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.

David Eigen, Jason Rolfe, Rob Fergus, and Yann LeCun. Understanding deep architectures using a recursive convolutional network. *arXiv preprint arXiv:1312.1847*, 2013.

Cristobal Eyzaguirre and Alvaro Soto. Differentiable adaptive computation time for visual reasoning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12817–12825, 2020.

Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-aware minimization for efficiently improving generalization. *arXiv preprint arXiv:2010.01412*, 2020.

Karlis Freivalds and Renars Liepins. Improving the neural gpu architecture for algorithm learning. *arXiv preprint arXiv:1702.08727*, 2017.

Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A Wichmann, and Wieland Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. *arXiv preprint arXiv:1811.12231*, 2018.

Felix A Gers and E Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

Micah Goldblum, Avi Schwarzschild, Ankit B. Patel, and Tom Goldstein. Adversarial attacks on machine learning systems for high-frequency trading, 2020.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Klaus Greff, Rupesh K Srivastava, and Jürgen Schmidhuber. Highway and residual networks learn unrolled iterative estimation. *arXiv preprint arXiv:1612.07771*, 2016.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

Christian Hill. Making a maze, Apr 2017. URL `https://scipython.com/blog/making-a-maze/`.

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

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

W Ronny Huang, Zeyad Emam, Micah Goldblum, Liam Fowl, Justin K Terry, Furong Huang, and Tom Goldstein. Understanding generalization through visualizations. *Proceedings on "I Can't Believe It's Not Better!" at NeurIPS Workshops*, 2020.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

Herbert Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*, volume 5.01. GMD-Forschungszentrum Informationstechnik Bonn, 2002.

Andrew Jaegle, Felix Gimeno, Andrew Brock, Andrew Zisserman, Oriol Vinyals, and Joao Carreira. Perceiver: General perception with iterative attention. *arXiv preprint arXiv:2103.03206*, 2021.

Stanisław Jastrzebski, Devansh Arpit, Nicolas Ballas, Vikas Verma, Tong Che, and Yoshua Bengio. Residual connections encourage iterative inference. *arXiv preprint arXiv:1710.04773*, 2017.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Kohitij Kar, Jonas Kubilius, Kailyn Schmidt, Elias B Issa, and James J DiCarlo. Evidence that recurrent circuits are critical to the ventral stream's execution of core object recognition behavior. *Nature neuroscience*, 22(6):974–983, 2019.

Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. Shallow-deep networks: Understanding and mitigating network overthinking. In *International Conference on Machine Learning*, pages 3301–3310. PMLR, 2019.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited, 2019.

Alex Krizhevsky et al. Learning multiple layers of features from tiny images, 2009.

Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3367–3375, 2015.

Qianli Liao and Tomaso Poggio. Bridging the gaps between residual learning, recurrent neural networks and visual cortex. *arXiv preprint arXiv:1604.03640*, 2016.

Lichess. Lichess open puzzles database. `https://database.lichess.org/#puzzles`, 2021. Accessed: 2021-04-01.

Reid McIlroy-Young, Siddhartha Sen, Jon Kleinberg, and Ashton Anderson. Aligning superhuman ai with human behavior: Chess as a model system. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1677–1687, 2020.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning, 2011.

Kwwabena Nuamah. Deep algorithmic question answering: Towards a compositionally hybrid AI for algorithmic reasoning. *arXiv preprint arXiv:2109.08006*, 2021.

Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2(11): e7, 2017.

Rasmus Berg Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks. In *32nd Conference on Neural Information Processing Systems*, pages 3368–2278. Neural Information Processing Systems Foundation, 2018.

Pedro Pinheiro and Ronan Collobert. Recurrent convolutional neural networks for scene labeling. In *International conference on machine learning*, pages 82–90. PMLR, 2014.

Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2021.

Tord Romstad, Marco Costalba, and et al. Kiiski, Joona. Stockfish: A strong open source chess engine., 2010-2022. URL `https://stockfishchess.org/`.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Jürgen Schmidhuber. Self-delimiting neural networks. *arXiv preprint arXiv:1210.0118*, 2012.

Jürgen Schmidhuber, Daan Wierstra, Matteo Gagliolo, and Faustino Gomez. Training recurrent networks by evolino. *Neural computation*, 19(3):757–779, 2007.

Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Arpit Bansal, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. Datasets for studying generalization from easy to hard examples. *arXiv preprint arXiv:2108.06011*, 2021a.

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. *Advances in Neural Information Processing Systems*, 34:6695–6706, 2021b.

Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

Manli Shu, Zuxuan Wu, Micah Goldblum, and Tom Goldstein. Preparing for the worst: Making networks less brittle with adversarial batch normalization. *arXiv preprint arXiv:2009.08965*, 2020.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. *arXiv preprint arXiv:1602.02867*, 2016.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

Hongxu Yin, Pavlo Molchanov, Jose M Alvarez, Zhizhong Li, Arun Mallya, Derek Hoiem, Niraj K Jha, and Jan Kautz. Dreaming to distill: Data-free knowledge transfer via deepinversion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8715–8724, 2020.

Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.