Copyright
by
Garrett Joseph Bingham
2023

The Dissertation Committee for Garrett Joseph Bingham
certifies that this is the approved version of the following dissertation:

Optimizing Neural Networks through Activation Function Discovery and Automatic Weight Initialization

Committee:

Risto Miikkulainen, Supervisor

Quoc Le

Qiang Liu

Yuke Zhu

Optimizing Neural Networks through Activation Function Discovery and Automatic Weight Initialization

by

Garrett Joseph Bingham

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2023

Dedicated to anyone on a mental health journey;

Acknowledgments

This dissertation wouldn't have been possible without incredible support from my family, friends, mentors, and coworkers.

To Mom, for understanding when no one else does. To Dad, for time spent in the mountains. To Hunter, for listening. To Kate, for being my friend. To Cole, for making me laugh. To Luke, for being yourself. To Blake, for reminding me how fun it is to explore. To Drew, for giving the best hugs.

To all my wonderful friends from Ogden, Cambridge, New Haven, Bilbao, Wilmington, Budapest, New York, Seattle, Austin, and Oakland: thank you for making me smile.

Thank you to all of my professors, internship advisors, and especially the teachers from my childhood. I love learning because of you.

The LEAF team at Cognizant directly supported my research over the last few years. I am especially grateful to Elliot and Xin for our brainstorming sessions, to Mohak, Sid, and Dan for their help implementing my ideas, and to Olivier and Talia for making me feel at home away from home.

Finally, to Risto: thank you for giving me all the space to explore and all the support to succeed.

Optimizing Neural Networks through Activation Function Discovery and Automatic Weight Initialization

Publication No. _____

Garrett Joseph Bingham, Ph.D.
The University of Texas at Austin, 2023

Supervisor: Risto Miikkulainen

Automated machine learning (AutoML) methods improve upon existing models by optimizing various aspects of their design. While present methods focus on hyperparameters and neural network topologies, other aspects of neural network design can be optimized as well. To further the state of the art in AutoML, this dissertation introduces techniques for discovering more powerful activation functions and establishing more robust weight initialization for neural networks. These contributions improve performance, but also provide new perspectives on neural network optimization. First, the dissertation demonstrates that discovering solutions specialized to specific architectures and tasks gives better performance than reusing general approaches. Second, it shows that jointly optimizing different components of neural networks is synergistic, and results in better performance than optimizing individual components alone. Third, it demonstrates that learned representations are easier to optimize than hard-coded ones, creating further opportunities for AutoML. The dissertation thus makes concrete progress towards fully automatic machine learning in the future.

Table of Contents

Acknowledgments **5**

Abstract **6**

List of Tables **12**

List of Figures **16**

Chapter 1. Introduction **22**

1.1 Motivation 22
1.2 Challenges 24
1.3 Approach 26
1.4 Guide to the Reader 29

Chapter 2. Background **31**

2.1 Automated Machine Learning 31
2.1.1 Hyperparameter Optimization 32
2.1.2 Neural Architecture Search 32
2.1.3 Other Aspects of Neural Network Design 33
2.1.4 Zero-Cost Proxies 33
2.1.5 Meta-Learning 34
2.2 Activation Functions 34
2.2.1 Manually Designed Functions 34
2.2.2 Automatically Discovered Functions 35
2.3 Weight Initialization 36
2.3.1 Activation-Function-Dependent Initialization 37
2.3.2 Topology-Dependent Initialization 38
2.3.3 Layer-Dependent Initialization 40
2.3.4 Data-Dependent Initialization 40

2.3.5 Summary 41
2.4 Fisher Information Matrix 41
2.5 Conclusion 42

Chapter 3. CAFE: Evolutionary Optimization of Deep Learning Activation Functions 43
3.1 Evolving Activation Functions 43
3.1.1 Search Space 43
3.1.2 Mutation 44
3.1.3 Crossover 46
3.1.4 Evolution 46
3.2 Experiments 47
3.2.1 Search Strategies 47
3.2.2 Activation Function Specialization 48
3.3 Results 50
3.3.1 Improving Performance 50
3.3.2 Specialized Activation Functions 55
3.4 Discussion 56
3.5 Conclusion 57

Chapter 4. PANGAEA: Discovering Parametric Activation Functions 60
4.1 The PANGAEA Method 61
4.1.1 Representing and Modifying Activation Functions 61
4.1.2 Discovering Activation Functions with Evolution 66
4.2 Datasets and Architectures 67
4.3 Main Results 68
4.3.1 Overview 69
4.3.2 Specialized Activation Functions 69
4.3.3 General Activation Functions 72
4.3.4 Shapes of Discovered Functions 73
4.4 Diving Deeper: Experiments on Ablations, Variations, and Training Dynamics 76
4.4.1 Additional Baseline Search Strategies 76
4.4.2 Effect of Parameterization 78

4.4.3 Adding Parameters to Fixed Baseline Activation Functions 78
4.4.4 Scaling Up to Larger Networks 79
4.4.5 A New Task: All-CNN-C on CIFAR-10 81
4.4.6 Training Dynamics of Evolved Activation Functions 82
4.5 Evaluating Robustness: Experiments on Reliability, Flexibility, and Efficiency 83
4.5.1 Reliability of PANGAEA 85
4.5.2 Parameters: per-layer, per-channel, or per-neuron? 89
4.5.3 Efficiency of PANGAEA 91
4.6 Additional Results with Learnable Activation Functions 93
4.7 Baseline Activation Function Details 96
4.8 Scope of PANGAEA Search Space 97
4.9 Size of the PANGAEA Search Space 100
4.10 Discussion 102
4.11 Conclusion 104

Chapter 5. AutoInit: Analytic Signal-Preserving Weight Initialization for Neural Networks 105

5.1 Motivation 105
5.2 Neural Network Signal Propagation 106
5.3 The AutoInit Framework 107
5.4 Mean and Variance Estimation for Different Layer Types 110
5.5 Hyperparameter Variation in CNNs 116
5.6 Stability in Deep ResNets 118
5.7 High-Resolution Images with Transformers 121
5.8 Scaling up to ImageNet 122
5.9 Contrast with Data-Dependent Initialization 123
5.10 Enabling Neural Architecture Search 124
5.11 Enabling Activation Function Discovery 128
5.12 Statistical Significance of Results in Activation Function Meta-Learning . . 133
5.13 Discussion 134
5.14 Conclusion 138

Chapter 6. AQuaSurF: Efficient Activation Function Optimization through Surrogate Modeling 140
6.1 Activation Function Benchmark Datasets . . . 142
6.2 Fisher Information Matrix Details . . . 144
6.2.1 Preliminaries . . . 144
6.2.2 Approximations . . . 145
6.2.3 Layer-Specific Implementation . . . 146
6.2.4 Eigenvalue Calculation . . . 148
6.3 Features and Distance Metrics . . . 149
6.3.1 FIM Eigenvalues . . . 149
6.3.2 Activation Function Outputs . . . 150
6.4 Using the Features as a Surrogate . . . 151
6.4.1 Visualization with UMAP . . . 151
6.4.2 FIM Eigenvalues . . . 152
6.4.3 Activation Function Outputs . . . 155
6.4.4 Combining Features: Eigenvalues & Outputs . . . 156
6.4.5 Constructing a Surrogate . . . 159
6.5 Features and Surrogate Details . . . 159
6.6 Searching on the Benchmark Tasks . . . 161
6.6.1 Setup . . . 161
6.6.2 Results . . . 162
6.7 Searching on New Tasks . . . 165
6.7.1 Setup . . . 165
6.7.2 Results . . . 166
6.8 Discussion . . . 170
6.9 Conclusion . . . 174

Chapter 7. Discussion and Future Work 175
7.1 Optimizing Other Aspects of Neural Network Design . . . 175
7.2 General and Specialized Solutions . . . 175
7.3 Joint Optimization of Multiple Components . . . 176
7.4 Computational Cost Considerations . . . 177
7.5 Optimizing Alternative Objectives . . . 177

7.6 Designing Better Representations 178
7.7 Towards Artificial General Intelligence 178

Chapter 8. Conclusion **180**
8.1 Contributions 180
8.2 Big Picture 182

Appendices **183**

Appendix A. Training Details **184**
A.1 CAFE 184
A.1.1 Training Setup 184
A.1.2 Search Implementation 185
A.2 PANGAEA 185
A.2.1 Training Details 186
A.2.2 Implementing Custom Activation Functions 188
A.3 AutoInit 190
A.3.1 Convolutional, Residual, and Transformer Network Experiment Details 190
A.3.2 Neural Architecture Search Experiment Details 193
A.4 AQuaSurF 196
A.4.1 Implementation Details 196
A.4.2 Preliminary Experiments 198
A.4.3 Activation Function Search Spaces 201

Appendix B. Compute Infrastructure and Cost **203**
B.1 Compute Infrastructure 203
B.2 Cost 204
B.2.1 CAFE 204
B.2.2 PANGAEA 204
B.2.3 AutoInit 205
B.2.4 AQuaSurF 205

Bibliography **205**

Vita **233**

List of Tables

3.1 The top three activation functions discovered by each search strategy, along with the baseline activation functions ReLU and Swish. The functions achieved the highest validation set accuracy after 50 epochs of training with WRN-28-10 on CIFAR-10. The final test accuracies listed on the right are the median of five runs after training WRN-28-10 from scratch for 200 epochs with each activation function on CIFAR-10 and CIFAR-100. Function plots have domain $x \in (-5, 5)$ but have different ranges. Functions marked with an asterisk (*) occasionally did not train to completion due to asymptotes at $x = -\epsilon$. Exhaustive search finds multiple activation functions in a simple search space (S_1) that consistently outperform ReLU and Swish. Evolution with loss-based fitness was the only technique that was able to discover such activation functions in a more complicated search space (S_2), suggesting that it is the most promising technique for scaling up in the future. 51

3.2 Confidence intervals (95%) and independent t-tests comparing mean accuracies after training WRN-28-10 on CIFAR-100 for 200 epochs with the best function from S_1, $(\arctan(x)) * (\min\{x, 0\})$, the best function from S_2, $(-((\arctan((x)^3)) * (\cos(1)))) * (-((\arctan(\min\{x, 0\})) * (\max\{|x|, 0\})))$, and baseline functions ReLU and Swish. The discovered functions perform significantly better than both baselines. WRN-28-10 was not trained 50 times with Best from S_2 and with Swish due to time constraints. In repeated trials, Swish occasionally caused the network to stall during training, explaining its low mean accuracy. 54

3.3 Test set accuracy of WRN-40-4 with various activation functions after 200 epochs of training on CIFAR-100. Results reported are median of five runs. The top activation function discovered for WRN-28-10 on CIFAR-10, $\tanh(x) * \min\{x, 0\}$, successfully transfers to this new task, outperforming both baselines. However, a search designed specifically for WRN-40-4 on CIFAR-100 discovers a novel activation function, $\sigma(x) * \mathrm{erf}(x)$ that results in even higher performance. This result demonstrates that the main power of activation function metalearning is to be able to specialize the function to the architecture and dataset. 56

4.1 The operator search space consists of basic unary and binary functions as well as existing activation functions (Section 4.7). $\sigma(x) = (1 + e^{-x})^{-1}$. The unary operators bessel_i0e and bessel_i1e are the exponentially scaled modified Bessel functions of order 0 and 1, respectively. 62

4.2 CIFAR-100 test set accuracy aggregated over ten runs, shown as mean ± sample standard deviation. Asterisks indicate a statistically significant improvement in mean accuracy over ReLU, with * if $p \leq 0.05$, ** if $p \leq 0.01$, and *** if $p \leq 0.001$; p-values are from one-tailed Welch's t-tests. The top accuracy for each architecture is in bold. Baseline activation function details and references are given in Section 4.7. 71

4.3 WRN-10-4 accuracy with different activation functions on CIFAR-100, shown as mean ± sample standard deviation across ten runs. PANGAEA discovers better activation functions than random search and nonparametric evolution. 77

4.4 CIFAR-100 test set accuracy aggregated over ten runs, shown as mean ± sample standard deviation. The parametric evolved functions tend to outperform their nonparametric counterparts, demonstrating the value of parameterization. 78

4.5 CIFAR-100 test set accuracy aggregated over ten runs, shown as mean ± sample standard deviation. Trivially parameterizing existing fixed activation functions does not substantially improve performance. PANGAEA, on the other hand, utilizes evolutionary search to discover functional forms that are well suited to taking advantage of the parameters, leading to better performance. 80

4.6 Specialized activation functions discovered for WRN-10-4, ResNet-v1-56, and ResNet-v2-56 are evaluated on larger versions of those architectures: WRN-16-8, ResNet-v1-110, and ResNet-v2-110, respectively. CIFAR-100 test accuracy is reported as mean ± sample standard deviation across three runs. Specialized activation functions successfully transfer to WRN-16-8 and ResNet-v2-110, outperforming ReLU. 81

4.7 All-CNN-C accuracy with different activation functions on CIFAR-10, shown as mean ± sample standard deviation across ten runs. PANGAEA improves performance significantly also with this different architecture and task. 82

4.8 Performance of the best activation functions from multiple PANGAEA runs with ResNet-v1-56. CIFAR-100 test accuracy is shown as the mean ± sample standard deviation across three runs. The three independent per-channel runs produce activation functions of similar performance, demonstrating the reliability of PANGAEA. PANGAEA also discovers good activation functions with per-layer or per-neuron parameters, showing its flexibility. The very best per-layer and per-neuron functions are difficult to find, suggesting that their distribution is long-tailed. 86

4.9 CIFAR-100 test accuracy with different architectures and PReLU variants reported as mean ± sample standard deviation across ten runs. Per-neuron PReLU gets the best performance on WRN-10-4 and ResNet-v2-56, but per-layer PReLU is the best for ResNet-v1-56. 90

4.10 CIFAR-100 test accuracy with PAU using a specialized training setup. Performance is comparable to other baseline activation functions, but some runs fail due to training instability. 94

4.11 The number of activation functions representable by a computation graph with a given number of nodes. The PANGAEA search space contains over ten trillion activation functions, and therefore provides a good foundation for finding powerful activation functions with different properties. 101

5.1 CoAtNet top-1 accuracy on Imagenette, shown as median of three runs. The first four experiments vary the activation function, while the fifth removes all normalization layers from the architecture. A "-" indicates that training diverged. AutoInit produces the best model in three of the five settings, and remains stable even without normalization layers. 122

5.2 ResNet-50 top-1 and top-5 validation accuracy on ImageNet. AutoInit improves performance, even with large and challenging datasets. 122

6.1 Accuracy with different activation functions. The CIFAR-100 results show the median test accuracy from three runs, and the ImageNet results show the validation accuracy from a single run. AQuaSurF discovers novel activation functions that outperform all baselines in every case. This result demonstrates both that good functions matter, and the power of optimizing them to the task. 168

6.2 ResNet-50 top-1 accuracy on ImageNet. Results are the median of three runs. The best activation functions discovered in the searches (Table 6.1) successfully transfer to this new task, with eight of the nine functions outperforming ReLU. 169

A.1 Configuration of neural architecture search experiments in the five tasks. Entries with a (B) or (M) suffix apply to CoDeepNEAT blueprints or modules, respectively. These values were found to work well in preliminary experiments. When AutoInit is applied to an evolved network, it replaces the weight initialization method selected by evolution, but the setup otherwise remains unchanged. The neural architecture search experiments were designed to show that AutoInit improves performance in a wide variety of settings, including those with different data modalities, network topologies, computational complexities, and hyperparameter configurations. 194

A.2 Training details and hyperparameter values used in the experiments in Chapter 6. 197

A.3 Activation function search spaces were defined through computation graphs consisting of basic unary and binary operators as well as existing activation functions [15]. 202

List of Figures

3.1 An example of activation function mutation. The tree represents the activation function $(\min\{1, \cosh(x)\})^3 * \sigma(e^x + \arctan(x))$. One node in the tree is selected uniformly at random and replaced with another operator in the search space, also uniformly at random. The resulting activation function is $(\min\{1, \cosh(x)\})^3 * |e^x + \arctan(x)|$. By introducing variability, mutation ensures evolution explores the search space sufficiently. It prevents high-performing activation functions from overly skewing the early generations of the search process. 45

3.2 In crossover, two parent activation functions exchange randomly selected subtrees of equivalent depth, producing one new child activation function. Crossover enables the best activation functions to pass on their characteristics to the rest of the population. This mechanism is what enables evolution to discover better activation functions more quickly than random search. 45

3.3 Top validation accuracy per generation for three search strategies in the S_2 search space. There are 50 activation functions in each generation of a search. All activation functions are trained with WRN-28-10 on CIFAR-10 for 50 epochs, and the highest validation accuracy obtained among all activation functions in a given generation is reported. Evolution with loss-based fitness finds better activation functions more quickly than evolution with accuracy-based fitness or random search. The first generation of random search is poor due to chance; since each generation is independent, the generations of random search could be arbitrarily reordered. 49

3.4 Best activation function by WRN-28-10 validation accuracy after 50 epochs of training on CIFAR-10 using evolution with loss-based fitness. Top validation accuracy improves from 92.2 in Generation 1 to 93.9 in Generation 10. Evolution is able to discover effective activation functions that are not likely to be discovered by hand. In particular, the top activation function discovered by evolution is smooth everywhere, unlike ReLU, which is not smooth at $x = 0$. This difference is likely the reason for its superior performance. 53

3.5 Activation function accuracy across tasks. Each data point represents validation accuracy when training with a given activation function from S_1 with WRN-40-4 on CIFAR-100 for 50 epochs, and with WRN-28-10 on CIFAR-10 for 50 epochs. Some activation functions perform well when paired with a different architecture and dataset. Other functions are specialized to a given architecture and dataset, and do not transfer. The results suggest that reasonable performance can be expected from general evolved activation functions, but that the best performance comes from evolving activation functions for a specific task. . . 58

3.6 A summary of the best activation functions found in the first set of experiments. (a): Exhaustive search in S_1 discovers $(\tanh(x)) * (\min\{x, 0\})$, and evolution in S_2 discovers $(e^{(\min\{\mathrm{erf}(x),0\})-(\max\{x,0\})}) * (\min\{(\arctan((x)^3)) * (\max\{|x|, 0\}), 0\})$. Both functions achieve a median test accuracy of 94.1 with WRN-28-10 on CIFAR-10, outperforming that of ReLU (94.0) and Swish (93.8). (b): The functions $(\arctan(x)) * (\min\{x, 0\})$ in S_1 and $(-((\arctan((x)^3)) * (\cos(1)))) * (-((\arctan(\min\{x, 0\})) * (\max\{|x|, 0\})))$ in S_2 outperform ReLU and Swish by statistically significant margins with WRN-28-10 on CIFAR-100, demonstrating the power of novel activation functions. (c): Functions evolved for WRN-28-10 on CIFAR-10 perform well with WRN-40-4 on CIFAR-100, but a new function discovered specifically for WRN-40-4 on CIFAR-100, $\sigma(x)*\mathrm{erf}(x)$, achieves even higher performance. This result shows that the biggest advantage of activation function search is the ability to discover functions that are specialized to the architecture and dataset. (d): The baseline activation functions, ReLU and Swish, are included for visual comparison. 59

4.1 Random activation function initialization. The initial population consists of random samples of two kinds of computation graphs, randomly initialized with the operators in Table 4.1. In this manner, the search starts with simple graphs and gradually expands to more complex forms. 62

4.2 Evolutionary operations on activation functions. In an 'Insert' mutation, a new operator is inserted in one of the edges of the computation graph, like the Swish(x) in *(b)*. In a 'Remove' mutation, a node in the computation graph is deleted, like the addition in *(c)*. In a 'Change' mutation, an operator at a node is replaced with another, like addition with multiplication in *(d)*. These first three mutations are useful in refining the function locally. In contrast, in a 'Regenerate' mutation *(e)*, every operator in the graph is replaced by a random operator, thus increasing exploration. 64

4.3 Parameterization of activation functions. In this example, parameters are added to $k = 3$ random edges, yielding the parametric activation function $\alpha\sigma(\beta|x| - \arctan(\gamma x))$. 66

4.4 Progress of PANGAEA with three different neural networks. The plots show the best accuracy achieved among all activation functions evaluated so far. The stars on the plot specify the time when notable activation functions were discovered during evolution; the expression for each such function is written next to the star. Evolution quickly discovered activation functions that outperform ReLU (accuracy with ReLU shown at $x = 0$), and continued to improve throughout the experiment. Note that this figure shows validation accuracy, while Table 4.2 lists test set accuracy. 70

4.5 Adaptation of parametric activation functions over time and space. **Top:** The parameters change during training, resulting in different activation functions in the early and late stages. The plots were created by averaging the values of α, β, and γ across the entire network at different training epochs. **Bottom:** The parameters are updated separately in each channel, inducing different activation functions at different locations of a neural network. The plots were created by averaging α, β, and γ at each layer of the network after the completion of training. 75

4.6 Training curves of All-CNN-C on CIFAR-10 with different activation functions. Accuracy with ReLU is shown in gray, and accuracy with the discovered functions in different colors. The training accuracy is shown as a solid line, and validation accuracy as a dashed line. All of these discovered functions outperformed ReLU, but the training curves show different behavior in each case. In (a) both curves are above those of ReLU the whole time, suggesting ease of learning. In (b), they exceed those of ReLU only in the end, suggesting early regularization. More complex profiles (such as those in (c) and (d)) are also observed, suggesting that their combinations are possible as well. 84

4.7 Average population fitness across time for four independent PANGAEA runs. The plots show the average validation accuracy achieved with the 64 most recently evaluated activation functions at any given time in the search process. All four PANGAEA runs gradually discover better activation functions as they explore the search space, with the per-layer run slightly below the others. Importantly, the two per-channel PANGAEA runs progress at similar rates, demonstrating the reliability of PANGAEA. 87

4.8 Fitness (validation accuracy) and compute cost (runtime in hours) among all activation functions considered in four independent PANGAEA processes on ResNet-v1-56. Each point represents a different activation function. The distribution of fitness and compute cost are shown in histograms on the top and right, respectively. All PANGAEA variants explore activation functions of similar complexity and reliably discover many novel functions that outperform ReLU. 88

4.9 Computational efficiency of PANGAEA. The plot shows the performance of the best activation function discovered so far (y-axis) after a given amount of compute was spent (x-axis). All four PANGAEA runs discover activation functions that outperform ReLU with relatively little compute, demonstrating the efficiency of PANGAEA. If even better performance is needed, additional compute can be spent. 92

4.10 ResNet-v1-20 and ResNet-v1-32 test accuracy on CIFAR-100 with the SPLASH activation function. SPLASH units work well in shallow networks, but become unstable with increased depth. This result explains the success of SPLASH in the original work by Tavakoli et al. [184], and also shows why SPLASH units fail with the architectures considered in this work. 95

5.1 All-CNN-C test accuracy on CIFAR-10. AutoInit results in comparable or better performance with different activation functions, better performance across all dropout rates and weight decay settings, and is less sensitive to the choice of learning rate than the default initialization. 117

5.2 Signal propagation in All-CNN-C networks with different (a) activation functions and (b) dropout rates. With the default initialization, signals often vanish with depth, and their behavior is inconsistent across activation functions and dropout rates. With AutoInit, the variance fluctuates naturally as each layer modifies its input. At layers with weights (marked in red), AutoInit scales the weights appropriately to return the variance to approximately 1.0, stabilizing training in each case. 118

5.3 Signal propagation in residual networks. Gaussian input was fed to the networks and empirical variance computed at each layer. Since `ReLU`, `BatchNormalization`, and `Add` are counted as individual layers in this diagram, the total number of layers is different from that in the architecture name (i.e. ResNet-164 has 164 convolutional layers but over 500 total layers). The default initialization causes exploding signals, while AutoInit ensures signal propagation is stable. 119

5.4 ResNet accuracy on CIFAR-10 with different settings. (a) Accuracy of unnormalized ResNet architectures after five epochs of training with different learning rates and weight initializations. While default initialization makes training difficult in ResNet-56 and impossible at greater depths, AutoInit results in consistent training at all depths. (b) Accuracy of ResNet-164 with a variety of learning rate schedules and initializations. AutoInit is comparable to or outperforms the default initialization in every case. 120

5.5 Mean CIFAR-10 test accuracy for AutoInit vs. LSUV with different numbers of samples S. Each evaluation is repeated 10 times; the shaded area shows the maximum and minimum accuracy among all trials. AutoInit is consistent, but LSUV struggles when S is small or the network is deep. 123

5.6 (a) The CoDeepNEAT method. Modules replace nodes in the blueprint to create a candidate neural network. (b) An example activation function created with the PANGAEA method. The computation graph represents the parametric function $f(x) = \alpha \cdot \sigma(\beta \cdot |x| - \arctan(\gamma \cdot x))$. CoDeepNEAT and PANGAEA generate a variety of architectures and activation functions that can be used to evaluate AutoInit's generality and flexibility. 125

5.7 Evaluation of AutoInit with neural architecture search. (a) Performance improvement over generations in the five tasks. AutoInit outperforms the evolved initialization on four tasks and matches it on one. (b) Representative networks evolved with AutoInit. Although the networks are distinct, AutoInit initializes them properly, leading to good performance in each case. 126

5.8 Progress of neural architecture search in the five tasks. The data is the same as that in Figure 5.7, but this plot also shows how AutoInit can stabilize mean population performance, leading to more reliable discovery of powerful models. 127

5.9 Evaluation of AutoInit with activation function discovery. (a) Distribution of accuracies achieved with 200 activation functions and different weight initialization strategies. AutoInit and AutoInit++ make training more stable and allow more high-performing activation functions to be discovered than the default initialization does. (b) High-performing activation functions. The red line shows the function at initialization, with $\alpha = \beta = \gamma = 1$. The blue lines show the shapes the activation function takes during training, created by sampling α, β, γ from $\mathcal{U}(0.5, 2.0)$. AutoInit's flexibility should turn out useful for developing new activation functions in the future. 131

5.10 Error EDFs for PANGAEA activation functions when using different weight initialization strategies. The Kolmogorov-Smirnov statistic D quantifies the maximum vertical distance between the EDFs, and shows that proper initialization provides a measurable increase in expected performance. Notice that the x-axis shows percent error, and not accuracy as in Figure 5.9. 134

6.1 Distribution of validation accuracies with 2,913 unique activation functions from the three benchmark datasets. Many activation functions result in failed training (indicated by the chance accuracy of 0.1), suggesting that searching for activation functions is a challenging problem. However, most of these functions do not have valid FIM eigenvalues, and can thus be filtered out effectively. . 143

6.2 Distribution of validation accuracies across the benchmark datasets. Each point represents one activation function, and its x and y coordinates represent its performance in two of the three datasets. Some activation functions perform well on all tasks, and their improvements correlate. Others are more specialized and succeed only with one or two tasks. 143

6.3 UMAP embedding of the 2,913 activation functions in the benchmark datasets. Each point stands for a unique activation function, represented by an 80-dimensional output feature vector. The embedding locations of four common activation functions are labeled. The black x's mark coordinates interpolating between these four functions, and the grid of plots on the bottom shows reconstructed activation functions at each of these points. UMAP interpolates smoothly between different kinds of functions, suggesting that it is a good approach for learning low-dimensional representations of activation functions. 153

6.4 UMAP embeddings of activation functions for each dataset (column) and feature type (row). Each point represents a unique activation function, and the points are colored according to their validation accuracy on the given dataset. The colored triangles identify the locations of six well-known activation functions. The areas of similar performance are more continuous in the bottom row; that is, using both FIM eigenvalues and activation function outputs provides a better low-dimensional representation than using either feature alone. 154

6.5 FIM eigenvalue distributions for different architectures and activation functions. The legends show the activation function and the corresponding validation accuracy in different tasks. Although negating an activation function changes its shape, it does not substantially change its behavior nor its performance. FIM eigenvalues capture this relationship between activation functions. The eigenvalues are thus useful for finding activation functions that appear different but in fact behave similarly, and these discoveries in turn improve the efficiency of activation function search. 158

6.6 Search results on the three benchmark datasets. Each curve represents a different search algorithm (KNR, RFR, or SVR) utilizing a different UMAP feature (FIM eigenvalues, function outputs, or both; these features are visualized in Figure 6.4). The curves represent the validation accuracy of the best activation function discovered so far, averaged across 100 independent trials, and the shaded areas show the 95% confidence interval around the mean. In all cases, regression with UMAP features outperforms random search, and searching with both eigenvalues and outputs outperforms searching with either feature alone. Of the three regression algorithms, KNR performs the best, rapidly surpassing ReLU and quickly discovering near-optimal activation functions in all benchmark tasks. Thus, the features make it possible to find good activation functions efficiently and reliably even with off-the-shelf search methods; the benchmark datasets make it possible to demonstrate these conclusions with statistical reliability. 163

6.7 Progress of activation function searches. Each point represents the validation accuracy with a unique activation function, and the solid line indicates the performance of the best activation function found so far. AQuaSurF discovers new activation functions that outperform all baseline functions in every case. 166

6.8 Low-dimensional UMAP representation of the 425,896 function search space. The activation functions are embedded according to their outputs; each point represents a unique function. The larger points represent activation functions that were evaluated during the searches; they are colored according to their validation accuracy. Although the space is vast, the searches require only tens of evaluations to discover good activation functions. 167

6.9 Sample activation functions discovered with AQuaSurF. "HS" stands for HardSigmoid. (a) The top three functions discovered in each search. Many of these functions are fine-tuned versions of existing activation functions like ELU and Swish. (b) Selected novel activation functions. All of these functions outperformed ReLU and are distinct from existing activation functions. Such designs may serve as a foundation for further improvement and specialization in new tasks. 170

A.1 UMAP projections of FIM eigenvalues using the default hyperparameter of `n_neighbors=15`. The embedding is informative but also noisy. Using `n_neighbors=3`, as shown in the main text, improved performance. 199

Chapter 1

Introduction

> We want AI agents that can discover like we can, not which contain what we have discovered.
>
> Rich Sutton, *The Bitter Lesson*

Recursive self-improvement is hypothesized to be one means to artificial general intelligence (AGI) [18, 206]. The idea is straightforward: if an AI is sophisticated enough, it could design an improved version of itself. The improved AI, which is smarter than the original, could then design an even more capable version of itself. This process could continue, potentially indefinitely, resulting in an arbitrarily capable agent.

Currently, AI systems do not exhibit this kind of recursive self-improvement. In the field of automated machine learning (AutoML), however, a single step of self-improvement is possible. Research in AutoML could therefore conceivably lead to recursive self-improvement. This dissertation improves upon the state of the art in AutoML, and thus provides a stepping stone towards eventual AGI.

1.1 Motivation

Early machine learning approaches relied on human-engineered features in order to learn representations of data. As computing power increased, these approaches gave way to

more general methods that automatically extract relevant features from the data [180]. This trend has persisted across subfields of machine learning.

For example, computer vision algorithms initially relied on detecting edges, corners, and other human-inspired image features [125]. By leveraging additional compute and large amounts of data, convolutional neural networks learn such features automatically, and result in networks that surpass humans in image classification [41, 72, 105].

Similarly, in the field of natural language processing, manually crafted features like bag-of-words or TF-IDF (term frequency-inverse document frequency) were initially useful for rudimentary text understanding [166]. Later, techniques like word2vec captured more nuanced word semantics by learning representations automatically from a billion word corpus [139]. Today, large language models have pushed this trend even further, using enormous amounts of compute and data in order to model long-range dependencies in text and handle complex tasks such as question answering and machine translation [20, 96, 188].

To date, methods leveraging large amounts of compute for general feature learning have been more successful than specialized approaches relying on human knowledge across a broad range of tasks. Interestingly, this pattern extends beyond feature learning to algorithm design itself: AI can design better AI than humans can [162, 176, 214].

However, even state-of-the-art AutoML algorithms still make use of human-inspired designs in some areas [43, 197]. For example, many of these approaches will automate the design of a neural network topology, but will reuse human designed activation functions or weight initialization strategies. This situation is suboptimal, but also points to an opportunity for improvement. Indeed, suboptimal human designs could function as a bottleneck that

prevents single step self-improvement from progressing to recursive self-improvement. This bottleneck can be avoided by automating the entire machine learning pipeline.

With this motivation, this dissertation introduces automated approaches to activation function discovery and weight initialization, two areas where suboptimal human designs are frequently used. Experiments show the contributions already improve upon the state of the art. As more compute becomes available, the approaches can be scaled up, providing even better solutions for machine learning problems in the future. This dissertation thus provides concrete steps towards fully automatic machine learning.

1.2 Challenges

Automating activation function discovery and weight initialization introduces a number of challenges, discussed below.

Evaluating new activation functions is computationally expensive. In order to do it, a neural network must be trained from scratch. The training process is already costly, and repeating this process for many different activation functions can become prohibitively so. In order to make this research possible, Chapters 3 and 4 utilize distributed high performance computing. The evaluation of different activation functions is parallelized across multiple machines, and the results are later aggregated. However, Chapter 6 makes significant contribution to this area: It introduces a surrogate that makes the search for new activation functions orders of magnitude more efficient. The surrogate makes it possible to search for new activation functions with more standard hardware, or alternatively search in larger spaces with parallel hardware.

In order for a weight initialization system to be general, it must be able to stabilize the signal propagation for an arbitrary neural network. This is difficult, due to the wide variety of neural network designs. In Chapter 5, a method is developed to address this obstacle by analyzing neural network signal propagation at the level of individual layers. Although many types of neural networks exist, modern architectures typically use similar kinds of layers. The proposed method models signal propagation based on these layer types, and initializes the weights accordingly. The method also provides fallback mechanisms in case unknown layer types are encountered: The signal propagation can be derived manually, or using Monte Carlo sampling.

Perhaps the biggest challenge of AutoML is overcoming previous human design biases, especially when it comes to interactions between different neural network components. For example, ReLU is an extremely popular activation function, and dropout was designed with ReLU in mind [146, 175]. However, when using the SELU activation function, dropout has to be modified because this activation function does not saturate to zero [99]. Indeed, neural networks are famously brittle systems, and changing one component often ruins performance if the changes are not made carefully. This dissertation addresses this challenge in two ways. First, for activation functions, the dissertation discovers functions that are specialized to individual tasks. Thus, even if a model has a set of hyperparameters that were tuned with a different activation function in mind, a better function can still be discovered by treating those hyperparameters as fixed and optimizing the activation function against them. Second, the automated weight initialization system is designed to be as general as possible. Thus, even if a neural network has suboptimal manual designs, the initialization algorithm still takes them into account and improves performance. Indeed, experiments in Chapter 5 show

that the initialization approach is more robust to suboptimal hyperparameters than previous methods are. Thus, although the eventual goal of AutoML is to fully automate the entire machine learning pipeline, the contributions in this dissertation are able to progress towards this goal while still being compatible with suboptimal human designs that are often used in practice.

1.3 Approach

This dissertation introduces automated methods for activation function discovery and weight initialization. Four separate systems were created in order to thoroughly understand these areas. Each system provides unique insights; the systems and discoveries are briefly summarized next.

In the first system, called CAFE, novel activation functions are discovered with a variety of approaches, namely exhaustive search in a smaller search space and random search and evolution in a larger space. Evolution discovers activation functions that achieve high accuracy and outperform baseline functions like ReLU and Swish, demonstrating its creativity and ability to efficiently explore large spaces. CAFE reveals two key findings. The first is that optimizing the design of the activation function is by itself a means to improving the performance of neural networks. This discovery therefore elevates the importance of activation functions in neural network design by showing that their design must be considered in order to maximize performance. The second conclusion is that evolutionary search is creative, discovering designs unlikely to be created by humans. It therefore provides a promising search mechanism for activation functions and other aspects of neural network design in the future.

The second system, PANGAEA, combines evolution and gradient descent into one

optimization process. PANGAEA extends CAFE by making it more flexible in several ways. First, instead of using fixed tree structures, activation functions are represented as arbitrary computation graphs, and more powerful mutation operators are introduced to explore the larger search space efficiently. Second, while the first system used fixed activation functions, PANGAEA utilizes parametric functions. This construction allows the activation functions to change shape during the different stages of training and at different locations within a network. Third, PANGAEA discovers specialized activation functions that are customized to specific tasks, leading to even better performance. In conclusion, PANGAEA shows that better performance can be found by removing human design biases and by giving the optimization process more flexibility and freedom to be creative.

In order to automate weight initialization for different neural networks, a third system, AutoInit, was developed. While many initialization strategies have been proposed in the past, they usually apply only to neural networks with specific activation functions, topologies, or layer types. This situation makes it difficult to evaluate new architectures or activation functions, because it is hard to initialize them properly with existing techniques. AutoInit addresses this issue by calculating analytic mean- and variance-preserving weight initialization for neural networks automatically. It provides an appropriate default initialization automatically, resulting in better and more reliable performance. Thus, in addition to better performance from optimizing hyperparameters, architectures, and activation functions, AutoInit shows that the weight initialization can also be optimized. It therefore provides further evidence of the power of fully automatic machine learning. In addition to improving performance on individual tasks, AutoInit accelerates neural architecture search and activation function discovery. This result is particularly illuminating: It shows that there is a

synergistic effect when multiple aspects of neural network design are optimized in tandem. This result should further motivate progress in AutoML in the future.

The fourth system, AQuaSurF, makes activation function AutoML practical and scalable by introducing a surrogate approach. It also demonstrates that a surrogate can be learned from the data, leading to fundamental insights into what activation functions are made of. Note that early computer vision systems looked for human-designed features like edges and corners, but eventually the features were automatically learned with convolutional neural networks. Similarly, researchers often design new activation functions based on intuitive characteristics like smoothness, groundedness, monotonicity, and limit behavior [6, 149], but it should also be possible to learn automatically what features an activation function must possess in order to be successful. To this end, AQuaSurF learns better representations of activation functions in a data-driven way. Convolutional, residual, and vision transformer based architectures are trained from scratch with 2,913 different activation functions, resulting in three activation function benchmark datasets: `Act-Bench-CNN`, `Act-Bench-ResNet`, and `Act-Bench-ViT`. Exploratory data analysis with these benchmark datasets reveals two activation function properties that are highly indicative of performance: (1) the spectrum of the Fisher information matrix associated with the model's predictive distribution at initialization, and (2) the activation function's output distribution. Based on these features, a metric space is created where a low-dimensional representation of the activation functions can be learned. This space is then used as a surrogate in the search for good activation functions. It turns out that the space is so powerful that out-of-the-box regression algorithms discover good activation functions in only tens of evaluations, improving performance on datasets as large as ImageNet. Thus, AQuaSurF provides a unique perspective by showing that the underlying

representation of the activation function is important. By moving away from human-inspired encodings to automatically learned ones, AI is better able to improve itself. Learning new representations for other aspects of neural network design may similarly be an important step towards fully automated machine learning in the future.

Thus, the four systems automate the design of activation functions and weight initialization. Each system provides unique insights that will be useful in extending this work towards fully automated machine learning. The first system, CAFE, provides a new perspective on neural network optimization, showing that performance can be improved by optimizing the design of the activation function. The second system, PANGAEA, shows that introducing flexibility across multiple dimensions in the search process is both possible and beneficial, resulting in more creative and powerful solutions. The third system, AutoInit, shows that proper weight initialization helps to fairly evaluate novel architectures and activation functions, thus accelerating research in these areas. AutoInit also shows that optimizing multiple aspects of neural network design in tandem produces better results than just focusing on one aspect alone. The fourth system, AQuaSurF, learns new representations for activation functions in a data-driven way; the resulting surrogate approach is orders of magnitude more efficient than previous work and may serve as a foundation for practical optimization of other aspects of neural network design in the future.

1.4 Guide to the Reader

The remainder of this dissertation is organized as follows:

Chapter 2 details previous research that inspired the contributions in this dissertation. Chapter 3 introduces CAFE, demonstrating that designing better activation functions is a

new way to optimize neural networks. Chapter 4 presents PANGAEA, which combines evolutionary search and gradient descent into one optimization process for optimizing parametric activation functions. Chapter 5 automates weight initialization with AutoInit, and shows that automating the design of both activation functions and weight initialization simultaneously leads to better results. Chapter 6 introduces the activation function benchmark datasets, and learns a new representation for activation functions. The new representation leads to the AQuaSurF surrogate-based method for activation function optimization that is orders of magnitude more efficient than existing work. Chapter 7 discusses the contributions of this dissertation, and includes ideas for possible future research. The main conclusions are reviewed in Chapter 8.

Chapter 2

Background

This chapter reviews related research that motivates and provides a foundation for the work in this dissertation. First, automated machine learning is discussed, since it provides the main inspiration for this work. Next, research on activation functions is discussed, with connections to Chapters 3 and 4. Then, weight initialization research is reviewed as inspiration to the contributions in Chapter 5. Finally, the Fisher information matrix is introduced, which informs the surrogate approach in Chapter 6.

2.1 Automated Machine Learning

Building a machine learning system requires making many design decisions: hyperparameters, neural architectures, data augmentation, and other components need to be configured [87]. Instead of ad hoc decisions by human researchers, automated machine learning (AutoML) serves to make one or more of these decisions in an automated, principled manner. AutoML simultaneously makes machine learning accessible, since human expertise is not required in every scenario.

AutoML is a broad research area. This section reviews the components of AutoML that are most relevant to this dissertation.

2.1.1 Hyperparameter Optimization

Neural networks have many hyperparameters that need to be chosen before learning can proceed, and choosing suitable values can play a large role in how successful the learning process is. There exist many approaches to hyperparameter optimization, including Bayesian optimization, bilevel programming, evolution strategies, random search, and others [13, 14, 46, 47, 50, 89, 100, 122, 128, 203].

Hyperparameters can often be represented as a low-dimensional, real-valued vector, making it straightforward to experiment with existing optimization methods. In contrast, other aspects of neural network design, such as their topology, activation function, or weight initialization strategy, are more complicated objects that require a more creative optimization approach.

2.1.2 Neural Architecture Search

In neural architecture search [NAS; 21, 22, 25, 27, 43, 60, 119, 126, 191, 197, 212, 214, 215], the goal is to design a neural network architecture automatically. NAS approaches typically focus on optimizing the type and location of the layers and the connections between them. A popular NAS approach is neuroevolution, where neural networks are optimized with evolutionary algorithms [4, 59, 133, 161, 162, 164, 173, 176–179, 196, 202, 205]. Reinforcement learning [10, 53, 214], Monte Carlo tree search [147], gradient descent [120], and random search [111] are also used. NAS approaches often use standard choices for other components like the activation function, weight initialization, loss function, and so on. These components have received less attention in AutoML, but can similarly be optimized, as demonstrated in this dissertation.

2.1.3 Other Aspects of Neural Network Design

Just like the topology of a neural network can be optimized, so too can other aspects of neural network design. For instance, Gonzalez and Miikkulainen [61, 62] used a genetic algorithm [102, 193] to construct novel loss functions, and then optimized the coefficients of the loss functions with a covariance-matrix adaptation evolutionary strategy. They discovered a loss function that results in faster training and higher accuracy compared to the standard cross-entropy loss. Liu et al. [121] evolved normalization-activation layers. They searched for a computation graph that replaced both batch normalization and the activation function in neural networks. The design of components like the learning rate schedule [23, 39], data augmentation strategy [32, 33, 118], optimization algorithm [3, 12, 28, 34], and other objects can similarly be automated [81, 163].

2.1.4 Zero-Cost Proxies

A common drawback to AutoML is its computational cost. Candidate designs must be evaluated in order to understand their performance, and this evaluation is often expensive. In order to partially alleviate this issue, recent work has developed proxy measures that approximate the final performance of neural networks [137, 168, 192]. Instead of training the network, these proxies use only cheap surrogate calculations. These proxies inspired the surrogate approach in Chapter 6, where the efficiency of activation function search was dramatically improved.

2.1.5 Meta-Learning

The field of meta-learning, also called learning to learn, is another related subfield of AutoML [5, 40, 48, 49, 79, 80, 113, 141]. Rather than optimize the design of the model for a single task, meta-learning approaches typically incorporate knowledge and experience from multiple tasks in order to guide the learning algorithm on new, unseen tasks.

2.2 Activation Functions

Activation functions are a crucial component of neural networks. They are what allow deep networks to learn complex, nonlinear relationships from the training data. As such, various types of activation functions have been used in neural networks over the years, including manually designed functions and automatically discovered ones [6, 56, 57, 90, 95, 149, 169, 186, 189].

2.2.1 Manually Designed Functions

Early on, sigmoid and tanh were often used as activation functions [24, 35]. These functions have limited range, and thus were helpful in restricting the magnitude of signals propagating through neural network layers. However, their asymptotic behavior often caused optimization difficulty due to vanishing gradients. ReLU, being unbounded as $x \to \infty$, addressed this limitation, and to this day is arguably the most widely used activation function [146]. Later, Leaky ReLU was introduced to address the dying neuron problem: with ReLU, neurons often become stuck and always output zero [127]. The ELU activation function contains a negative saturation regime which helps to control the forward propagated variance, and the SELU activation function contributed subtle refinements to ELU for even better

stable signal propagation [30, 99].

2.2.2 Automatically Discovered Functions

The activation functions above are some of the most widely known and illustrative examples, but many more exist [6, 149]. In general, researchers often design activation functions to have specific properties in order to increase performance. This practice works in certain cases, but eventually new and better activation functions are needed as more difficult tasks arise. This situation points to two opportunities. First, automating the design of activation functions is a promising means to discovering better functions and avoiding human design biases. Second, using one activation function for all tasks is likely suboptimal, while leveraging different activation functions specialized to specific tasks can yield better results. Fortunately, the two opportunities are complimentary: while it may be time-consuming for researchers to manually design specialized activation functions for new tasks, the specialization can be done easily through automated search.

Indeed, prior work on automatic activation function discovery has shown that it is a promising research area. The approaches have been based on reinforcement learning (RL), evolutionary computation, or gradient descent, and are summarized next.

Reinforcement Learning Ramachandran et al. [160] used RL to design novel activation functions. They discovered multiple functions, but analyzed just one in depth: $\text{Swish}(x) = x \cdot \sigma(x)$. Of the top eight functions discovered, only Swish and $\max\{x, \sigma(x)\}$ consistently outperformed ReLU across multiple tasks, suggesting that improvements are possible but often task specific. This hypothesis is confirmed in Chapter 4, where specialized activation

functions are discovered for different tasks, yielding higher performance.

Evolutionary Computation Marchisio et al. [130] and Hagg et al. [66] used evolutionary computation to select activation functions from predefined lists, but did not discover novel functions. Basirat and Roth [11] used a genetic algorithm to discover novel task-specific piecewise activation functions, but the functions did not outperform ELiSH and HardELiSH, two hand-designed activation functions [11]. This dissertation scales up these approaches in a number of meaningful ways in order to advance the state of the art, including using larger search spaces and more powerful exploration methods.

Gradient Descent Learnable activation functions (LAFs) encode functions with general forms such as polynomial, rational, or piecewise linear, and utilize gradient descent to discover optimal parameterizations during training [2, 64, 143, 184]. The general forms allow most LAFs to approximate arbitrary continuous functions. However, just because a LAF can represent an activation function does not guarantee that the optimal function will be discovered by gradient descent. In Chapter 4, evolutionary computation and gradient descent are combined into a single optimization process that discovers activation functions that outperform LAFs.

2.3 Weight Initialization

In order to ensure that a new activation function does not cause vanishing or exploding signals, it is important to initialize the weights of the network appropriately. This section reviews previous research in neural network weight initialization, which has focused on

stabilizing signals by accounting for specific components of neural networks such as the activation function, topology, layer types, and training data distribution. However, these approaches fail to generalize to networks that do not meet certain design restrictions, limiting their effectiveness. Chapter 5 introduces AutoInit, an adaptive weight initialization algorithm designed to address these limitations.

2.3.1 Activation-Function-Dependent Initialization

As is common in the literature, `fan_in` and `fan_out` refer to the number of connections feeding into and out of a node, respectively. LeCun et al. [108] recommend sampling weights from a distribution with mean zero and standard deviation $\sqrt{\texttt{fan_in}}$. This initialization encourages propagated signals to have variance approximately one if used with an activation function symmetric about the origin, like $1.7159 \tanh\left(\frac{2}{3}x\right)$ or $\tanh(x) + \alpha x$ for some small choice of α. The standard sigmoid $f(x) = 1/(1 + e^{-x})$ induces a mean shift and should not be used in this setting.

Glorot and Bengio [55] proposed one initialization strategy to ensure unit variance in the forward-propagated signals and another to ensure unit variance for the backward-propagated gradients. As a compromise between the two strategies, they initialized weights by sampling from $\mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}, \frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}\right)$. They also avoided sigmoid, and instead chose symmetric functions with unit derivatives at 0, such as tanh or $\text{Softsign}(x) = x/(1+|x|)$.

He et al. [72] introduced the PReLU activation function and a variance-preserving weight initialization to be used with it that samples weights from $\mathcal{N}(0, \sqrt{2/\texttt{fan_in}})$. Similarly, Klambauer et al. [99] introduced SELU, an activation function with self-normalizing properties. These properties are only realized when SELU is used with the initialization scheme by LeCun

et al. [108].

The above weight initialization strategies attempt to solve the same fundamental problem: How can weights be scaled so that repeated applications of the activation function do not result in vanishing or exploding signals? While these approaches solve this problem in a few special cases, the issue is more general. Manually deriving the correct scaling is intractable for complicated activation functions. One approach for an arbitrary function f is to sample Gaussian inputs x and adjust the weights according to the empirical variance $\mathrm{Var}(f(x))$ [19]. Chapter 5 proposes an alternative and potentially more accurate approach: integration by adaptive quadrature [70, 156, 190]. The result is a weight initialization strategy that is compatible with any integrable activation function. Indeed, previous activation-function-dependent initializations are special cases of the AutoInit algorithm.

2.3.2 Topology-Dependent Initialization

The activation-function-dependent initializations discussed above were designed for neural networks composed of convolutional or dense layers. After the introduction of residual networks [ResNets; 73, 74], new weight initialization schemes had to be developed to account for the effect of shortcut connections and various types of residual branches.

Taki [182] analyzed signal propagation in plain and batch-normalized ResNets. They developed a new weight initialization to stabilize training, but did not consider modifications like using deeper residual blocks or reordering components like the activation function or batch normalization layers. In contrast, AutoInit is topology-agnostic: It adapts to any of these changes automatically.

Zhang et al. [210] introduced Fixup, an initialization method that rescales residual branches to stabilize training. Fixup replaces batch normalization in standard and wide residual networks [73, 74, 88, 208] and replaces layer normalization [8] in transformer models [188]. The disadvantages of this scheme are that it only applies to residual architectures, needs proper regularization to get optimal performance, and requires additional learnable scalars that slightly increase model size.

Arpit et al. [7] proposed a new initialization scheme for weight-normalized networks [165] that relies on carefully scaling weights, residual blocks, and stages in the network. Like related approaches, this technique improves performance in specific cases, but imposes design constraints, like requiring ReLU activation functions and a specific Conv $\rightarrow$ ReLU $\rightarrow$ Conv block structure.

Just as tanh-inspired weight initialization does not stabilize training of ReLU networks, initialization schemes designed for non-residual networks fail with ResNets [9, 19, 68]. This observation suggests that future classes of neural networks will again require developing new weight initializations. Additionally, practitioners with models that do not fit neatly within the restricted settings of existing weight initialization research are left to derive their own initialization or use a suboptimal one. For example, many initialization schemes assume that the activation function is ReLU [7, 38, 72, 182, 210]. Indeed, ReLU is currently the most popular activation function [6, 149], but it is not the best choice in every case [69]. ReLU prevents dynamical isometry [155, 167], weakens adversarial training [201], and results in poorer accuracy compared to other activation functions in certain tasks [15]. A general weight initialization strategy that does not impose architectural constraints and achieves good performance in diverse settings is needed. AutoInit is designed to meet this challenge.

2.3.3 Layer-Dependent Initialization

Hendrycks and Gimpel [75] noted that dropout layers [175] also affect the variance of forward-propagated signals in a network. To stabilize training properly, it is necessary to take dropout layers and the specific dropout rate into account in weight initialization. In fact, pooling, normalization, recurrent, padding, concatenation, and other layer types affect the signal variance in a similar way, but current initialization schemes do not take this effect into account. AutoInit is designed to adapt to each of these layer types dynamically, and can be extended to include new layer types as they are introduced in the future.

2.3.4 Data-Dependent Initialization

Mishkin and Matas [140] fed data samples through a network and normalized the output of each layer to have unit variance. Krähenbühl et al. [103] adopted a similar approach, but opted to normalize along the channel dimension instead of across an entire layer. Data-dependent weight initializations are most similar in spirit to AutoInit; they rely on empirical variance estimates derived from the data in order to be model-agnostic. However, data-dependent weight initializations introduce a computational overhead [140], and are not applicable in settings where data is not available or its distribution may shift over time, such as online learning or reinforcement learning. The quality of the initialization is also dependent on the number of the data samples chosen, and suffers when the network is very deep [210]. AutoInit instead uses an analytic approach for greater efficiency and higher accuracy.

2.3.5 Summary

Previous techniques solved the initialization problem for networks with specific activation functions, topologies, and layer types. In contrast, AutoInit does not impose design constraints, depend on data samples, or incur a parameter overhead [37, 213] and is therefore a good starting point especially in new settings.

2.4 Fisher Information Matrix

Proper initialization makes evaluating new activation functions more reliable; however, this evaluation step is still computationally expensive. By providing an inexpensive prediction of final performance, a surrogate model could accelerate the search for new functions. The Fisher information matrix (FIM) makes this surrogate approach possible.

Consider a neural network f parameterized by weights $\boldsymbol{\theta}$ and given inputs $\mathbf{x}$ drawn from a training distribution $Q_{\mathbf{x}}$. This neural network defines the conditional distribution $R_{\mathbf{y}|f(\mathbf{x};\boldsymbol{\theta})}$. The FIM associated with this model is

$$\mathbf{F} = \mathop{\mathrm{E}}_{\substack{\mathbf{x}\sim Q_{\mathbf{x}} \\ \mathbf{y}\sim R_{\mathbf{y}|f(\mathbf{x};\boldsymbol{\theta})}}} \left[\nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{y}, f(\mathbf{x};\boldsymbol{\theta})) \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{y}, f(\mathbf{x};\boldsymbol{\theta}))^{\top} \right], \tag{2.1}$$

where the loss function $\mathcal{L}(\mathbf{y}, \mathbf{z})$ represents the negative log-likelihood associated with $R_{\mathbf{y}|f(\mathbf{x};\boldsymbol{\theta})}$.

The FIM and its eigenvalues $\lambda(\mathbf{F})$ are important quantities in machine learning with many uses. For example, in optimal experiment design [44], the cost of experimentation is minimized by optimizing a chosen criterion. Different criteria with different statistical guarantees are used, but they are typically functions of the eigenvalues of the FIM, such as the maximum or minimum eigenvalue, or the trace of the FIM (sum of the eigenvalues) or determinant of the FIM (product of the eigenvalues).

Past work has also used the eigenvalues of the FIM to analyze the learning dynamics of neural networks in order to infer optimal values of the batch size or learning rate [51, 52, 71, 94, 117]. Because the FIM is related to the optimization landscape at initialization, it provides insights on the learning dynamics of SGD [92] and the dynamics of signal propagation at different layers of neural networks [86]. This information can be used to inform network design by identifying specific layers that are poorly conditioned and therefore difficult to optimize. The FIM is also used in second-order optimization algorithms for neural networks [65, 131, 132]. These approaches are more expensive but often require fewer iterations than first-order methods.

Chapter 6 builds on this work by using the FIM in a surrogate approach to accelerate activation function search. However, instead of choosing one optimality criterion or only considering one summary statistic, all of the eigenvalues of the FIM are kept and an optimal distribution is learned experimentally. These eigenvalues are then used as a feature vector to predict the performance of different activation functions efficiently.

2.5 Conclusion

Work in AutoML has shown that it is possible to optimize the design of complex structures. Previous work on activation function design and weight initialization strategies shows that they are critical components for neural network performance, and that improving upon them can lead to better results. These results suggest that automating activation function design and weight initialization is a promising avenue towards improving the current state of the art as well as an important step progressing towards fully automatic machine learning.

Chapter 3

CAFE: Evolutionary Optimization of Deep Learning Activation Functions

This chapter demonstrates that novel activation functions can outperform baseline functions by statistically significant margins. Because this chapter serves as a foundation for other work in this dissertation, part of the motivation for this work is exploratory. To this end, activation functions are discovered in both small and large search spaces using a variety of search algorithms, including exhaustive search, random search, and evolution with two kinds of fitness functions. The experiments show that evolutionary search is particularly effective and creative [110]. This work, called CAFE (Creative Activation Function Evolution), was done in collaboration with graduate student William Macke [17].

3.1 Evolving Activation Functions

This section presents the approach to evolving activation functions, introducing the search space, mutation and crossover implementations, and the overall evolutionary algorithm.

3.1.1 Search Space

Each activation function is represented as a tree consisting of unary and binary operators. Functions are grouped in layers such that two unary operators always feed into

one binary operator. The following operators, modified slightly from the search space of Ramachandran et al. [160], are used:

- **Unary:** 0, 1, x, $-x$, $|x|$, x^2, x^3, $\sqrt{x}$, e^x, e^{-x^2}, $\log(1+e^x)$, $\log(|x+\epsilon|)$, $\sin(x)$, $\sinh(x)$, $\text{arcsinh}(x)$, $\cos(x)$, $\cosh(x)$, $\tanh(x)$, $\text{arctanh}(x)$, $\max\{x, 0\}$, $\min\{x, 0\}$, $\sigma(x)$, $\text{erf}(x)$, $\text{sinc}(x)$;
- **Binary:** $x_1 + x_2$, $x_1 - x_2$, $x_1 \cdot x_2$, $x_1/(x_2+\epsilon)$, $\max\{x_1, x_2\}$, $\min\{x_1, x_2\}$.

Following Ramachandran et al., a "core unit" is an activation function that can be represented as `core_unit = binary(unary1(x), unary2(x))`. Let F be the set of balanced core unit trees. S is then defined as a family of search spaces

$$S_{d\in\mathbb{N}} = \{f \in F \mid \text{depth}(f) = d\}. \tag{3.1}$$

For example, S_1 corresponds to the set of functions that can be represented by one core unit, S_2 represents functions of the form: `core_unit1(core_unit2(x), core_unit3(x))`, and so on. Examples of functions in S_2 are illustrated in Figures 3.1 and 3.2.

3.1.2 Mutation

In mutation, one node in an activation function tree is selected uniformly at random. The operator at that node is replaced with another random operator in the search space. Unary operators are always replaced with unary operators, and binary operators with binary operators. An example of mutation is shown in Figure 3.1. Theoretically, mutation alone is sufficient for constructing any activation function. However, preliminary experiments showed that crossover can increase the rate at which good activation functions are found.

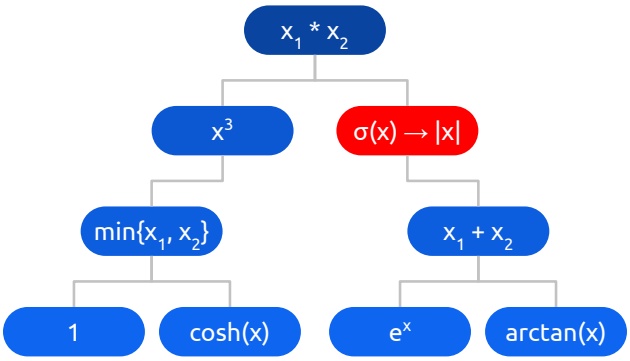


Figure 3.1: An example of activation function mutation. The tree represents the activation function $(\min\{1, \cosh(x)\})^3 * \sigma(e^x + \arctan(x))$. One node in the tree is selected uniformly at random and replaced with another operator in the search space, also uniformly at random. The resulting activation function is $(\min\{1, \cosh(x)\})^3 * |e^x + \arctan(x)|$. By introducing variability, mutation ensures evolution explores the search space sufficiently. It prevents high-performing activation functions from overly skewing the early generations of the search process.

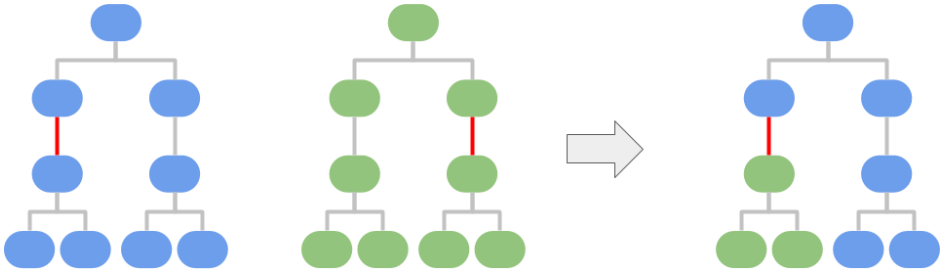


Figure 3.2: In crossover, two parent activation functions exchange randomly selected subtrees of equivalent depth, producing one new child activation function. Crossover enables the best activation functions to pass on their characteristics to the rest of the population. This mechanism is what enables evolution to discover better activation functions more quickly than random search.

3.1.3 Crossover

In crossover, two parent activation functions exchange randomly selected subtrees, producing one new child activation function. The subtrees are constrained to be of the same depth, ensuring the child activation function is a member of the same search space as its parents. Crossover is depicted in Figure 3.2.

3.1.4 Evolution

Starting with a population of N activation functions, a neural network is trained with each function on a given training dataset. Each function is assigned a fitness p_i equal to the softmax of an evaluation metric L_i. This metric could be either accuracy or negative loss obtained on the validation dataset. More specifically,

$$p_i = \frac{e^{L_i}}{\sum\limits_{j=1..N} e^{L_j}}. \tag{3.2}$$

The softmax operation converts the fitness values to a probability distribution, allowing functions to be randomly sampled. From the N activation functions, $2(N-m)$ are selected with replacement for reproduction with probability proportional to their fitness. Crossover followed by mutation is applied to the selected activation functions to obtain a new population of size $N-m$. In order to increase exploration, m randomly generated functions are added to a population that will again be of size N. This process is repeated for several generations, and the activation functions with the best performance over the history of the search are returned as a result.

3.2 Experiments

This section presents experiments with the WRN-28-10 and WRN-40-4 architectures on the CIFAR-10 and CIFAR-100 datasets [104, 208]. In the experiments, the default ReLU activation functions are replaced with candidate functions, and all other settings are kept to default values. Implementation details are in Appendix A.1.

3.2.1 Search Strategies

Three different techniques are used to explore the space of activation functions: exhaustive search, random search, and evolution. Exhaustive search evaluates every function in S_1, while random search and evolution explore S_2. It is noteworthy that evolution is able to discover high-performing activation functions in S_2, where the search space contains over 41 billion possible function strings. In each experiment, the top three activation functions by validation accuracy from the entire search are kept. These functions are then reevaluated on the test set, and the median accuracy of five independent runs is reported.

Exhaustive Search Ramachandran et al. search for activation functions using reinforcement learning and argue that simple activation functions consistently outperform more complicated ones [160]. Although evolution is capable of discovering high-performing, complex activation functions in an enormous search space, exhaustive search can be effective in smaller search spaces. S_1, for example, contains 3,456 possible function strings.

Random Search An illustrative baseline comparison with evolution is random search. Instead of evolving a population of 50 activation functions for 10 generations, 500 random

activation functions from S_2 are grouped into 10 "generations" of 50 functions each.

Evolution As shown in Figure 3.3, evolution discovers better activation functions more quickly than random search in S_2, a search space where exhaustive search is infeasible. During evolution, candidate activation functions are assigned a fitness value based on either accuracy or loss on the validation set. Accuracy-based fitness favors exploration over exploitation: activation functions with poor validation accuracy still have a reasonable probability of surviving to the next generation. A hypothetical activation function that achieves 90% validation accuracy is only 2.2 times more likely to be chosen for the next generation than a function with only 10% validation accuracy since $e^{0.9}/e^{0.1} \approx 2.2$.

Loss-based fitness sharply penalizes poor activation functions. It finds high-performing activation functions more quickly, and gives them greater influence over future generations. An activation function with 0.01 validation loss is 21,807 times more likely to be selected for the following generation than a function with a validation loss of 10. $e^{-0.01}/e^{-10} \approx 21{,}807$.

Both experiments begin with an initial population of $N = 50$ random activation functions, and run through 10 generations of evolution. Each new generation of 50 activation functions is comprised of the top five functions from the previous generation, $m = 10$ random functions, and 35 functions created by applying crossover and mutation to existing functions in the population.

3.2.2 Activation Function Specialization

An important question is the extent to which activation functions are specialized for the architecture and dataset for which they were evolved, or perform well across different

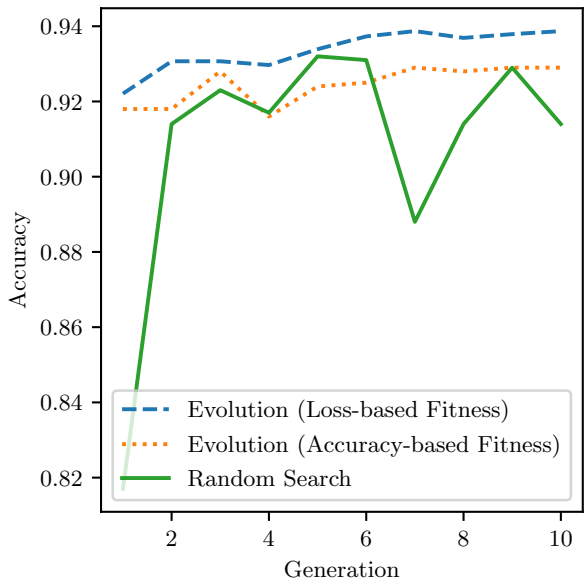


Figure 3.3: Top validation accuracy per generation for three search strategies in the S_2 search space. There are 50 activation functions in each generation of a search. All activation functions are trained with WRN-28-10 on CIFAR-10 for 50 epochs, and the highest validation accuracy obtained among all activation functions in a given generation is reported. Evolution with loss-based fitness finds better activation functions more quickly than evolution with accuracy-based fitness or random search. The first generation of random search is poor due to chance; since each generation is independent, the generations of random search could be arbitrarily reordered.

architectures and datasets. To address this question, activation functions discovered for WRN-28-10 on CIFAR-10 are transferred to WRN-40-4 on CIFAR-100. These activation functions are compared with the best from a small search (1.9K activation functions from S_1) with WRN-40-4 on CIFAR-100.

3.3 Results

This section presents the experimental results, which demonstrate that evolved activation functions can outperform baseline functions like ReLU and Swish.

3.3.1 Improving Performance

Table 3.1 lists the activation functions that achieved the highest validation set accuracies after 50 epochs of training with WRN-28-10 on CIFAR-10. The top three activation functions for each search strategy are included. To emulate their true performance, a WRN-28-10 with each activation function was trained for 200 epochs five times on both CIFAR-10 and CIFAR-100 and evaluated on the test set. The median accuracy of these tests is reported in Table 3.1. Although no search was performed on CIFAR-100 with WRN-28-10, the functions that perform well on CIFAR-10 successfully generalize to CIFAR-100.

The best three activation functions discovered through exhaustive search in S_1 outperform ReLU and Swish. This finding shows how important it is to have an effective search method. There are good functions even in S_1. It is likely that there are even better functions in S_2, but with billions of possible functions, a more sophisticated search method is necessary.

The activation functions discovered by random search have unintuitive shapes (Table 3.1). Although they fail to outperform the baseline activation functions, it is impressive that they still consistently reach a reasonable accuracy. One of the functions (marked with (*) in Table 3.1) discovered by random search occasionally failed to train to completion due to an asymptote at $x = -\epsilon$.

Evolution with accuracy-based fitness is less effective because it does not penalize poor

Table 3.1: The top three activation functions discovered by each search strategy, along with the baseline activation functions ReLU and Swish. The functions achieved the highest validation set accuracy after 50 epochs of training with WRN-28-10 on CIFAR-10. The final test accuracies listed on the right are the median of five runs after training WRN-28-10 from scratch for 200 epochs with each activation function on CIFAR-10 and CIFAR-100. Function plots have domain $x \in (-5, 5)$ but have different ranges. Functions marked with an asterisk (*) occasionally did not train to completion due to asymptotes at $x = -\epsilon$. Exhaustive search finds multiple activation functions in a simple search space (S_1) that consistently outperform ReLU and Swish. Evolution with loss-based fitness was the only technique that was able to discover such activation functions in a more complicated search space (S_2), suggesting that it is the most promising technique for scaling up in the future.

Function Plots		**Accuracy** CIFAR-10	CIFAR-100
	Evolution with Loss-based Fitness (S_2)		
	$(e^{(\min\{\mathrm{erf}(x),0\})-(\max\{x,0\})}) * (\min\{(\arctan((x)^3)) * (\max\{\lvert x\rvert,0\}),0\})$	**94.1**	73.9
	$(e^{\max\{\min\{\mathrm{erf}(x),0\},\max\{x,0\}\}}) * (\min\{(\arctan((x)^3)) * (\max\{\lvert x\rvert,0\}),0\})$	10.0	01.0
	$(-((\arctan((x)^3)) * (\cos(1)))) * (-((\arctan(\min\{x,0\})) * (\max\{\lvert x\rvert,0\})))$	93.9	74.1
	Evolution with Accuracy-based Fitness (S_2)		
	$\min\{e^{-(\min\{(\sinh(x))^2,(0)^2\})^2}, \min\{\min\{\mathrm{erf}(\log(1+e^x)), \mathrm{arcsinh}(x)\},0\}\}$	10.0	01.0
	$\min\{\cos(\max\{(\min\{x,0\})^3, \log(1+e^1)\}), e^{-((\lvert\max\{x,0\}\rvert)+(e^{\sigma(x)}))^2}\}$	93.5	72.5
	$\max\{\max\{(\log(\lvert(\min\{x,0\})+\epsilon\rvert)) * (\sigma(\mathrm{erf}(x))),0\},0\}$ (*)	92.5	72.3
	Random Search (S_2)		
	$(\min\{\max\{(\log(\lvert(x)+\epsilon\rvert))^3, -(\log(\lvert(x)+\epsilon\rvert))\},0\}) + (e^{(\min\{\tanh(x),0\})+(\log(\lvert(\max\{x,0\})+\epsilon\rvert))})$ (*)	93.8	73.9
	$(\arctan(\min\{\sinh(\sin(x)), \arctan(\max\{x,0\})\})) * (\tanh((-(\sin(x))) * (\mathrm{arcsinh}(x))))$	93.3	72.1
	$(\max\{\frac{\min\{(x)^3,0\}}{\max\{\sin(x),0\}+\epsilon},0\}) - (\min\{(x)^2, \max\{x,0\}\})$	93.9	73.2
	Exhaustive Search (S_1)		
	$(\arctan(x)) * (\min\{x,0\})$	94.0	**74.5**
	$(\tanh(x)) * (\min\{x,0\})$	**94.1**	74.3
	$(\min\{x,0\}) * (\mathrm{erf}(x))$	94.0	74.2
	Baseline Activation Functions		
	$\mathrm{ReLU}(x)$	94.0	73.3
	$\mathrm{Swish}(x)$	93.8	71.1

activation functions severely enough. One of the functions failed to learn anything better than random guessing. It was likely too sensitive to random initialization or was unable to learn with the slightly different learning rate schedule of a full 200-epoch training. Another function (marked with (*) in Table 3.1) often did not train to completion due to an asymptote at $x = -\epsilon$. The one function that consistently trained well still failed to outperform ReLU.

Evolution with loss-based fitness is able to find good functions in S_2. One of the three activation functions discovered by evolution outperformed both ReLU and Swish on CIFAR-10, and two of the three discovered outperformed ReLU and Swish on CIFAR-100. Figure 3.4 shows the top activation function after each generation of loss-based evolution. This approach discovered both novel and unintuitive functions that perform reasonably well, as well as simple, smooth, and monotonic functions that outperform ReLU and Swish. It is therefore the most promising search method in large spaces of activation functions.

The performance gains on CIFAR-10 are consistent but small, and the improvement on CIFAR-100 is larger. It is possible that more difficult datasets provide more room for improvement that a novel activation function can exploit.

To evaluate the significance of these results, WRN-28-10 was trained on CIFAR-100 for 200 epochs 50 times with ReLU, 50 times with the best function found by exhaustive search in S_1, $(\arctan(x)) * (\min\{x, 0\})$, 25 times with Swish, and 15 times with the best function found by evolution in S_2, $(-((\arctan((x)^3)) * (\cos(1)))) * (-((\arctan(\min\{x, 0\})) * (\max\{|x|, 0\})))$. Table 3.2 shows 95% confidence intervals and the results of independent t-tests comparing the mean accuracies achieved with each activation function. The results show that replacing a baseline activation function with an evolved one results in a statistically significant increase in accuracy.

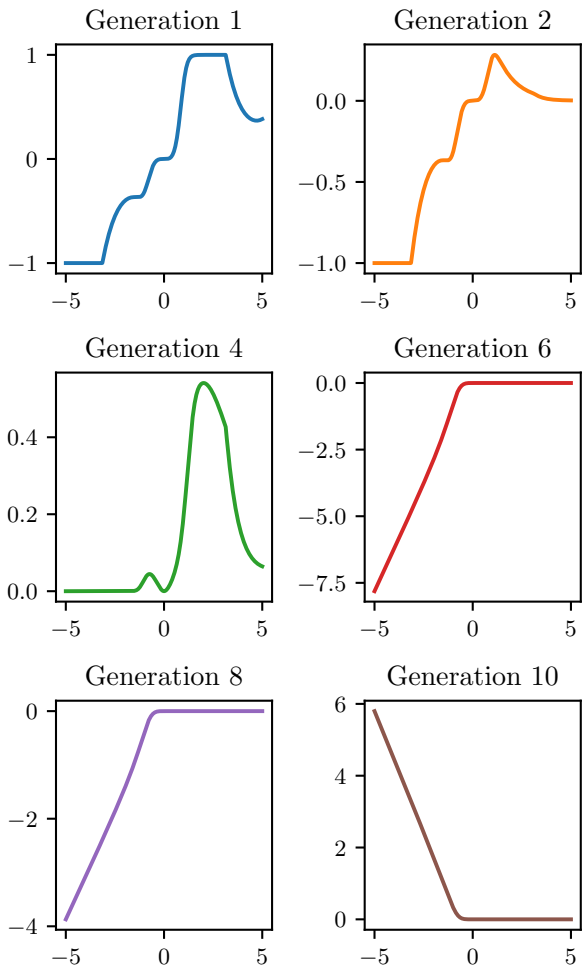


Figure 3.4: Best activation function by WRN-28-10 validation accuracy after 50 epochs of training on CIFAR-10 using evolution with loss-based fitness. Top validation accuracy improves from 92.2 in Generation 1 to 93.9 in Generation 10. Evolution is able to discover effective activation functions that are not likely to be discovered by hand. In particular, the top activation function discovered by evolution is smooth everywhere, unlike ReLU, which is not smooth at $x = 0$. This difference is likely the reason for its superior performance.

Table 3.2: Confidence intervals (95%) and independent t-tests comparing mean accuracies after training WRN-28-10 on CIFAR-100 for 200 epochs with the best function from S_1, $(\arctan(x)) * (\min\{x, 0\})$, the best function from S_2, $(-((\arctan((x)^3)) * (\cos(1)))) * (-((\arctan(\min\{x, 0\})) * (\max\{|x|, 0\})))$, and baseline functions ReLU and Swish. The discovered functions perform significantly better than both baselines. WRN-28-10 was not trained 50 times with Best from S_2 and with Swish due to time constraints. In repeated trials, Swish occasionally caused the network to stall during training, explaining its low mean accuracy.

Activation Function	Mean Accuracy (95% C.I.)	Repeats
Best from S_1	74.2 ($\pm$0.1)	50
Best from S_2	74.0 ($\pm$0.2)	15
ReLU	73.2 ($\pm$0.2)	50
Swish	49.6 ($\pm$11.6)	25

Activation Functions	t-statistic; p-value
Best from S_1 vs. ReLU	9.73; 4.64×10^{-16}
Best from S_1 vs. Swish	5.91; 9.91×10^{-8}
Best from S_2 vs. ReLU	4.51; 2.91×10^{-5}
Best from S_2 vs. Swish	3.17; 2.98×10^{-3}

Among the top activation functions discovered, many are smooth and monotonic. Hand-engineered activation functions frequently share these properties [149]. Two notable exceptions were found by random search and evolution with accuracy-based fitness. Although these functions do not outperform ReLU, the fact that WRN-28-10 was able to achieve such high accuracy with these arbitrary functions raises questions as to what makes an activation function effective. Ramachandran et al. [160] asserted that simpler activation functions consistently outperformed more complicated ones. However, the high accuracy achieved with activation functions discovered by evolution in S_2 demonstrates that complicated activation functions can compete with simpler ones. Such flexibility may be particularly useful in

specialization to different architectures and datasets. It is plausible that there exist many unintuitive activation functions which can outperform the more general ones in specialized settings. Evolution is well-positioned to discover them.

3.3.2 Specialized Activation Functions

Since different functions are seen to emerge in different experiments, an important question is: How general or specialized are they to a particular architecture and dataset? To answer this question, the top activation function discovered for WRN-28-10 on CIFAR-10, $\tanh(x) * \min\{x, 0\}$, was trained with WRN-40-4 on CIFAR-100 for 200 epochs. This result was then compared with performance achieved by $\sigma(x) * \mathrm{erf}(x)$, an activation function discovered specifically for WRN-40-4 on CIFAR-100. Table 3.3 summarizes the result: the activation function discovered for the first task does transfer to the second task, but even higher performance is achieved when a specialized function is discovered specifically for the second task.

The specialized activation function, $\sigma(x) * \mathrm{erf}(x)$, is shown in Figure 3.6c. It is similar to $\sigma(x)$ in that it tends towards 0 as $x \to -\infty$, and approaches 1 as $x \to \infty$. It differs from $\sigma(x)$ in that it has a non-monotonic bump for small, negative values of x. A 50-epoch training of WRN-40-4 on CIFAR-100 with activation $\sigma(x)$ achieved validation accuracy of just 63.2. The superior performance of $\sigma(x) * \mathrm{erf}(x)$ suggests that the negative bump was important, as the shapes of the two activation functions are otherwise similar. This result demonstrates how evolution can discover specializations that make a significant difference.

Table 3.3: Test set accuracy of WRN-40-4 with various activation functions after 200 epochs of training on CIFAR-100. Results reported are median of five runs. The top activation function discovered for WRN-28-10 on CIFAR-10, $\tanh(x) * \min\{x, 0\}$, successfully transfers to this new task, outperforming both baselines. However, a search designed specifically for WRN-40-4 on CIFAR-100 discovers a novel activation function, $\sigma(x) * \mathrm{erf}(x)$ that results in even higher performance. This result demonstrates that the main power of activation function metalearning is to be able to specialize the function to the architecture and dataset.

Activation Function	Accuracy
$\sigma(x) * \mathrm{erf}(x)$	**72.6**
$\tanh(x) * \min\{x, 0\}$	72.1
$\mathrm{Swish}(x)$	71.1
$\mathrm{ReLU}(x)$	71.0

3.4 Discussion

Among the top activation functions discovered, many are smooth and monotonic. Hand-engineered activation functions frequently share these properties [149]. Two notable exceptions were found by random search and evolution with accuracy-based fitness. Although these functions do not outperform ReLU, the fact that WRN-28-10 was able to achieve such high accuracy with these arbitrary functions raises questions as to what makes an activation function effective. Ramachandran et al. [160] asserted that simpler activation functions consistently outperformed more complicated ones. However, the high accuracy achieved with activation functions discovered by evolution in S_2 demonstrates that complicated activation functions can compete with simpler ones. Such flexibility may be particularly useful in specialization to different architectures and datasets. It is plausible that there exist many unintuitive activation functions which can outperform the more general ones in specialized settings. Evolution is well-positioned to discover them.

Activation functions discovered by evolution perform best on the architectures and datasets for which they were evolved. Figure 3.5 demonstrates this principle. More generally, it shows the performance of several activation functions when trained with WRN-28-10 for 50 epochs on CIFAR-10 and when trained with WRN-40-4 for 50 epochs on CIFAR-100. Activation functions that perform well for one task often perform well on another task, but not always. Therefore, if possible, one should evolve them specifically for each architecture and dataset. However, as the results in Section 3.3 show, it is feasible to evolve using smaller architectures and datasets and then transfer to scaled up architectures and more difficult datasets within the same domain.

In the future, it may be possible to push such generalization further, by evaluating functions across multiple architectures and datasets. In this manner, evolution may be able to combine the requirements of multiple tasks, and discover functions that perform well in general. However, the main power in activation function search is to discover functions that are specialized to each architecture and dataset. In that setting most significant improvements are possible.

3.5 Conclusion

Multiple strategies for discovering novel, high-performing activation functions were presented and evaluated: namely exhaustive search in a small search space (S_1) and random search and evolution in a larger search space (S_2). Evolution with loss-based fitness finds activation functions that achieve high accuracy and outperform standard functions such as ReLU and novel functions such as Swish, demonstrating the power of search in large spaces. The best activation functions successfully transfer from CIFAR-10 to CIFAR-100

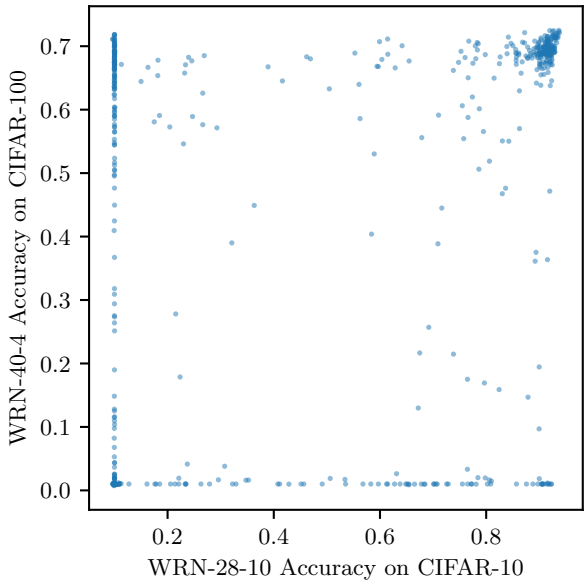


Figure 3.5: Activation function accuracy across tasks. Each data point represents validation accuracy when training with a given activation function from S_1 with WRN-40-4 on CIFAR-100 for 50 epochs, and with WRN-28-10 on CIFAR-10 for 50 epochs. Some activation functions perform well when paired with a different architecture and dataset. Other functions are specialized to a given architecture and dataset, and do not transfer. The results suggest that reasonable performance can be expected from general evolved activation functions, but that the best performance comes from evolving activation functions for a specific task.

and from WRN-28-10 to WRN-40-4. However, the main power of activation function search is in finding specialized functions for each architecture and dataset, leading to significant improvement. These results (summarized in Figure 3.6) provide a foundation for the next chapter, where evolution and gradient descent explore a flexible search space and discover activation functions that adapt to different network locations and stages of training.

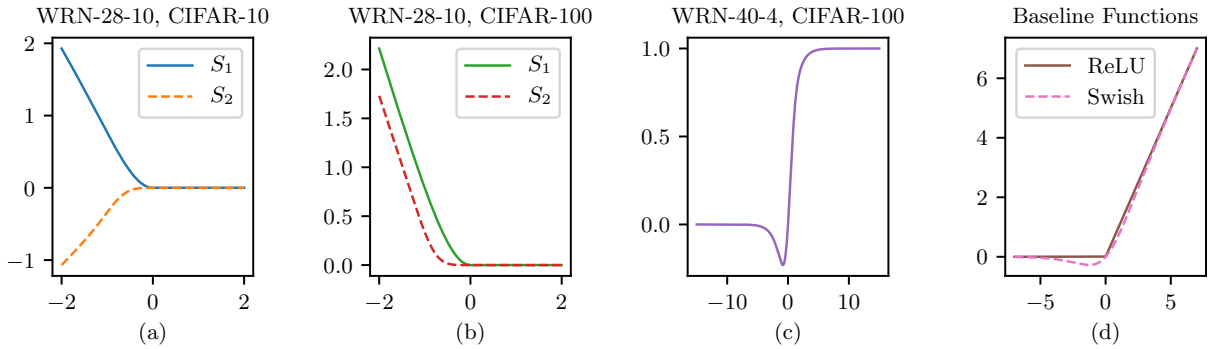


Figure 3.6: A summary of the best activation functions found in the first set of experiments. (a): Exhaustive search in S_1 discovers $(\tanh(x)) * (\min\{x, 0\})$, and evolution in S_2 discovers $(e^{(\min\{\mathrm{erf}(x),0\})-(\max\{x,0\})}) * (\min\{(\arctan((x)^3)) * (\max\{|x|, 0\}), 0\})$. Both functions achieve a median test accuracy of 94.1 with WRN-28-10 on CIFAR-10, outperforming that of ReLU (94.0) and Swish (93.8). (b): The functions $(\arctan(x)) * (\min\{x, 0\})$ in S_1 and $(-((\arctan((x)^3))*(\cos(1))))*(-((\arctan(\min\{x, 0\}))*(\max\{|x|, 0\})))$ in S_2 outperform ReLU and Swish by statistically significant margins with WRN-28-10 on CIFAR-100, demonstrating the power of novel activation functions. (c): Functions evolved for WRN-28-10 on CIFAR-10 perform well with WRN-40-4 on CIFAR-100, but a new function discovered specifically for WRN-40-4 on CIFAR-100, $\sigma(x) * \mathrm{erf}(x)$, achieves even higher performance. This result shows that the biggest advantage of activation function search is the ability to discover functions that are specialized to the architecture and dataset. (d): The baseline activation functions, ReLU and Swish, are included for visual comparison.

Chapter 4

PANGAEA: Discovering Parametric Activation Functions

This chapter describes an extended approach for discovering activation functions called PANGAEA (Parametric ActivatioN functions Generated Automatically by an Evolutionary Algorithm) [15]. Chapter 3 explored activation function discovery with multiple search algorithms and search spaces, and identified evolution in large spaces as a creative and powerful approach. In order to understand the full potential of such a system, this chapter relaxes design constraints and scales up in multiple ways. The activation function representation is expanded to include arbitrary computation graphs, which allows for representing more complex functional forms. Additional mutation operators are then introduced to allow for more efficient exploration of this search space. The approach utilizes a synergy of two different optimization processes: evolutionary population-based search discovers the general form of the activation function, and gradient descent introduces further flexibility by fine-tuning the shape of the function across locations in a network and over time as training progresses. PANGAEA discovers general activation functions that improve performance overall over previously-proposed functions. It also produces specialized functions for different architectures, such as Wide ResNet, ResNet, and Preactivation ResNet, that perform even better than the general functions, demonstrating its ability to customize activation functions to architectures.

4.1 The PANGAEA Method

Activation functions in PANGAEA are represented as computation graphs, which allow for comprehensive search, efficient implementation, and effective parameterization. Regularized evolution with reranking is used as the search method to encourage exploration and to reduce noise.

4.1.1 Representing and Modifying Activation Functions

Activation functions are represented as computation graphs in which each node is a unary or a binary operator (Table 4.1). All of these operators have TensorFlow [1] implementations, which allows for taking advantage of under-the-hood optimizations. Safe operator implementations are chosen when possible (e.g. the binary operator x_1/x_2 is implemented as `tf.math.divide_no_nan`, which returns 0 if $x_2 = 0$). Operators that are periodic (e.g. $\sin(x)$) and operators that contain repeated asymptotes are not included; in preliminary experiments they often caused training instability. All of the operators have domain $\mathbb{R}$, making it possible to compose them arbitrarily. The operators in Table 4.1 were chosen to create a large and expressive search space that contains activation functions unlikely to be discovered by hand. Indeed, all piecewise real analytic functions can be represented with a PANGAEA computation graph (Theorem 1 of Section 4.8).

PANGAEA begins with an initial population of P random activation functions. Each function is either of the form $f(x) = \texttt{unary1}(\texttt{unary2}(x))$ or $f(x) = \texttt{binary}(\texttt{unary1}(x), \texttt{unary2}(x))$, as shown in Figure 4.1. Both forms are equally likely, and the unary and binary operators are also selected uniformly at random. The computation graphs in Figure 4.1 represent the simplest non-trivial computation graphs with and without a binary operator.

Table 4.1: The operator search space consists of basic unary and binary functions as well as existing activation functions (Section 4.7). $\sigma(x) = (1 + e^{-x})^{-1}$. The unary operators bessel_i0e and bessel_i1e are the exponentially scaled modified Bessel functions of order 0 and 1, respectively.

Unary							**Binary**	
0	$\lvert x \rvert$	$\mathrm{erf}(x)$	$\tanh(x)$	$\mathrm{arcsinh}(x)$	$\mathrm{ReLU}(x)$	$\mathrm{Softplus}(x)$	$x_1 + x_2$	$x_1^{x_2}$
1	x^{-1}	$\mathrm{erfc}(x)$	$e^x - 1$	$\arctan(x)$	$\mathrm{ELU}(x)$	$\mathrm{Softsign}(x)$	$x_1 - x_2$	$\max\{x_1, x_2\}$
x	x^2	$\sinh(x)$	$\sigma(x)$	$\mathrm{bessel_i0e}(x)$	$\mathrm{SELU}(x)$	$\mathrm{HardSigmoid}(x)$	$x_1 \cdot x_2$	$\min\{x_1, x_2\}$
$-x$	e^x	$\cosh(x)$	$\log(\sigma(x))$	$\mathrm{bessel_i1e}(x)$	$\mathrm{Swish}(x)$		x_1/x_2	

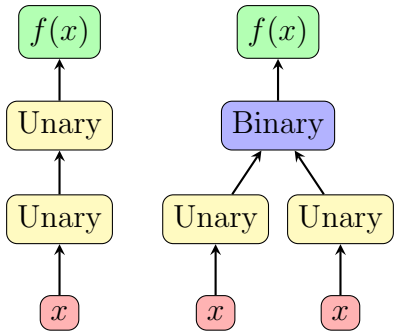


Figure 4.1: Random activation function initialization. The initial population consists of random samples of two kinds of computation graphs, randomly initialized with the operators in Table 4.1. In this manner, the search starts with simple graphs and gradually expands to more complex forms.

This design choice is inspired by previous work in neuroevolution, which demonstrated the power of starting from simple structures and gradually complexifying them [176].

During the search, all ReLU activation functions in a given neural network are replaced with a candidate activation function. No other changes to the network or training setup are made. The network is trained on the dataset, and the activation function is assigned a fitness score equal to the network's accuracy on the validation set.

Given a parent activation function, a child activation function is created by applying

one of four possible mutations (Figure 4.2). The possible mutations include elementary graph modifications like inserting, removing, or changing a node. These mutations are useful for local exploration. A special "regenerate" mutation is also introduced to accelerate exploration. Other possible evolutionary operators like crossover are not used in this chapter. All mutations are equally likely with two special cases. If a remove mutation is selected for an activation function with just one node, a change mutation is applied instead. Additionally, if an activation function with greater than seven nodes is selected, the mutation is a remove mutation, in order to reduce bloat.

Insert In an insert mutation, one operator in the search space is selected uniformly at random. This operator is placed on a random edge of a parent activation function graph. In Figure 4.2*b*, the unary operator $\text{Swish}(x)$ is inserted at the edge connecting the output of $\tanh(x)$ to the input of $x_1 + x_2$. After mutating, the parent activation function $(\tanh(x) + |\text{erf}(x)|)^2$ produces the child activation function $(\text{Swish}(\tanh(x)) + |\text{erf}(x)|)^2$. If a binary operator is randomly chosen for the insertion, the incoming input value is assigned to the variable x_1. If the operator is addition or subtraction, the input to x_2 is set to 0. If the operator is multiplication, division, or exponentiation, the input to x_2 is set to 1. Finally, if the operator is the maximum or minimum operator, the input to x_2 is a copy of the input to x_1. When a binary operator is inserted into a computation graph, the activation function computed remains unchanged. However, the structure of the computation graph is modified and can be further altered by future mutations.

Remove In a remove mutation, one node is selected uniformly at random and deleted. The node's input is rewired to its output. If the removed node is binary, one of the two inputs

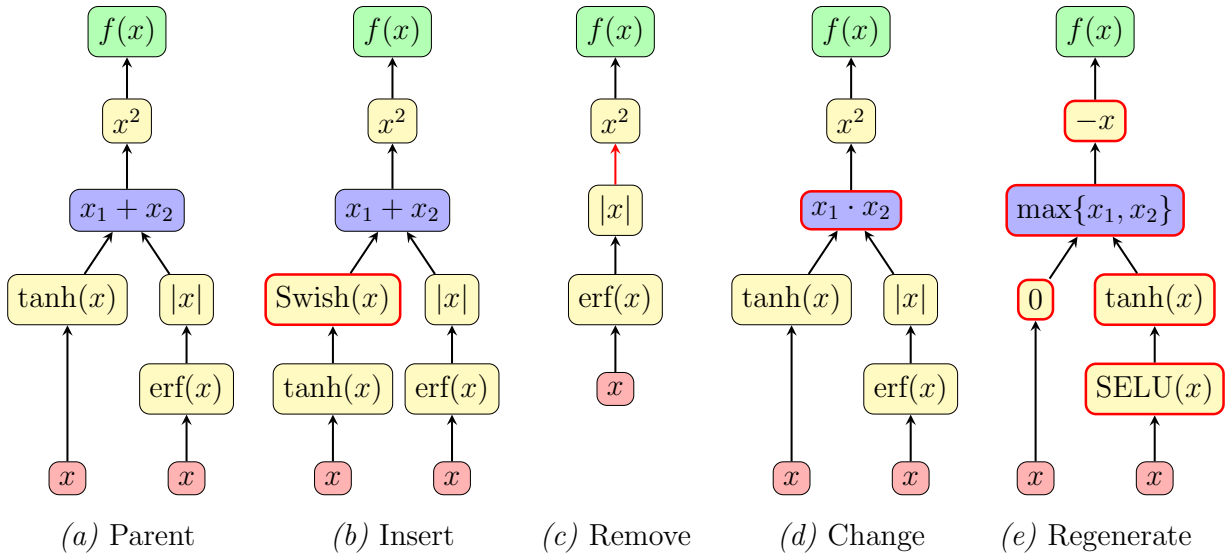


Figure 4.2: Evolutionary operations on activation functions. In an 'Insert' mutation, a new operator is inserted in one of the edges of the computation graph, like the Swish(x) in *(b)*. In a 'Remove' mutation, a node in the computation graph is deleted, like the addition in *(c)*. In a 'Change' mutation, an operator at a node is replaced with another, like addition with multiplication in *(d)*. These first three mutations are useful in refining the function locally. In contrast, in a 'Regenerate' mutation *(e)*, every operator in the graph is replaced by a random operator, thus increasing exploration.

is chosen at random and is deleted. The other input is kept. In Figure 4.2*c*, the addition operator is removed from the parent activation function. The two inputs to addition, tanh(x) and $|\text{erf}(x)|$, cannot both be kept. By chance, tanh(x) is discarded, resulting in the child activation function $|\text{erf}(x)|^2$.

Change To perform a change mutation, one node in the computation graph is selected at random and replaced with another operator from the search space, also uniformly at random. Unary operators are always replaced with unary operators, and binary operators with binary operators. Figure 4.2*d* shows how changing addition to multiplication produces the activation

function $(\tanh(x) \cdot |\mathrm{erf}(x)|)^2$.

Regenerate In a regenerate mutation, every operator in the computation graph is replaced with another operator from the search space. As with change mutations, unary operators are replaced with unary operators, and binary operators with binary operators. Although every node in the graph is changed, the overall structure of the computation graph remains the same. Regenerate mutations are useful for increasing exploration, and are similar in principle to burst mutation and delta coding [58, 194]. Figure 4.2*e* shows the child activation function $-\max\{0, \tanh(\mathrm{SELU}(x))\}$, which is quite different from the parent function in Figure 4.2*a*.

Parameterization of Activation Functions After mutation (or random initialization), activation functions are parameterized (Figure 4.3). A value $k \in \{0, 1, 2, 3\}$ is chosen uniformly at random, and k edges of the activation function graph are randomly selected. Multiplicative per-channel parameters are inserted at these edges and initialized to one. Whereas evolution is well suited for discovering the general form of the activation function in a discrete, structured search space, parameterization makes it possible to fine-tune the function using gradient descent. The function parameters are updated at every epoch during backpropagation, resulting in different activation functions in different stages of training. As the parameters are per-channel, the process creates different activation functions at different locations in the neural network. Thus, parameterization gives neural networks additional flexibility to customize activation functions.

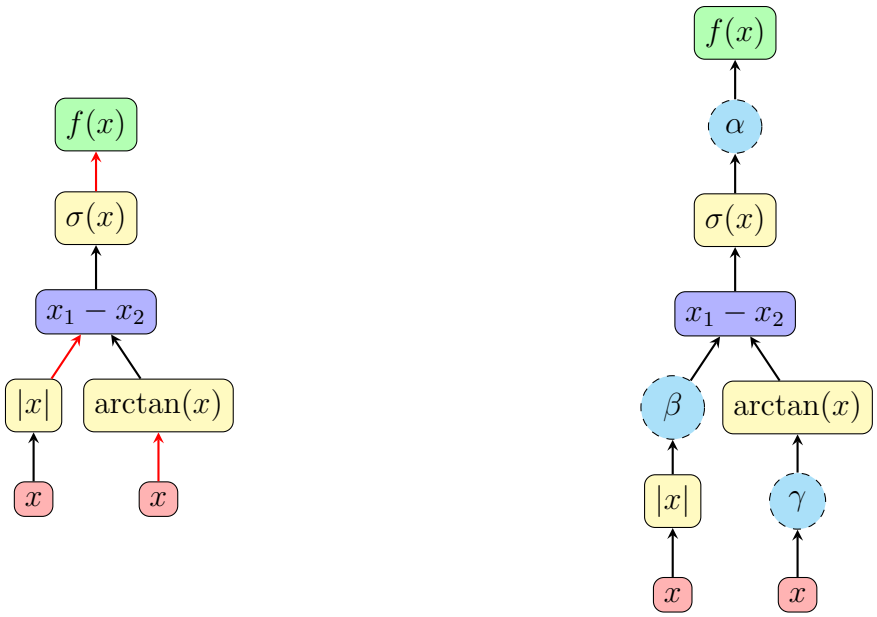


Figure 4.3: Parameterization of activation functions. In this example, parameters are added to $k = 3$ random edges, yielding the parametric activation function $\alpha\sigma(\beta|x| - \arctan(\gamma x))$.

4.1.2 Discovering Activation Functions with Evolution

Activation functions are discovered by regularized evolution [162]. Initially, P random activation functions are created, parameterized, and assigned fitness scores. To generate a new activation function, S functions are sampled with replacement from the current population. The function with the highest validation accuracy serves as the parent, and is mutated to create a child activation function. This function is parameterized and assigned a fitness score. The new activation function is then added to the population, and the oldest function in the population is removed, ensuring the population is always of size P. This process continues until C functions have been evaluated in total, and the top functions over the history of the search are returned as a result.

Any activation function that achieves a fitness score less than a threshold V is discarded. These functions are not added to the population, but they do count towards the total number of C activation functions evaluated for each architecture. This quality control mechanism allows evolution to focus only on the most promising candidates.

To save computational resources during evolution, each activation function is evaluated by training a neural network for 100 epochs using a compressed learning rate schedule. After evolution is complete, the top 10 activation functions from the entire search are reranked. Each function receives an adjusted fitness score equal to the average validation accuracy from two independent 200-epoch training runs using the original learning rate schedule. The top three activation functions after reranking proceed to the final testing experiments.

During evolution, it is possible that some activation functions achieve unusually high validation accuracy by chance. The 100-epoch compressed learning rate schedule may also have a minor effect on which activation functions are optimal compared to a full 200-epoch schedule. Reranking thus serves two purposes. Full training reduces bias from the compressed schedule, and averaging two such runs lessens the impact of activation functions that achieved high accuracy by chance.

4.2 Datasets and Architectures

The experiments in this chapter focus primarily on the CIFAR-100 image classification dataset [104]. This dataset is a more difficult version of the popular CIFAR-10 dataset, with 100 object categories instead of 10. Fifty images from each class were randomly selected from the training set to create a balanced validation set, resulting in a training/validation/test split of 45K/5K/10K images.

To demonstrate that PANGAEA can discover effective activation functions in various settings, it is evaluated with three different neural networks. The models were implemented in TensorFlow [1], mirroring the original authors' training setup as closely as possible (see Appendix A.2 for training details and Appendix A.2.2 for code that shows how to train with custom activation functions).

Wide Residual Network [WRN-10-4; 208] has a depth of 10 and widening factor of four. Wide residual networks provide an interesting comparison because they are shallower and wider than many other popular architectures, while still achieving good results. WRN-10-4 was chosen because its CIFAR-100 accuracy is competitive, yet it trains relatively quickly.

Residual Network [ResNet-v1-56; 73], with a depth of 56, provides an important contrast to WRN-10-4. It is significantly deeper and has a slightly different training setup, which may have an effect on the performance of different activation functions.

Preactivation Residual Network [ResNet-v2-56; 74] has identical depth to ResNet-v1-56, but is a fundamentally different architecture. Activation functions are not part of the skip connections, as is the case in ResNet-v1-56. Since information does not have to pass through an activation function, this structure makes it easier to train very deep architectures. PANGAEA should exploit this structure and discover different activation functions for ResNet-v2-56 and ResNet-v1-56.

4.3 Main Results

This section demonstrates that PANGAEA is able to discover good activation functions for various architectures. General activation functions that perform well on multiple

architectures are found, but the best performance comes from activation functions that are specialized to a given neural network.

4.3.1 Overview

Separate evolution experiments were run to discover novel activation functions for each of the three architectures. Evolutionary parameters $P = 64$, $S = 16$, $C = 1{,}000$, and $V = 20\%$ were used since they were found to work well in preliminary experiments.

Figure 4.4 visualizes progress in these experiments. For all three architectures, PANGAEA quickly discovered activation functions that outperform ReLU. It continued to make further progress, gradually discovering better activation functions, and did not plateau during the time allotted for the experiment. Each run took approximately 1,000 GPU hours on GeForce GTX 1080 and 1080 Ti GPUs (see Appendices A.2 and B for implementation and compute details).

Table 4.2 shows the final test accuracy for the top specialized activation functions discovered by PANGAEA in each run. For comparison, the accuracy of the top general functions discovered in this process are also shown, as well as the accuracy of several baseline activation functions (see Section 4.7 for baseline activation function details and Section 4.6 for additional results with learnable baseline functions). In sum, PANGAEA discovered the best activation function for each of the three architectures.

4.3.2 Specialized Activation Functions

For all three architectures, there are baseline activation functions that outperform ReLU by statistically significant margins. This result already demonstrates that activation

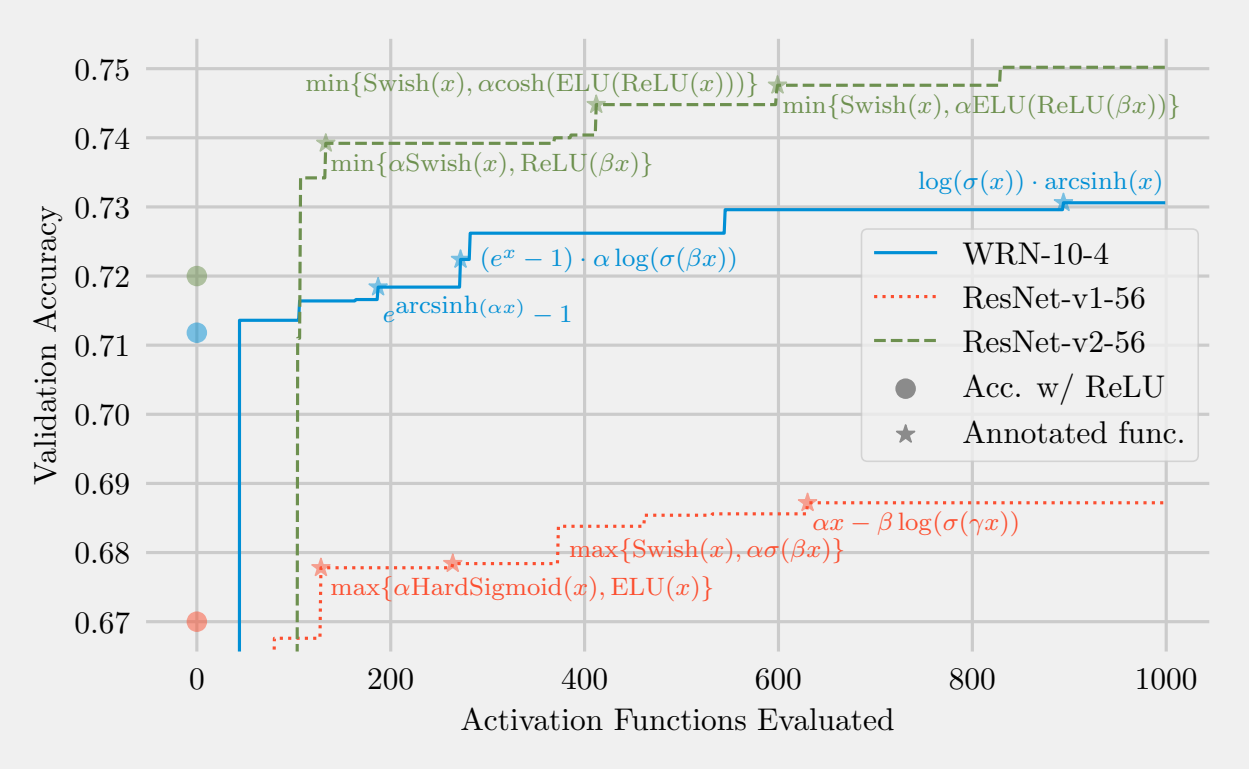


Figure 4.4: Progress of PANGAEA with three different neural networks. The plots show the best accuracy achieved among all activation functions evaluated so far. The stars on the plot specify the time when notable activation functions were discovered during evolution; the expression for each such function is written next to the star. Evolution quickly discovered activation functions that outperform ReLU (accuracy with ReLU shown at $x = 0$), and continued to improve throughout the experiment. Note that this figure shows validation accuracy, while Table 4.2 lists test set accuracy.

Table 4.2: CIFAR-100 test set accuracy aggregated over ten runs, shown as mean ± sample standard deviation. Asterisks indicate a statistically significant improvement in mean accuracy over ReLU, with * if $p \leq 0.05$, ** if $p \leq 0.01$, and *** if $p \leq 0.001$; p-values are from one-tailed Welch's t-tests. The top accuracy for each architecture is in bold. Baseline activation function details and references are given in Section 4.7.

	WRN-10-4	**ResNet-v1-56**	**ResNet-v2-56**
Specialized for WRN-10-4			
$\log(\sigma(\alpha x)) \cdot \beta \text{arcsinh}(x)$	$\mathbf{73.20}_{\pm 0.37}$ ***	$18.63_{\pm 21.04}$	$45.88_{\pm 30.70}$
$\log(\sigma(\alpha x)) \cdot \text{arcsinh}(x)$	$73.16_{\pm 0.41}$ ***	$19.34_{\pm 20.14}$	$64.30_{\pm 21.32}$
$-\text{Swish}(\text{Swish}(\alpha x))$	$72.49_{\pm 0.55}$ ***	$58.86_{\pm 2.88}$	$74.71_{\pm 0.20}$ *
Specialized for ResNet-v1-56			
$\alpha x - \beta \log(\sigma(\gamma x))$	$70.28_{\pm 0.37}$	$\mathbf{71.01}_{\pm 0.64}$ ***	$74.35_{\pm 0.45}$
$\alpha x - \log(\sigma(\beta x))$	$70.47_{\pm 0.53}$	$70.30_{\pm 0.58}$ *	$74.70_{\pm 0.23}$ *
$\max\{\text{Swish}(x), 0\}$	$72.10_{\pm 0.33}$ **	$69.43_{\pm 0.69}$	$74.97_{\pm 0.25}$ **
Specialized for ResNet-v2-56			
$\text{Softplus}(\text{ELU}(x))$	$71.36_{\pm 0.34}$	$69.96_{\pm 0.39}$	$\mathbf{75.61}_{\pm 0.42}$ ***
$\min\{\log(\sigma(x)), \alpha \log(\sigma(\beta x))\}$	$72.04_{\pm 0.34}$ **	$69.56_{\pm 0.48}$	$75.19_{\pm 0.39}$ ***
$\text{SELU}(\text{Swish}(x))$	$01.00_{\pm 0.00}$	$01.00_{\pm 0.00}$	$75.02_{\pm 0.35}$ **
General Activation Functions			
$\max\{\text{Swish}(x), \alpha \log(\sigma(\text{ReLU}(x)))\}$	$72.54_{\pm 0.26}$ ***	$69.91_{\pm 0.37}$	$75.20_{\pm 0.41}$ ***
$\min\{\text{Swish}(x), \alpha \text{ELU}(\text{ReLU}(\beta x))\}$	$72.39_{\pm 0.29}$ ***	$69.82_{\pm 0.40}$	$75.27_{\pm 0.38}$ ***
$\log(\sigma(x))$	$72.33_{\pm 0.32}$ ***	$69.58_{\pm 0.35}$	$75.53_{\pm 0.37}$ ***
Fixed Baseline Functions			
ReLU	$71.46_{\pm 0.50}$	$69.64_{\pm 0.65}$	$74.39_{\pm 0.44}$
ELiSH	$01.00_{\pm 0.00}$	$01.00_{\pm 0.00}$	$75.20_{\pm 0.31}$ ***
ELU	$72.30_{\pm 0.32}$ ***	$69.67_{\pm 0.46}$	$74.95_{\pm 0.30}$ **
GELU	$71.95_{\pm 0.35}$ *	$70.19_{\pm 0.40}$ *	$74.86_{\pm 0.33}$ **
HardSigmoid	$54.99_{\pm 1.00}$	$32.55_{\pm 4.06}$	$64.90_{\pm 0.69}$
Leaky ReLU	$71.73_{\pm 0.33}$	$69.78_{\pm 0.33}$	$74.73_{\pm 0.35}$ *
Mish	$71.95_{\pm 0.41}$ *	$69.88_{\pm 0.54}$	$75.32_{\pm 0.29}$ ***
SELU	$70.53_{\pm 0.42}$	$68.52_{\pm 0.29}$	$73.79_{\pm 0.36}$
sigmoid	$56.10_{\pm 0.98}$	$36.47_{\pm 3.32}$	$66.45_{\pm 0.92}$
Softplus	$72.27_{\pm 0.26}$ ***	$69.71_{\pm 0.36}$	$75.46_{\pm 0.52}$ ***
Softsign	$56.30_{\pm 2.16}$	$58.38_{\pm 0.96}$	$69.33_{\pm 0.39}$
Swish	$72.26_{\pm 0.28}$ ***	$69.68_{\pm 0.38}$	$75.08_{\pm 0.36}$ ***
tanh	$56.52_{\pm 1.53}$	$63.88_{\pm 0.38}$	$70.44_{\pm 0.40}$
Parametric Baseline Functions			
PReLU	$72.23_{\pm 0.37}$ ***	$69.77_{\pm 0.40}$	$75.10_{\pm 0.53}$ **
$\text{PSwish} = x \cdot \sigma(\beta x)$	$72.40_{\pm 0.31}$ ***	$70.16_{\pm 0.46}$ *	$75.39_{\pm 0.28}$ ***
Learnable Baseline Functions			
APL	$72.88_{\pm 0.32}$ ***	$70.81_{\pm 0.20}$ ***	$75.02_{\pm 0.28}$ ***
PAU	$41.46_{\pm 22.66}$	$01.00_{\pm 0.00}$	$02.38_{\pm 4.36}$
SPLASH	$72.16_{\pm 0.81}$ *	$01.00_{\pm 0.00}$	$73.45_{\pm 0.43}$

functions should be chosen carefully, and that the common practice of using ReLU by default is suboptimal. Furthermore, the best baseline activation function is different for different architectures, suggesting that specializing activation functions to the architecture is a good approach.

Because PANGAEA uses validation accuracy from a single neural network to assign fitness scores to activation functions, there is selective pressure to discover functions that exploit the structure of the network. The functions thus become specialized to the architecture. They increase the performance of that architecture; however, they may not be as effective with other architectures. Specialized activation function accuracies are highlighted with the gray background in Table 4.2. To verify that the functions are customized to a specific architecture, the functions were cross-evaluated with other architectures.

PANGAEA discovered two specialized activation functions for WRN-10-4 that outperformed all baseline functions by a statistically significant margin ($p \leq 0.05$). The top specialized function on ResNet-v1-56 also significantly outperformed all baseline functions, except APL (for which $p = 0.19$). The top specialized activation function on ResNet-v2-56 similarly significantly outperformed all except Softplus ($p = 0.25$) and PSwish ($p = 0.09$). These results strongly demonstrate the power of customizing activation functions to architectures. Indeed, specializing activation functions is a new dimension of activation function search not considered by previous work [11, 160].

4.3.3 General Activation Functions

Although the best performance comes from specialization, it is also useful to discover activation functions that achieve high accuracy across multiple architectures. For instance,

they could be used initially on a new architecture before spending compute on specialization. A powerful albeit computationally demanding approach would be to evolve general functions directly, by evaluating candidates on multiple architectures during evolution. However, it turns out that each specialized evolution run already generates a variety of functions, many of which are general.

To evaluate whether the PANGAEA runs discovered general functions as well, the top 10 functions from each run were combined into a pool of 30 candidate functions. Each candidate was assigned three fitness scores equal to the average validation accuracy from two independent training runs on each of the three architectures. Candidate functions that were Pareto-dominated, were functionally equivalent to one of the baseline activation functions, or had already been selected as a specialized activation function were discarded, leaving three Pareto-optimal general activation functions.

These functions indeed turned out to be effective as general activation functions. All three achieved good accuracy with ResNet-v1-56 and significantly outperformed ReLU with WRN-10-4 and ResNet-v2-56. However, specialized activation functions, i.e. those specifically evolved for each architecture, still give the biggest improvements.

4.3.4 Shapes of Discovered Functions

Many of the top discovered activation functions are compositions of multiple unary operators. These functions do not exist in the core unit search space of Ramachandran et al. [160], which requires binary operators. They also do not exist in the S_1 or S_2 search spaces in CAFE, which are too shallow. The design of the search space is therefore as important as the search algorithm itself. Previous search spaces that rely on repeated fixed building blocks

only have limited representational power. In contrast, PANGAEA utilizes a flexible search space that can represent activation functions in an arbitrary computation graph (see Section 4.9 for an analysis on the size of the PANGAEA search space).

Furthermore, while the learnable baseline functions can in principle approximate the functions discovered by PANGAEA, they do not consistently match its performance. PANGAEA utilizes both evolutionary search and gradient descent to discover activation functions, and apparently this combination of optimization processes is more powerful than gradient descent alone.

Figure 4.5 shows examples of parametric activation functions discovered by PANGAEA. As training progresses, gradient descent makes small adjustments to the function parameters α, β, and γ, resulting in activation functions that change over time. This result suggests that it is advantageous to have one activation function in the early stages of training when the network learns rapidly, and a different activation function in the later stages of training when the network is focused on fine-tuning. The parameters α, β, and γ are also learned separately for the different channels, resulting in activation functions that vary with location in a neural network. Functions in deep layers (near the output) are more nonlinear than those in shallow layers (closer to the input), possibly contrasting the need to form regularized embeddings with the need to form categorizations. In this manner, PANGAEA customizes the activation functions to both time and space for each architecture.

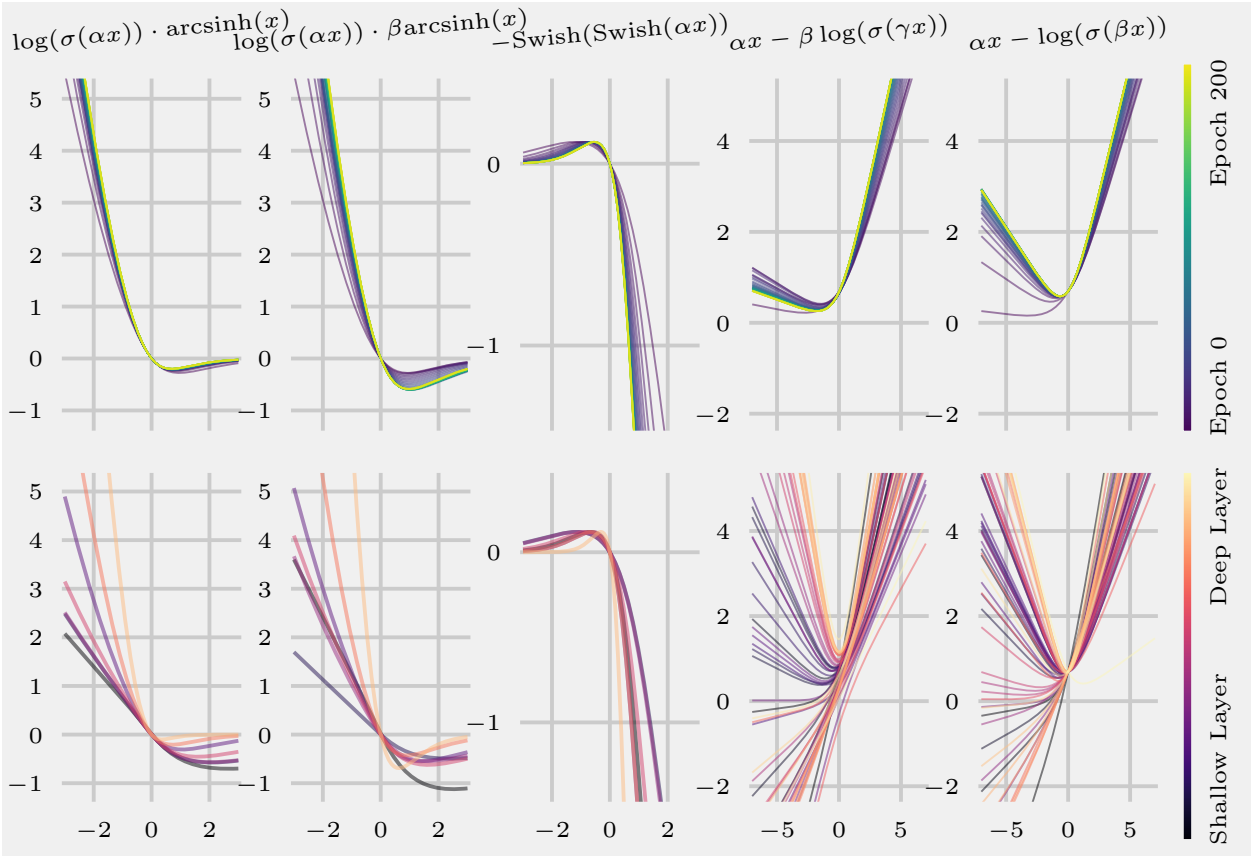


Figure 4.5: Adaptation of parametric activation functions over time and space. **Top:** The parameters change during training, resulting in different activation functions in the early and late stages. The plots were created by averaging the values of α, β, and γ across the entire network at different training epochs. **Bottom:** The parameters are updated separately in each channel, inducing different activation functions at different locations of a neural network. The plots were created by averaging α, β, and γ at each layer of the network after the completion of training.

4.4 Diving Deeper: Experiments on Ablations, Variations, and Training Dynamics

PANGAEA is a method with many moving parts. The main results from Section 4.3 already showed the power of PANGAEA, and the experiments from this section illuminate how each component of PANGAEA contributes to its success. Evolutionary search and gradient descent working in tandem provided a better strategy than either optimization algorithm alone (Section 4.4.1). The top activation functions benefitted from their learnable parameters (Section 4.4.2), but baseline functions did not (Section 4.4.3), showing how PANGAEA discovered functional forms well-suited to parameterization. PANGAEA is robust: the activation functions it discovers transfer to larger networks (Section 4.4.4) and PANGAEA achieves impressive results with a new dataset and architecture (Section 4.4.5). The activation functions discovered by PANGAEA improve accuracy by easing optimization and implicitly regularizing the network (Section 4.4.6).

4.4.1 Additional Baseline Search Strategies

As additional baseline comparisons, two alternative search strategies were used to discover activation functions for WRN-10-4. First, a random search baseline was established by applying random mutations without regard to fitness values. This approach corresponds to setting evolutionary parameters $P = 1$, $S = 1$, and $V = 0\%$. Second, to understand the effects of function parameterization, a nonparametric evolution baseline was run. This setting is identical to PANGAEA, except functions are not parameterized (Figure 4.3). Otherwise, both baselines follow the same setup as PANGAEA, including evaluating $C = 1{,}000$ candidate functions and reranking the most promising ones (Section 4.1.2).

Table 4.3: WRN-10-4 accuracy with different activation functions on CIFAR-100, shown as mean ± sample standard deviation across ten runs. PANGAEA discovers better activation functions than random search and nonparametric evolution.

PANGAEA	
$\log(\sigma(\alpha x)) \cdot \beta \text{arcsinh}(x)$	$\mathbf{73.20}_{\pm 0.37}$
$\log(\sigma(\alpha x)) \cdot \text{arcsinh}(x)$	$73.16_{\pm 0.41}$
$-\text{Swish}(\text{Swish}(\alpha x))$	$72.49_{\pm 0.55}$
Random Search	
$\alpha \text{Swish}(x)$	$72.85_{\pm 0.25}$
$\text{Softplus}(x) \cdot \arctan(\alpha x)$	$72.81_{\pm 0.35}$
$\text{ReLU}(\alpha \text{arcsinh}(\beta \sigma(x))) \cdot \text{SELU}(\gamma x)$	$72.69_{\pm 0.21}$
Nonparametric Evolution	
$\cosh(1) \cdot \text{Swish}(x)$	$72.78_{\pm 0.24}$
$(e^1 - 1) \cdot \text{Swish}(x)$	$72.52_{\pm 0.34}$
$\text{ReLU}(\text{Swish}(x))$	$72.04_{\pm 0.54}$
ReLU	$71.46_{\pm 0.50}$
Swish	$72.26_{\pm 0.28}$

Table 4.3 shows the results of this experiment. Random search is able to discover good functions that outperform ReLU, but the functions are not as powerful as those discovered by PANGAEA. This result demonstrates the importance of fitness selection in evolutionary search. The functions discovered by nonparametric evolution similarly outperform ReLU but underperform PANGAEA. Interestingly, without parameterization, evolution is not as creative: two of the three functions discovered are merely Swish multiplied by a constant. Random search and nonparametric evolution both discovered good functions that improved accuracy, but PANGAEA achieves the best performance by combining the advantages of fitness selection and function parameterization.

Table 4.4: CIFAR-100 test set accuracy aggregated over ten runs, shown as mean ± sample standard deviation. The parametric evolved functions tend to outperform their nonparametric counterparts, demonstrating the value of parameterization.

WRN-10-4	
$\log(\sigma(\alpha x)) \cdot \beta \text{arcsinh}(x)$	$\mathbf{73.20}_{\pm 0.37}$
$\log(\sigma(\alpha x)) \cdot \text{arcsinh}(x)$	$73.16_{\pm 0.41}$
$\log(\sigma(x)) \cdot \text{arcsinh}(x)$	$72.51_{\pm 0.30}$
$-\text{Swish}(\text{Swish}(\alpha x))$	$\mathbf{72.49}_{\pm 0.55}$
$-\text{Swish}(\text{Swish}(x))$	$71.97_{\pm 0.22}$
ResNet-v1-56	
$\alpha x - \beta \log(\sigma(\gamma x))$	$\mathbf{71.01}_{\pm 0.64}$
$\alpha x - \log(\sigma(\beta x))$	$70.30_{\pm 0.58}$
$x - \log(\sigma(x))$	$69.29_{\pm 0.45}$
ResNet-v2-56	
$\min\{\log(\sigma(x)), \alpha \log(\sigma(\beta x))\}$	$75.19_{\pm 0.39}$
$\log(\sigma(x))$	$\mathbf{75.53}_{\pm 0.37}$

4.4.2 Effect of Parameterization

To understand the effect that parameterizing activation functions has on their performance, the specialized functions (Table 4.2) were trained without them. As Table 4.4 shows, when parameters are removed, performance drops. The function $\log(\sigma(x))$ is the only exception to this rule, but its high performance is not surprising, since it was previously discovered as a general activation function (Table 4.2). These results confirm that the learnable parameters contributed to the success of PANGAEA.

4.4.3 Adding Parameters to Fixed Baseline Activation Functions

As demonstrated in Tables 4.2 and 4.4, learnable parameters are an important component of PANGAEA. An interesting question is whether accuracy can be increased simply

by augmenting existing activation functions with learnable parameters. Table 4.5 shows that this is not the case: trivially adding parameters to fixed activation functions does not reliably improve performance. This experiment implies that certain functional forms are better suited to taking advantage of parameterization than others. By utilizing evolutionary search, PANGAEA is able to discover these functional forms automatically.

4.4.4 Scaling Up to Larger Networks

PANGAEA discovered specialized activation functions for WRN-10-4, ResNet-v1-56, and ResNet-v2-56. Table 4.6 shows the performance of these activation functions when paired with the larger WRN-16-8, ResNet-v1-110, and ResNet-v2-110 architectures. Due to time constraints, ReLU is the only baseline activation function in these experiments.

Two of the three functions discovered for WRN-10-4 outperform ReLU with WRN-16-8, and all three functions discovered for ResNet-v2-56 outperform ReLU with ResNet-v2-110. Interestingly, ReLU achieves the highest accuracy for ResNet-v1-110, where activation functions are part of the skip connections, but not for ResNet-v2-110, where they are not. Thus, it is easier to achieve high performance with specialized activation functions on very deep architectures when they are not confounded by skip connections. Notably, ResNet-v2-110 with Softplus(ELU(x)) performs comparably to the much larger ResNet-v2-1001 with ReLU (77.14 vs. 77.29, as reported by He et al. [74]).

Evolving novel activation functions can be computationally expensive. The results in Table 4.6 suggest that it is possible to reduce this cost by evolving activation functions for smaller architectures, and then using the discovered functions with larger architectures.

Table 4.5: CIFAR-100 test set accuracy aggregated over ten runs, shown as mean ± sample standard deviation. Trivially parameterizing existing fixed activation functions does not substantially improve performance. PANGAEA, on the other hand, utilizes evolutionary search to discover functional forms that are well suited to taking advantage of the parameters, leading to better performance.

	WRN-10-4	**ResNet-v1-56**	**ResNet-v2-56**
Best Specialized Functions			
$\log(\sigma(\alpha x)) \cdot \beta \text{arcsinh}(x)$	$\mathbf{73.20}_{\pm 0.37}$		
$\alpha x - \beta \log(\sigma(\gamma x))$		$\mathbf{71.01}_{\pm 0.64}$	
$\text{Softplus}(\text{ELU}(x))$			$\mathbf{75.61}_{\pm 0.42}$
Parameterized Functions			
$\alpha\text{ReLU}(\beta x)$	$71.96_{\pm 0.31}$	$68.93_{\pm 0.22}$	$73.52_{\pm 0.37}$
$\alpha\text{ELiSH}(\beta x)$	$01.00_{\pm 0.00}$	$01.00_{\pm 0.00}$	$73.94_{\pm 0.33}$
$\alpha\text{ELU}(\beta x)$	$71.98_{\pm 0.24}$	$69.06_{\pm 0.37}$	$73.97_{\pm 0.45}$
$\alpha\text{GELU}(\beta x)$	$71.96_{\pm 0.34}$	$69.39_{\pm 0.35}$	$73.83_{\pm 0.24}$
$\alpha\text{HardSigmoid}(\beta x)$	$66.70_{\pm 0.64}$	$34.33_{\pm 6.53}$	$65.10_{\pm 0.40}$
$\alpha\text{Leaky ReLU}(\beta x)$	$71.74_{\pm 0.39}$	$69.11_{\pm 0.47}$	$73.44_{\pm 0.29}$
$\alpha\text{Mish}(\beta x)$	$72.11_{\pm 0.31}$	$69.51_{\pm 0.67}$	$73.72_{\pm 0.32}$
$\alpha\text{SELU}(\beta x)$	$71.07_{\pm 0.33}$	$68.05_{\pm 0.39}$	$73.37_{\pm 0.38}$
$\alpha\text{sigmoid}(\beta x)$	$66.98_{\pm 0.66}$	$44.40_{\pm 2.62}$	$66.98_{\pm 0.85}$
$\alpha\text{Softplus}(\beta x)$	$71.73_{\pm 0.31}$	$68.84_{\pm 0.30}$	$73.95_{\pm 0.37}$
$\alpha\text{Softsign}(\beta x)$	$62.12_{\pm 0.83}$	$09.18_{\pm 13.75}$	$68.87_{\pm 0.38}$
$\alpha\text{Swish}(\beta x)$	$72.26_{\pm 0.29}$	$69.25_{\pm 0.28}$	$73.93_{\pm 0.22}$
$\alpha\tanh(\beta x)$	$63.55_{\pm 0.56}$	$02.92_{\pm 6.07}$	$69.55_{\pm 0.62}$

Table 4.6: Specialized activation functions discovered for WRN-10-4, ResNet-v1-56, and ResNet-v2-56 are evaluated on larger versions of those architectures: WRN-16-8, ResNet-v1-110, and ResNet-v2-110, respectively. CIFAR-100 test accuracy is reported as mean ± sample standard deviation across three runs. Specialized activation functions successfully transfer to WRN-16-8 and ResNet-v2-110, outperforming ReLU.

WRN-16-8	
$\log(\sigma(\alpha x)) \cdot \beta \text{arcsinh}(x)$	$\mathbf{78.36}_{\pm 0.17}$
$\log(\sigma(\alpha x)) \cdot \text{arcsinh}(x)$	$78.34_{\pm 0.20}$
$-\text{Swish}(\text{Swish}(\alpha x))$	$78.00_{\pm 0.35}$
ReLU	$78.15_{\pm 0.03}$
ResNet-v1-110	
$\alpha x - \beta \log(\sigma(\gamma x))$	$70.85_{\pm 0.50}$
$\alpha x - \log(\sigma(\beta x))$	$70.34_{\pm 0.60}$
$\max\{\text{Swish}(x), 0\}$	$70.36_{\pm 0.56}$
ReLU	$\mathbf{71.23}_{\pm 0.25}$
ResNet-v2-110	
$\text{Softplus}(\text{ELU}(x))$	$\mathbf{77.14}_{\pm 0.38}$
$\min\{\log(\sigma(x)), \alpha \log(\sigma(\beta x))\}$	$76.93_{\pm 0.19}$
$\text{SELU}(\text{Swish}(x))$	$76.96_{\pm 0.14}$
ReLU	$76.34_{\pm 0.11}$

4.4.5 A New Task: All-CNN-C on CIFAR-10

To verify that PANGAEA is effective with different datasets and types of architectures, activation functions were evolved for the All-CNN-C [174] architecture on the CIFAR-10 dataset. All-CNN-C is quite distinct from the architectures considered above: It contains only convolutional layers, activation functions, and a global average pooling layer, but it does not have residual connections.

As shown in Table 4.7, PANGAEA improves significantly over ReLU in this setting as well. The accuracy improvement from 88.47% to 92.77% corresponds to an impressive 37.29%

Table 4.7: All-CNN-C accuracy with different activation functions on CIFAR-10, shown as mean ± sample standard deviation across ten runs. PANGAEA improves performance significantly also with this different architecture and task.

$\alpha\mathrm{ReLU}(\beta\lvert\mathrm{ReLU}(\gamma x)\rvert)$	$\mathbf{92.77}_{\pm 0.13}$
$\alpha\mathrm{Swish}(x)\cdot\cosh(\beta)$	$92.66_{\pm 0.08}$
$\alpha\mathrm{Swish}(\beta x)$	$76.15_{\pm 34.86}$
ReLU	$88.47_{\pm 0.14}$

reduction in the error rate. This experiment provides further evidence that PANGAEA can improve performance for different architectures and tasks.

4.4.6 Training Dynamics of Evolved Activation Functions

PANGAEA discovers activation functions that improve accuracy over baseline functions. An interesting question is: What mechanisms do these evolved functions use in order to achieve better performance? By examining training and validation curves qualitatively for different activation functions, it appears that some functions ease optimization, while others improve performance through implicit regularization.

For instance, Figure 4.6 shows training and validation curves for the All-CNN-C architecture and four discovered activation functions, plus ReLU. With All-CNN-C, the learning rate starts at 0.01 and decreases by a factor of 0.1 after epochs 200, 250, and 300, with training ending at epoch 350. With some discovered activation functions, the training and validation curves are consistently higher than the curves from ReLU across all epochs of training, indicating easier optimization (Figure 4.6a). With other activation functions, the training and validation curves actually remain lower than those from ReLU until the final stage in the learning rate schedule, suggesting implicit regularization (Figure 4.6b). In such

cases, the network is learning difficult patterns in the early stages of training; in contrast, the ReLU model memorizes simpler patterns, which leads to early gains but difficulty generalizing later on [112].

Even more complex behavior can be observed in some cases. For example, some discovered functions have training and validation curves that start out higher than those from ReLU, plateau to a lower value, but then again surpass those from ReLU at later stages in the learning rate schedule (Figure 4.6c). Others have curves that start out lower than those from ReLU, but surpass it within a few dozen epochs (Figure 4.6d). Such diverse behavior suggests that these mechanisms can be combined in complex ways. Thus, the plots in Figure 4.6 suggest that in the future it may be feasible to search for activation functions with specific properties depending on the task at hand. For example, a larger network may benefit from an activation function that implicitly regularizes it, while a smaller network may be better suited to an activation function that eases optimization.

4.5 Evaluating Robustness: Experiments on Reliability, Flexibility, and Efficiency

This section demonstrates robustness of PANGAEA with three experiments. In the first experiment, two independent PANGAEA processes were run from scratch. The processes discovered similarly powerful activation functions, demonstrating that PANGAEA reliably discovers good activation functions each time it is run. In the second experiment, variations of PANGAEA with per-layer and per-neuron learnable parameters were run, to complement the original PANGAEA with per-channel parameters. The results show that PANGAEA is effective with all three variations. Third, statistics from activation functions

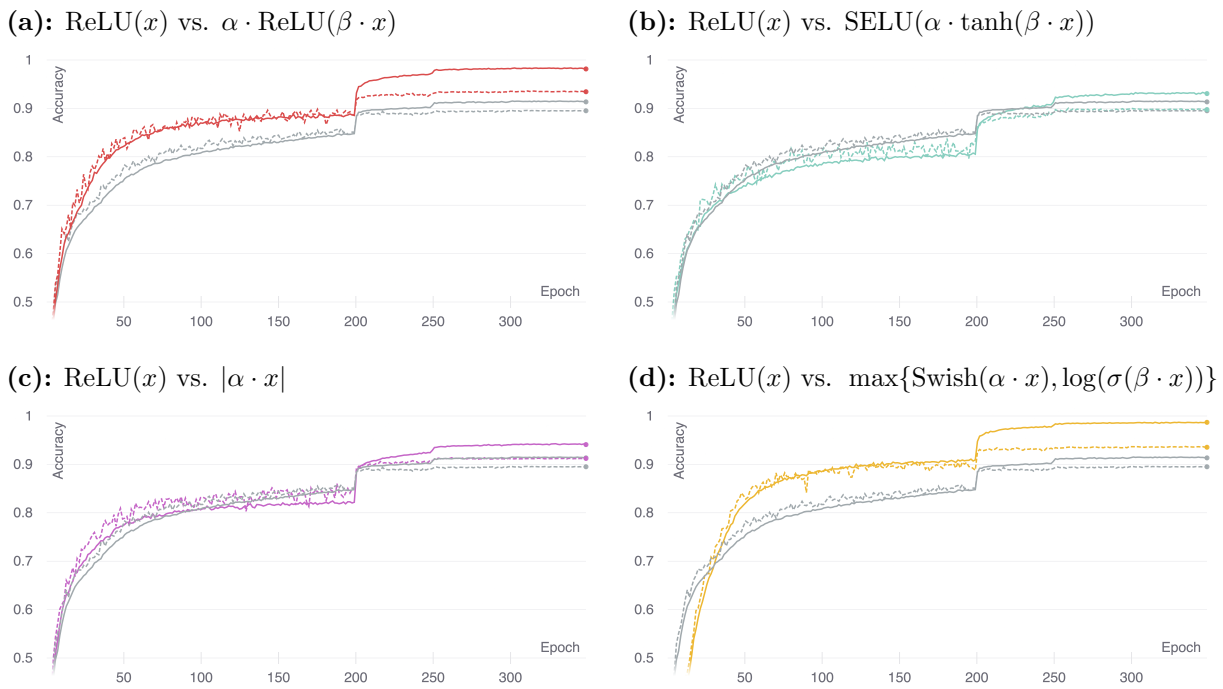


Figure 4.6: Training curves of All-CNN-C on CIFAR-10 with different activation functions. Accuracy with ReLU is shown in gray, and accuracy with the discovered functions in different colors. The training accuracy is shown as a solid line, and validation accuracy as a dashed line. All of these discovered functions outperformed ReLU, but the training curves show different behavior in each case. In (a) both curves are above those of ReLU the whole time, suggesting ease of learning. In (b), they exceed those of ReLU only in the end, suggesting early regularization. More complex profiles (such as those in (c) and (d)) are also observed, suggesting that their combinations are possible as well.

across the per-layer, two per-channel, and the per-neuron variations were aggregated to demonstrate computational efficiency of PANGAEA. Every PANGAEA run discovers an activation function that outperforms ReLU early on in the search process, before much compute is spent; they also eventually discover activation functions that perform much better and train almost as quickly as ReLU.

4.5.1 Reliability of PANGAEA

PANGAEA is inherently a stochastic process. Therefore, an important question is whether PANGAEA can discover good activation functions reliably every time it is run. To answer this question, PANGAEA was run from scratch on ResNet-v1-56 independently two times. These runs utilized per-channel parameters, and were identical to the original PANGAEA run, except they were allowed to evaluate up to $C = 2{,}000$ activation functions instead of $C = 1{,}000$ from the original run. They were run on ResNet-v1-56 since it proved to be the most difficult architecture to optimize (Table 4.2).

There are multiple ways to analyze the similarity of the two PANGAEA runs. First, a simple and relevant metric is to look at the test accuracies of the functions discovered by each search. Table 4.8 shows that both PANGAEA runs discovered multiple good activation functions. The accuracies achieved by these functions are substantially higher than those achieved by ReLU. Most importantly, although the functions themselves are different, they resulted in similar accuracies as functions from the original PANGAEA run.

A second way is to compare the time course of discovery. To this end, Figure 4.7 shows how the populations of $P = 64$ activation functions improved over time in the two PANGAEA runs. In both cases, the initial functions are relatively poor. As evolution progresses, both

Table 4.8: Performance of the best activation functions from multiple PANGAEA runs with ResNet-v1-56. CIFAR-100 test accuracy is shown as the mean ± sample standard deviation across three runs. The three independent per-channel runs produce activation functions of similar performance, demonstrating the reliability of PANGAEA. PANGAEA also discovers good activation functions with per-layer or per-neuron parameters, showing its flexibility. The very best per-layer and per-neuron functions are difficult to find, suggesting that their distribution is long-tailed.

Per-layer PANGAEA	
$\max\{\text{Swish}(x), x\}$	$71.00_{\pm 0.28}$
$\max\{x, \alpha \cdot \log(\sigma(\text{SELU}(x)))\}$	$70.76_{\pm 0.29}$
$\max\{\alpha \cdot \max\{\beta \cdot \text{ReLU}(\text{arcsinh}(x)), x\}, \max\{\gamma \cdot \text{ReLU}(\text{arcsinh}(x)), x\}\}$	$70.63_{\pm 0.35}$
Original Per-channel PANGAEA Run	
$\alpha x - \beta \log(\sigma(\gamma x))$	$71.01_{\pm 0.64}$
$\alpha x - \log(\sigma(\beta x))$	$70.30_{\pm 0.58}$
$\max\{\text{Swish}(x), 0\}$	$69.43_{\pm 0.69}$
Additional Per-channel PANGAEA Run 1	
$\max\{\min\{\alpha \cdot x, \text{ELU}(x)\}, 0\}$	$70.53_{\pm 0.31}$
$\alpha \cdot \max\left\{\beta \cdot \text{ReLU}\left(\frac{\text{Swish}(x)}{\gamma}\right), x\right\}$	$70.52_{\pm 0.39}$
$\max\{\text{ReLU}(\text{Swish}(\alpha \cdot x)), \beta \cdot x\}$	$70.44_{\pm 0.44}$
Additional Per-channel PANGAEA Run 2	
$\max\{\text{Swish}(\alpha \cdot x), x\}$	$71.03_{\pm 0.40}$
$\max\{\text{Swish}(x), \text{arcsinh}(\alpha \cdot \beta \cdot x)\}$	$70.52_{\pm 0.35}$
$\max\{\text{Swish}(x), \text{arcsinh}(\alpha \cdot x)\}$	$70.41_{\pm 0.38}$
Per-neuron PANGAEA	
$\alpha \cdot \max\{\beta \cdot \text{Swish}(\gamma \cdot x), \text{Swish}(x)\}$	$71.25_{\pm 0.35}$
$\alpha \cdot x - (\beta \cdot \text{Swish}(\gamma \cdot x))$	$71.23_{\pm 0.18}$
$\text{ELU}(\text{ELU}(\alpha \cdot x) + \log(\sigma(0)))$	$71.20_{\pm 0.25}$
Random Sample ($n = 500$)	
Per-layer $\max\{\alpha \cdot \text{Swish}(\beta \cdot x), \text{Softplus}(x)\}$	70.32
Per-channel $\alpha \cdot x^2$	70.91
Per-neuron $\text{bessel_i0e}(\lvert x \rvert) + \alpha \cdot \lvert x \rvert$	**71.66**
ReLU	$69.64_{\pm 0.65}$

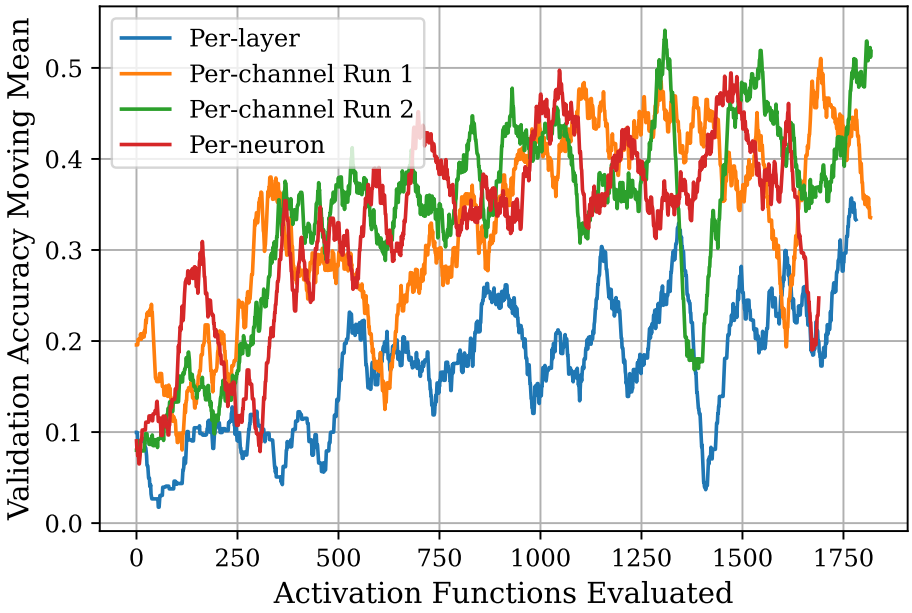


Figure 4.7: Average population fitness across time for four independent PANGAEA runs. The plots show the average validation accuracy achieved with the 64 most recently evaluated activation functions at any given time in the search process. All four PANGAEA runs gradually discover better activation functions as they explore the search space, with the per-layer run slightly below the others. Importantly, the two per-channel PANGAEA runs progress at similar rates, demonstrating the reliability of PANGAEA.

runs discover better functions at similar rates. This result shows that in addition to the final results, the PANGAEA process as a whole is stable and reliable.

A third way is to compare the complexity, i.e. time it takes to train the network with the discovered functions. Figure 4.8 shows the cost of all of the activation functions considered throughout each PANGAEA run. The runtimes of activation functions are comparable across different validation accuracies, suggesting that both runs discovered functions of similar complexity.

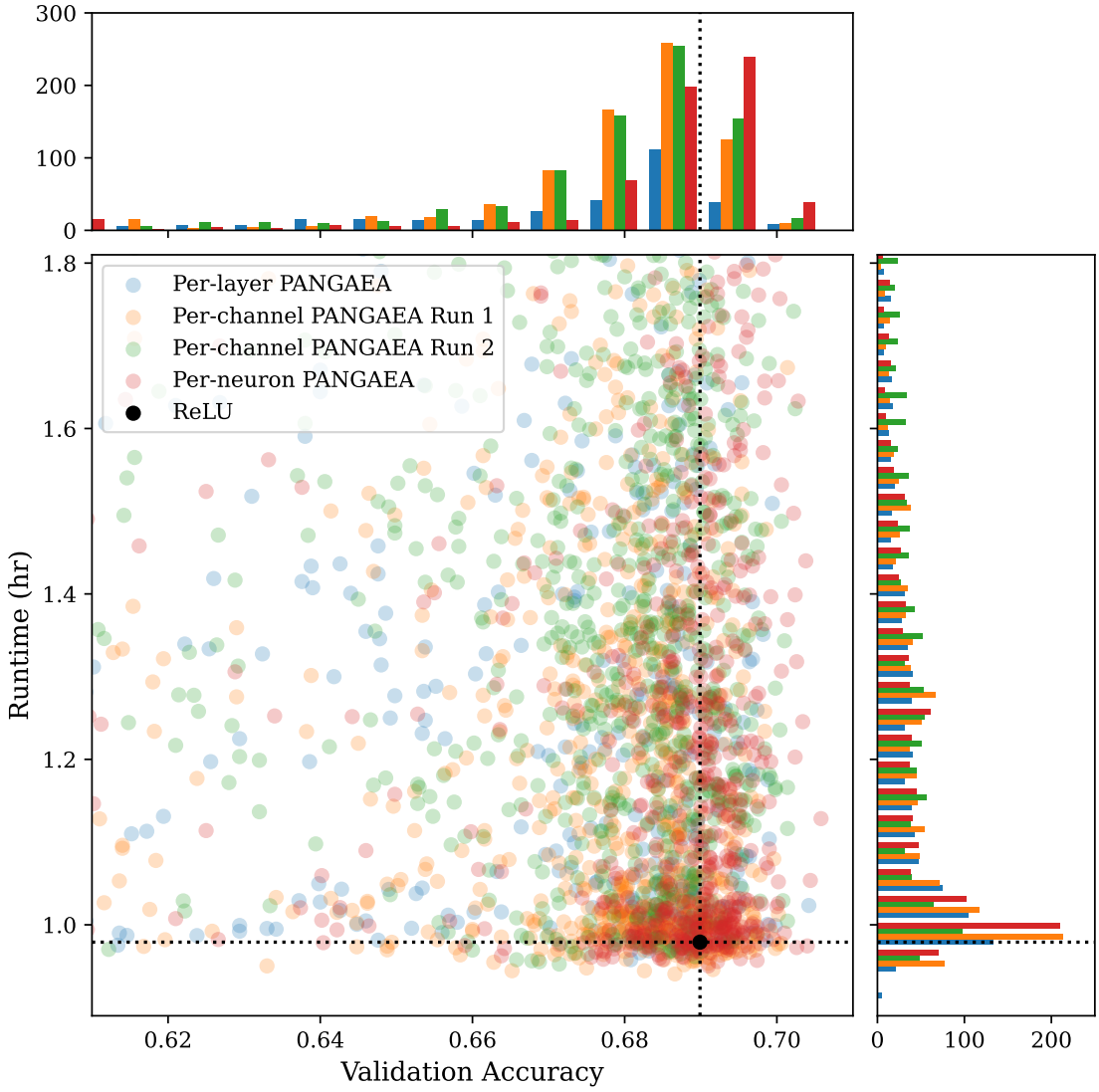


Figure 4.8: Fitness (validation accuracy) and compute cost (runtime in hours) among all activation functions considered in four independent PANGAEA processes on ResNet-v1-56. Each point represents a different activation function. The distribution of fitness and compute cost are shown in histograms on the top and right, respectively. All PANGAEA variants explore activation functions of similar complexity and reliably discover many novel functions that outperform ReLU.

In sum, the two PANGAEA runs produced comparable final results, progressed at comparable rates, and searched through functions of similar complexity. These results suggest that PANGAEA is a reliable process that can consistently outperform baseline activation functions.

4.5.2 Parameters: per-layer, per-channel, or per-neuron?

Learnable parameters in activation functions can be per-layer, per-channel, or per-neuron. It is not clear which setting is the best. For example, per-neuron parameters are the default setting in the TensorFlow implementation of PReLU [72]. However, He et al. [72] also experimented with per-channel and per-layer implementations. Further preliminary experiments for this chapter (Table 4.9) suggest that per-neuron PReLU is best for WRN-10-4 and ResNet-v2-56, but this setting is the worst for ResNet-v1-56, which benefits most from per-layer parameters.

Similarly, no clear trends were observed in preliminary PANGAEA experiments. For some activation functions and architectures per-neuron parameters were beneficial, presumably due to the added expressivity of each neuron learning its own optimal activation function. In other cases per-layer was better, possibly due to an implicit regularization effect caused by all neurons within a layer using the same activation function. As a compromise between the expressivity and regularization of these two strategies, per-channel parameters were utilized in the main experiments. However, the preliminary results suggest that performance may be further optimized by specializing the parameter setting to the activation function and to the architecture.

To explore this idea further, per-layer and per-neuron versions of PANGAEA were

Table 4.9: CIFAR-100 test accuracy with different architectures and PReLU variants reported as mean ± sample standard deviation across ten runs. Per-neuron PReLU gets the best performance on WRN-10-4 and ResNet-v2-56, but per-layer PReLU is the best for ResNet-v1-56.

	WRN-10-4	**ResNet-v1-56**	**ResNet-v2-56**
Per-layer PReLU	$71.92_{\pm 0.41}$	$\mathbf{71.40}_{\pm 0.59}$	$73.54_{\pm 0.21}$
Per-channel PReLU	$71.15_{\pm 0.41}$	$71.25_{\pm 0.54}$	$74.52_{\pm 0.24}$
Per-neuron PReLU	$\mathbf{72.23}_{\pm 0.37}$	$69.77_{\pm 0.40}$	$\mathbf{75.10}_{\pm 0.53}$

run from scratch on ResNet-v1-56. Both of these PANGAEA runs produced good activation functions that outperformed ReLU substantially (Table 4.8). Interestingly, although per-layer PReLU outperformed per-neuron PReLU with ResNet-v1-56 (Table 4.9), PANGAEA performed the best in the per-neuron setting (Table 4.8). Indeed, although the per-layer PANGAEA runs still discovered good activation functions, their average performance during search was often lower than that of the per-channel or per-neuron variants (Figure 4.7). These findings suggest that the distribution of per-layer activation functions may be long-tailed: Powerful per-layer activation functions do exist, but they may be more difficult to discover compared to per-channel or per-neuron activation functions.

In order to separate the search space from the search algorithm, in a further experiment 500 per-layer, per-channel, and per-neuron activation functions were randomly created and trained once with ResNet-v1-56 (the functions were first initialized randomly as shown in Figure 4.1, and then mutated randomly three times as shown in Figure 4.2). The best activation functions from these random samples are included in Table 4.8. The best per-layer activation function outperformed ReLU by a large margin, but was not as powerful as those discovered with PANGAEA; the best per-neuron function outperformed all other

variants. These results thus suggest that the distribution of good per-neuron functions may be long-tailed as well, but at a higher level of performance than per-layer and per-channel functions.

In sum, although more research is needed to discover a principled way to select per-layer, per-channel, or per-neuron parameters in a given situation, PANGAEA is flexible enough to discover good functions for all three of these cases.

4.5.3 Efficiency of PANGAEA

PANGAEA's computational efficiency needs to be evaluated from two perspectives. First, how much compute is necessary to find good activation functions? Second, once a good activation function is found, how much more expensive is it to use it in a network compared to a baseline function like ReLU? This section aggregates data from the per-layer, the two per-channel, and the per-neuron PANGAEA runs to demonstrate that PANGAEA is surprisingly efficient in both respects.

First, Figure 4.9 shows how the four PANGAEA runs discovered better activation functions with increasingly more compute. All four runs discovered an activation function that outperformed ReLU relatively early in the search. Because some activation functions are unstable and cause training to fail, they require negligible compute. Computational resources can instead be focused on functions that appear promising. The implications of Figure 4.9 are that in practice, PANGAEA can be used to improve over a baseline activation function relatively cheaply. If better performance is needed, additional compute can be used to continue the search until the desired performance is achieved.

Second, Figure 4.8 shows the distribution of accuracy and compute cost of all activation

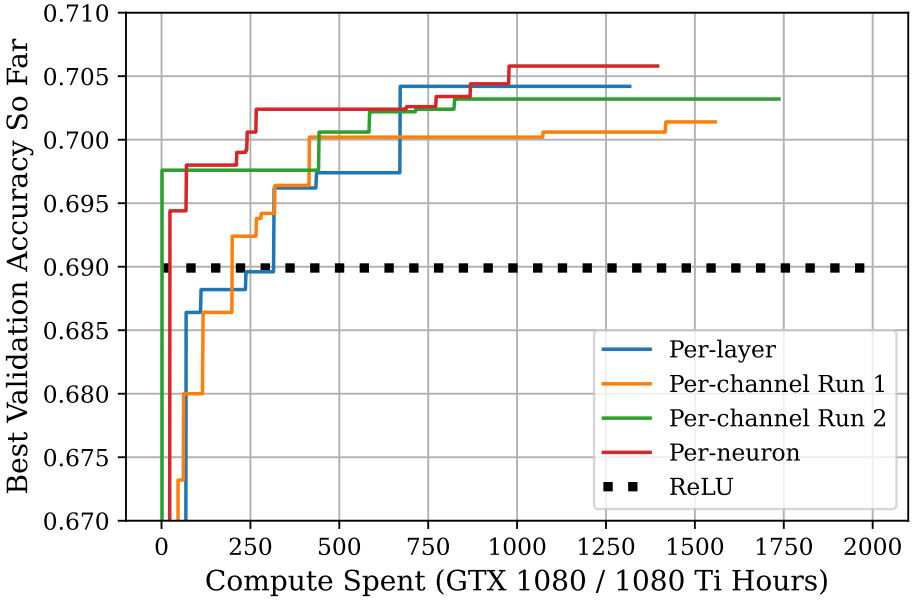


Figure 4.9: Computational efficiency of PANGAEA. The plot shows the performance of the best activation function discovered so far (y-axis) after a given amount of compute was spent (x-axis). All four PANGAEA runs discover activation functions that outperform ReLU with relatively little compute, demonstrating the efficiency of PANGAEA. If even better performance is needed, additional compute can be spent.

functions evaluated in all four PANGAEA runs on ResNet-v1-56. Each point in the scatter plot represents a unique activation function discovered in one of the searches, and the distribution of accuracies and compute costs are shown as histograms above and to the side of the main plot. The results confirm earlier conclusions: All strategies find many activation functions that beat ReLU, and per-neuron PANGAEA discovers more high-performing functions than per-channel or per-layer PANGAEA. The total amount of time it took to train the architecture with the given activation function is shown as "runtime" in the vertical axis. Interestingly, there is a wide range in this metric: some activation functions are significantly more expensive than ReLU, while others are essentially the same. Thus, the figure shows that there exist

plenty of activation functions that significantly beat ReLU but do not incur a significant computational overhead. This distribution also suggests that a multi-objective approach that optimizes for both accuracy and computational cost simultaneously could be effective.

4.6 Additional Results with Learnable Activation Functions

PAU and SPLASH achieved worse-than-expected performance in Table 4.2, so additional experiments were run to investigate their behavior.

PAU Molina et al. [143] utilized a specialized training setup to achieve their results with PAU. In particular, they used a constant learning rate and no weight decay for the PAU layers, but used a learning rate decay of 0.985 per epoch and weight decay of 0.0005 for the other weights. They also used a smaller batch size of 64, and trained for 400 epochs instead of 200. Even though the paper does not mention it, it is possible that such a specialized setup is necessary to achieve good performance with PAU. The experiments in this section utilized this same training setup (but only trained for 200 epochs for fairness) to verify that the PAU implementation was correct.

Unfortunately, some relevant hyperparameters were not included in the PAU paper [143]. These settings include the fixed learning rate used for the PAU layers, whether Nesterov momentum is utilized, and which approximation of Leaky ReLU is used to initialize the PAU weights. This missing information makes it difficult to replicate the original performance exactly. After significant trial-and-error, the following settings worked well: The learning rate for the PAU layers was 0.01, the initial learning rate for other weights was also 0.01, Nesterov momentum was not used, and the PAU weights were initialized to approximate

Table 4.10: CIFAR-100 test accuracy with PAU using a specialized training setup. Performance is comparable to other baseline activation functions, but some runs fail due to training instability.

	WRN-10-4	**ResNet-v2-56**
Accuracy	$62.56_{\pm 4.84}$	$69.59_{\pm 1.66}$
Failed Runs	3 of 10	7 of 10

Leaky ReLU with a slope of 0.01.

Table 4.10 shows the performance of WRN-10-4 and ResNet-v2-56 using these discovered hyperparameters and the specialized training setup from Molina et al. [143]. The performance is comparable to other baseline activation functions. In some cases, the runs failed because of training instability (results were filtered out if the training accuracy was below 0.5). For all hyperparameter combinations tested, PAU was unstable with ResNet-v1-56. Thus, it is possible to get good performance with PAU, but the performance is highly sensitive to the training setup and choice of hyperparameters.

Note that a standard, most commonly used setup was used throughout the main experiments in the chapter for all baseline comparisons. The reason is that there are dozens of such comparisons in this chapter, and it is possible that each one could benefit from a specialized setup—a setup that may not even be fully known at this time. Therefore, a standard setup was necessary to ensure that the comparisons are fair.

SPLASH In addition to ResNet-v1-56 in the main experiments, SPLASH was trained with ResNet-v1-20, ResNet-v1-32, and ResNet-v1-44 for a more thorough characterization of its performance. Each architecture was trained ten times, resulting in training curves shown in

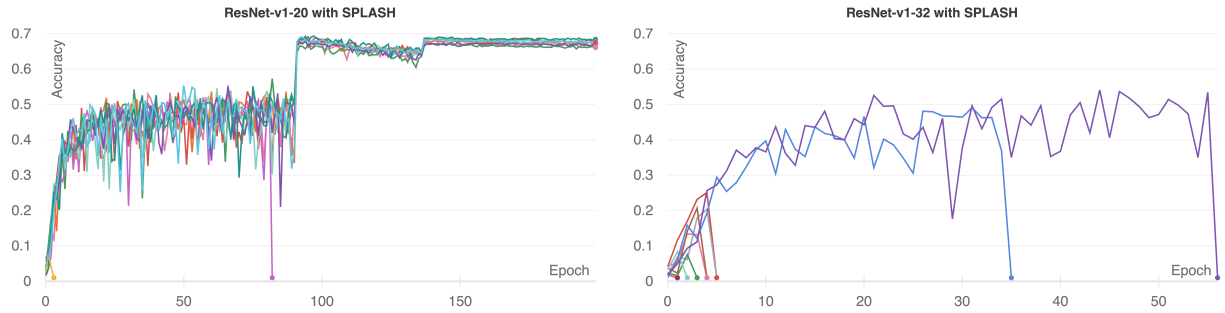


Figure 4.10: ResNet-v1-20 and ResNet-v1-32 test accuracy on CIFAR-100 with the SPLASH activation function. SPLASH units work well in shallow networks, but become unstable with increased depth. This result explains the success of SPLASH in the original work by Tavakoli et al. [184], and also shows why SPLASH units fail with the architectures considered in this work.

Figure 4.10.

With ResNet-v1-20, final test accuracy of the independent runs is between 0.661 and 0.684, which agrees with the results by Tavakoli et al. [184]. However, two of the ten runs failed in the middle of training because the loss became undefined (Figure 4.10), suggesting that SPLASH units can be unstable. Progressing to the deeper ResNet-v1-32, the effect was more pronounced. As shown in Figure 4.10, only two of the ten runs progressed past epoch 30, while no run trained to completion. With ResNet-v1-44 and ResNet-v1-56, training failed within the first epoch, so the training curves are not shown.

These results thus confirm that the implementation is correct, reproducing the results of Tavakoli et al. [184]. However, they also lead to the interesting observation that SPLASH units are effective with shallow networks but struggle with deeper ones, like the ones evaluated in this chapter.

4.7 Baseline Activation Function Details

The following activation functions were used as baseline comparisons in Table 4.2. Some functions were also utilized in the search space (Table 4.1).

- ReLU $= \max\{x, 0\}$ [146].
- ELiSH $= \frac{x}{1+e^{-x}}$ `if` $x \geq 0$ `else` $\frac{e^x-1}{1+e^{-x}}$ [11].
- ELU $= x$ `if` $x \geq 0$ `else` $\alpha(e^x - 1)$, with $\alpha = 1$ [30].
- GELU $= x\Phi(x)$, with $\Phi(x) = P(X \leq x), X \sim \mathcal{N}(0, 1)$, approximated as $0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$ [76].
- HardSigmoid $= \max\{0, \min\{1, 0.2x + 0.5\}\}$.
- Leaky ReLU $= x$ `if` $x \geq 0$ `else` $0.01x$ [127].
- Mish $= x \cdot \tanh(\text{Softplus}(x))$ [142].
- SELU $= \lambda x$ `if` $x \geq 0$ `else` $\lambda\alpha(e^x - 1)$, with $\lambda = 1.05070098$, $\alpha = 1.67326324$ [99].
- sigmoid $= (1 + e^{-x})^{-1}$.
- Softplus $= \log(e^x + 1)$.
- Softsign $= x/(|x| + 1)$.
- Swish $= x \cdot \sigma(x)$, with $\sigma(x) = (1 + e^{-x})^{-1}$ [42, 160].
- tanh $= \frac{e^x-e^{-x}}{e^x+e^{-x}}$.

- PReLU $= x$ `if` $x \geq 0$ `else` αx, where α is a per-neuron learnable parameter initialized to 0.25 [72].

- PSwish $= x \cdot \sigma(\beta x)$, where β is a per-channel learnable parameter [160].

- APL $= \max\{0, x\} + \sum_{s=1}^{S} a_s \max\{0, -x + b_s\}$, where $S = 7$ and a_s and b_s are per-neuron learnable parameters [2].

- PAU $= \frac{\sum_{j=0}^{m} a_j x^j}{1 + |\sum_{k=1}^{n} b_k x^k|}$, where $m = 5, n = 4$, and a_j and b_k are per-layer learnable parameters initialized so that the function approximates Leaky ReLU with a slope of 0.01 [143].

- SPLASH $= \sum_{s=1}^{(S+1)/2} a_s^+ \max\{0, x - b_s\} + a_s^- \max\{0, -x - b_s\}$, where $S = 7, b = [0, 1, 2, 2.5]$, and a_s^+ and a_s^- are per-layer learnable parameters initialized as $a_1^+ = 1$ and all other $a = 0$ [184].

4.8 Scope of PANGAEA Search Space

This section shows that any piecewise real analytic function can be represented as a PANGAEA computation graph containing operators from Table 4.1. In the main experiments, PANGAEA computation graphs were restricted to having at most seven nodes and three learnable parameters α, β, and γ for efficiency. Throughout this section the node and parameter constraints are removed. Parameters take on the role of any real-valued constant, and the set of functions in PANGAEA without node or parameter constraints is denoted as $\mathcal{G}_\infty$. Before proving the main result, the following two lemmas are needed.

Lemma 1. *If $f \in C^\omega$ is a real analytic function, then $f \in \mathcal{G}_\infty$.*

Proof. As f is real analytic, it can be expressed in the form

$$f(x) = \sum_{n=0}^{\infty} a_n (x - x_0)^n, \tag{4.1}$$

with parameters $x_0, a_0, a_1, \ldots \in \mathbb{R}$. As PANGAEA contains the zero, one, addition, and negation operators, the set of integers $\mathbb{Z}$ is contained in $\mathcal{G}_\infty$. This accounts for the exponent n in the expression above. All other operators (addition, subtraction, multiplication, exponentiation) in Equation 4.1 are included in Table 4.1, and so $f \in \mathcal{G}_\infty$. □

Lemma 2. *Given parameters $a, b \in \mathbb{R}$ where $a < b$, the indicator functions*

$$\mathbf{1}_{(-\infty,b)}(x) = \begin{cases} 1 & x < b \\ 0 & x \geq b \end{cases} \tag{4.2}$$

$$\mathbf{1}_{(a,\infty)}(x) = \begin{cases} 1 & x > a \\ 0 & x \leq a \end{cases} \tag{4.3}$$

$$\mathbf{1}_{(a,b)}(x) = \begin{cases} 1 & x \in (a, b) \\ 0 & x \notin (a, b) \end{cases} \tag{4.4}$$

$$\mathbf{1}_{a}(x) = \begin{cases} 1 & x = a \\ 0 & x \neq a \end{cases} \tag{4.5}$$

are in $\mathcal{G}_\infty$.

Proof. Recall that PANGAEA implements the binary division operator x_1/x_2 as `tf.math.divide_no_nan`, which returns 0 if $x_2 = 0$. The indicator function $\mathbf{1}_{(-\infty,b)}(x)$ can then be implemented as

$$\mathbf{1}_{(-\infty,b)}(x) = \frac{\max\{b - x, 0\}}{b - x}. \tag{4.6}$$

There are three cases: if $x < b$, the expression evaluates to one. If $x = b$ or $x > b$, the expression evaluates to zero. By the same reasoning,

$$\mathbf{1}_{(a,\infty)}(x) = \frac{\min\{a - x, 0\}}{a - x}, \tag{4.7}$$

which evaluates to one if $x > a$ and zero otherwise. Finally, note that

$$\mathbf{1}_{(a,b)} = \mathbf{1}_{(-\infty,b)}\mathbf{1}_{(a,\infty)} \tag{4.8}$$

and

$$\mathbf{1}_{a} = (1 - \mathbf{1}_{(-\infty,a)})(1 - \mathbf{1}_{(a,\infty)}). \tag{4.9}$$

All operators in Equations 4.6-4.9 (maximum, minimum, subtraction, multiplication, division, zero, one) are in the PANGAEA search space in Table 4.1, and so the indicator functions are in $\mathcal{G}_\infty$. □

Theorem 1. *If a function f is piecewise real analytic, then $f \in \mathcal{G}_\infty$.*

Proof. If a function f is piecewise real analytic, then it is representable by the form

$$f(x) = \begin{cases} f_0(x) & x \in (-\infty, k_1) \\ K_1 & x = k_1 \\ f_1(x) & x \in (k_1, k_2) \\ K_2 & x = k_2 \\ & \vdots \\ f_{n-1}(x) & x \in (k_{n-1}, k_n) \\ K_n & x = k_n \\ f_n(x) & x \in (k_n, \infty) \end{cases}, \tag{4.10}$$

where parameters $K_1, K_2, \ldots, K_N, k_1, k_2, \ldots, k_n \in \mathbb{R}$ are real-valued, $k_1 < k_2 < \cdots < k_n$ are increasing, and $f_0, f_1, \ldots, f_n \in C^\omega$ are real analytic functions. An equivalent representation of f is the following:

$$\begin{aligned} f(x) = \mathbf{1}_{(-\infty,k_1)}(x)f_0(x) + \mathbf{1}_{k_1}(x)K_1 + \mathbf{1}_{(k_1,k_2)}(x)f_1(x) + \mathbf{1}_{k_2}(x)K_2 + \cdots \\ + \mathbf{1}_{(k_{n-1},k_n)}(x)f_{n-1}(x) + \mathbf{1}_{k_n}(x)K_n + \mathbf{1}_{(k_n,\infty)}(x)f_n(x). \end{aligned} \tag{4.11}$$

By Lemmas 1 and 2, the real analytic functions f_i and the indicator functions $\mathbf{1}_{(\cdot,\cdot)}$ are in $\mathcal{G}_\infty$. Beyond these functions, Equation 4.11 utilizes only addition and multiplication, both of which are operators included in Table 4.1. Therefore, $f \in \mathcal{G}_\infty$. $\square$

4.9 Size of the PANGAEA Search Space

This section analyzes the size of the PANGAEA search space as implemented in the main experiments. Let g represent a general computation graph, and let f represent a specific activation function representable by g. For example, if we have $g(x) = \texttt{binary}(\texttt{unary1}(x), \texttt{unary2}(x))$, then one possible activation function is $f(x) = \tanh(x) + \mathrm{erf}(x)$, and another could be $f(x) = \alpha|x| \cdot \sigma(\beta \cdot x)$.

Let $U = 27$ and $B = 7$ be the number of unary and binary operators in the PANGAEA search space, respectively, and let $E = 3$ be the maximum number of learnable parameters that can be used to augment a given activation function. Given a computation graph g, let u_g and b_g be the number of unary and binary nodes and let e_g be the number of edges in g. For example, with the functional form $g(x) = \texttt{unary1}(\texttt{unary2}(x))$, we have $u_g = 2$, $b_g = 0$, and $e_g = 3$. With $g(x) = \texttt{binary}(\texttt{unary1}(x), \texttt{unary2}(x))$, we have $u_g = 2$, $b_g = 1$, and $e_g = 5$. The quantity e_g includes the edges from the input nodes x and edges to the output node $g(x)$ (see Figure 4.1).

Let $\mathcal{F}_g$ denote the set of all activation functions f that can be represented within the computation graph g. By enumerating the different choices of unary and binary operators, as well as the locations for up to three learnable parameters α, β, and γ, we find the size of

Table 4.11: The number of activation functions representable by a computation graph with a given number of nodes. The PANGAEA search space contains over ten trillion activation functions, and therefore provides a good foundation for finding powerful activation functions with different properties.

	Binary Nodes b_g	Unary Nodes u_g	Edges e_g	Arrangements	Activation Functions
$\mathcal{G}_1$	0	1	2	1	108
$\mathcal{G}_2$	0	2	3	1	5,832
$\mathcal{G}_3$	0	3	4	1	427,923
	1	2	5	1	
$\mathcal{G}_4$	0	4	5	1	31,177,872
	1	3	6	3	
$\mathcal{G}_5$	0	5	6	1	2,210,558,364
	1	4	7	6	
	2	3	8	2	
$\mathcal{G}_6$	0	6	7	1	152,059,087,566
	1	5	8	10	
	2	4	9	10	
$\mathcal{G}_7$	0	7	8	1	10,015,741,690,785
	1	6	9	15	
	2	5	10	30	
	3	4	11	1	

the set to be

$$|\mathcal{F}_g| = U^{u_g} \cdot B^{b_g} \cdot \sum_{i=0}^{E} \binom{e_g}{i}. \tag{4.12}$$

Let $\mathcal{G}_j$ denote the set of computation graphs g containing j nodes. For example,

$$\mathcal{G}_3 = \{g(x) = \texttt{unary1}(\texttt{unary2}(\texttt{unary3}(x))), g(x) = \texttt{binary}(\texttt{unary1}(x), \texttt{unary2}(x))\}. \tag{4.13}$$

Table 4.11 shows the possible combinations of binary nodes b_g, unary nodes u_g, and edges e_g for each set $\mathcal{G}_j$. Additionally, the table shows the number of computation graph arrangements

possible for a given b_g, u_g, and e_g. For example, if $b_g = 2$, $u_g = 3$, and $e_g = 8$, the computation graph could take one of two forms: either

$$g(x) = \texttt{binary1}(\texttt{binary2}(\texttt{unary1}(x), \texttt{unary2}(x)), \texttt{unary3}(x)) \tag{4.14}$$

or

$$g(x) = \texttt{binary1}(\texttt{unary1}(x), \texttt{binary2}(\texttt{unary2}(x), \texttt{unary3}(x))). \tag{4.15}$$

The number of activation functions in PANGAEA is therefore

$$\sum_{j=1}^{7} \sum_{g \in \mathcal{G}_j} |\mathcal{F}_g| = 10{,}170{,}042{,}948{,}450. \tag{4.16}$$

Naturally there exist duplicates within this space. The functions $f(x) = \mathrm{ReLU}(x)$ and $f(x) = \max\{x, 0\}$ have different computation graphs but are functionally identical. Nevertheless, this analysis still provides a useful characterization of the size and diversity of the PANGAEA search space. It is orders of magnitude larger than spaces considered in prior work [11, 17, 160], and yet PANGAEA consistently discovers functions that outperform ReLU and other baseline functions.

4.10 Discussion

It is difficult to select an appropriate activation function for a given architecture because the activation function, network topology, and training setup interact in complex ways. It is especially promising that PANGAEA discovered activation functions that significantly outperformed the baselines, since the architectures and training setups were standard and developed with ReLU. A compelling research direction is to jointly optimize the architecture, training setup, and activation function.

More specifically, there has been significant recent research in automatically discovering the architecture of neural networks through gradient-based, reinforcement learning, or neuroevolutionary methods [43, 162, 197]. In related work, evolution was used discover novel loss functions automatically [61, 62, 115], outperforming the standard cross entropy loss. In the future, it may be possible to optimize many of these aspects of neural network design jointly. Just as new activation functions improve the accuracy of existing network architectures, it is likely that different architectures will be discovered when the activation function is not ReLU. One such example is EfficientNet [183], which achieved state-of-the-art accuracy for ImageNet [41] using the Swish activation function [42, 160]. Coevolution [59, 144, 157, 204] of activation functions, topologies, loss functions, and possibly other aspects of neural network design could allow taking advantage of interactions between them, leading to further improvements in the future.

Similarly, there can be multiple properties of an activation function that make it useful in different scenarios. PANGAEA optimized for activation functions that lead to high accuracy. Another promising area of future work is to search for activation functions that are also efficient to compute (Figure 4.8), improve adversarial robustness [201], stabilize training [99], or meet other objectives like easing optimization or providing implicit regularization [112]. These functions could be discovered with multi-objective optimization, or through other methods like a carefully designed search space or adding regularization to the parameters of the activation functions.

4.11 Conclusion

This chapter introduced PANGAEA, a technique for automatically designing novel, high-performing, parametric activation functions. PANGAEA builds a synergy of two different optimization processes: evolutionary population-based search for the general form, and gradient descent-based fine-tuning of the parameters of the activation function. Compared to previous studies [17, 160], the search space is extended to include deeper and more complex functional forms, including ones unlikely to be discovered by humans. The parameters are adapted during training and are different in different locations of the architecture, thus customizing the functions over both time and space and resulting in improved performance as a result. PANGAEA is able to discover general activation functions that perform well across architectures, and specialized functions taking advantage of a particular architecture, significantly outperforming previously proposed activation functions in both cases. It is thus a promising step towards automatic configuration of neural networks. In order to continue this progress, the next chapter develops an automatic weight initialization algorithm. This algorithm provides a contribution to AutoML on its own, but also complements PANGAEA by making the evaluation of novel activation functions more robust.

Chapter 5

AutoInit: Analytic Signal-Preserving Weight Initialization for Neural Networks

In order to ensure stable signal propagation in neural networks, it is crucial that the weight initialization strategy takes the shape of the activation function into account. While the previous chapters discovered new activation functions that improved the performance of various neural networks, some activation functions evaluated during the searches turned out to be ineffective. Many of these functions could have been effective if the network had been properly initialized. Because current weight initialization algorithms to not automatically adapt to novel activation functions, this chapter presents an approach that does, called AutoInit. AutoInit adapts to novel activation functions and layer types in a neural network. It thus constitutes progress in AutoML in its own right, and compliments the previous chapters by making the evaluation of new activation functions more reliable. The AutoInit package is available at `https://github.com/cognizant-ai-labs/autoinit`.

5.1 Motivation

Proper weight initialization is crucial to achieve high performance with deep networks. A common motif in such networks is repeated layers or building blocks. Thus, if a given layer amplifies or diminishes the forward or backward propagation of signals, repeated

applications of that layer will result in exploding or vanishing signals, respectively [67, 77]. This phenomenon makes optimization difficult, and can even exceed machine precision. The issue persists regardless of whether the weights are sampled to be uniform, normal, or orthogonal [83, 167].

While many initialization strategies have been proposed in the past, these strategies apply only to neural networks with specific activation functions, topologies, or layer types. Thus, researchers designing new models or activation functions have two options. The first option is to derive weight initialization strategies manually for every architecture considered, which is generally difficult and time consuming. The second option is to use existing initialization strategies in new settings, where they may be incorrect and therefore misleading: A candidate model may appear poor when it is the suboptimal initialization that makes training difficult.

To overcome this problem, this chapter proposes AutoInit, an algorithm that automatically calculates analytic mean- and variance-preserving weight initialization for neural networks. Since AutoInit is algorithmic, it relieves the researcher from a difficult but consequential step in model design. It is no longer necessary to use existing weight initialization strategies in incorrect settings: AutoInit provides an appropriate default initialization automatically, resulting in better and more reliable performance.

5.2 Neural Network Signal Propagation

AutoInit aims to stabilize signal propagation throughout an entire neural network. More precisely, consider a layer that shifts its input by α and scales the input by a factor of β. Given an input signal with mean μ_{in} and variance ν_{in}, after applying the layer, the output

signal will have mean $\mu_{\text{out}} = \alpha + \beta\mu_{\text{in}}$ and variance $\nu_{\text{out}} = \beta^2\nu_{\text{in}}$. In a deep network in which the layer is applied L times the effect is compounded and the signal at the final layer has mean and variance

$$\mu_{\text{out}} = \beta^L\mu_{\text{in}} + \alpha(\beta^L + \beta^{L-1} + \cdots + \beta + 1), \quad \nu_{\text{out}} = \beta^{2L}\nu_{\text{in}}. \tag{5.1}$$

If $|\beta| > 1$, the network will suffer from a mean shift and exploding signals as it increases in depth:

$$\lim_{L\to\infty} \mu_{\text{out}} = \infty, \quad \lim_{L\to\infty} \nu_{\text{out}} = \infty. \tag{5.2}$$

In the case that $|\beta| < 1$, the network will suffer from a mean shift and vanishing signals:

$$\lim_{L\to\infty} \mu_{\text{out}} = \alpha/(1-\beta), \quad \lim_{L\to\infty} \nu_{\text{out}} = 0. \tag{5.3}$$

AutoInit calculates analytic mean- and variance-preserving weight initialization so that $\alpha = 0$ and $\beta = 1$, thus avoiding the issues of mean shift and exploding/vanishing signals.

5.3 The AutoInit Framework

AutoInit is a general framework that adapts to different layer types. Its implementation is outlined in Algorithm 1. A given `layer` in a neural network receives as its input a tensor x with mean μ_{in} and variance ν_{in}. After applying the `layer`, the output tensor has mean $\mu_{\text{out}} = \mathrm{E}(\texttt{layer}(x))$ and variance $\nu_{\text{out}} = \mathrm{Var}(\texttt{layer}(x))$. The function $g_{\texttt{layer}}$ maps input mean and variance to output mean and variance when the `layer` is applied:

$$g_{\texttt{layer}} : (\mu_{\text{in}}, \nu_{\text{in}}) \mapsto (\mu_{\text{out}}, \nu_{\text{out}}). \tag{5.4}$$

Note that g in Equation 5.4 depends on the type of `layer`; e.g. $g_{\texttt{Dropout}}$ and $g_{\texttt{ReLU}}$ are different functions. For layers with trainable weights, the mean and variance mapping will depend

Algorithm 1: AutoInit

```
Input: Network with layers L, directed edges E
output_layers = {l ∈ L | (l, l') ∉ E ∀ l' ∈ L}
for output_layer in output_layers do
    initialize(output_layer)

def initialize(layer):
    layers_in = {l ∈ L | (l, layer) ∈ E}
    i = 1
    for layer_in in layers_in do
        μ_in_i, ν_in_i = initialize(layer_in)
        i = i + 1
    μ_in = (μ_in_1, μ_in_2, ..., μ_in_N)
    ν_in = (ν_in_1, ν_in_2, ..., ν_in_N)
    if layer has weights θ then
        initialize θ s.t. g_layer,θ(μ_in, ν_in) = (0, 1)
        μ_out, ν_out = 0, 1
    else
        μ_out, ν_out = g_layer(μ_in, ν_in)
    return μ_out, ν_out
```

on those weights. For example, the function $g_{\texttt{Conv2D},\theta}$ maps input mean and variance to output mean and variance after the application of a `Conv2D` layer parameterized by weights θ. Deriving g for all layers makes it possible to model signal propagation across an entire neural network. Thus, if μ_{in} and ν_{in} are known, it is natural to calculate initial weights θ such that the layer output will have zero mean and unit variance. For example, for `Conv2D` layers, one possibility is

$$\theta \sim \mathcal{N}\left(0, 1/\sqrt{\texttt{fan_in}(\nu_{\text{in}} + \mu_{\text{in}}^2)}\right) \implies g_{\texttt{Conv2D},\theta}(\mu_{\text{in}}, \nu_{\text{in}}) = (0, 1). \tag{5.5}$$

The AutoInit framework includes mean and variance mapping functions g for the majority of layers used in modern architectures. Section 5.4 details how these functions and

the corresponding initialization strategies (e.g. Equation 5.5) are derived. New layers can be included by deriving g manually, or by approximating it through Monte Carlo simulation. This approach ensures that reliable estimates for μ_{in} and ν_{in} are available at all layers in a network, which in turn allows for weight initialization that stabilizes the signals to have zero mean and unit variance, avoiding the issues of mean shift and exploding/vanishing signals (Equations 5.2 and 5.3).

The main advantage of AutoInit is that it is a general method. Unlike prior work, which imposes design constraints, AutoInit adapts to different settings automatically in order to improve performance. Sections 5.5 through 5.11 demonstrate this adaptability experimentally from several perspectives: different classes of models (convolutional, residual, transformer), hyperparameter settings (activation function, dropout rate, weight decay, learning rate, optimizer), model depths (nine layer CNN to 812 layer ResNet), image sizes (ten-class 28×28 grayscale to 1,000-class 160×160 RGB), and data modalities (vision, language, tabular, multi-task, transfer learning). AutoInit also outperforms data-dependent initialization methods and stabilizes convolutional, residual, and transformer networks without normalization layers. This generality is shown to be particularly useful in neural architecture search and activation function discovery, where thousands of new designs need to be evaluated robustly. AutoInit produces specialized weight initialization strategies for each candidate, which allows for measuring their performance more accurately. As a result, better solutions are discovered. The experiments thus show that AutoInit is an effective initialization algorithm for existing networks as well as a good starting point for networks that may be developed in the future.

5.4 Mean and Variance Estimation for Different Layer Types

In the AutoInit framework of Algorithm 1, the mean and variance mapping function g needs to be defined for each type of layer in a given neural network. This section presents derivations for a majority of the most commonly used layers available in the TensorFlow package [1] at time of writing. Extending AutoInit to support new layers in the future will require deriving the function g for those layers. Monte Carlo sampling can also be used as an approximation for g before it is manually derived.

In the following paragraphs, x denotes the input to a layer, and y is the output. The incoming and outgoing means and variances are denoted as $\mu_{\text{in}} \coloneqq \mathrm{E}(x)$, $\mu_{\text{out}} \coloneqq \mathrm{E}(y)$, $\nu_{\text{in}} \coloneqq \mathrm{Var}(x)$, and $\nu_{\text{out}} \coloneqq \mathrm{Var}(y)$. The notation `Conv{1D,2D,3D}` is used to refer to `Conv1D`, `Conv2D`, and `Conv3D`, and analogously for other layer types. Inputs to each layer are assumed to be independent and normally distributed. Although these assumptions may not always hold exactly, experiments show that AutoInit models signal propagation across different types of networks well in practice.

Convolution and Dense Layers The analysis in the next paragraph applies to `Conv{1D, 2D,}`, `DepthwiseConv{1D,2D}`, and `Dense` layers, since convolution layers are dense layers with sparse connectivity. Notation and derivation are inspired by that of Glorot and Bengio [55] and He et al. [72].

A feedforward layer can be written as $y = Wx + b$, where x is the input, W is a `fan_out` $\times$ `fan_in` weight matrix, b is a vector of biases, and y is the result. Assume the elements of W are mutually independent and from the same distribution, and likewise for the elements of x. Further assume that W and x are independent of each other. The

outgoing mean can then be written as $\mu_{\text{out}} = \mathrm{E}(W)\mu_{\text{in}}$. For the outgoing variance, letting W have zero mean and expanding the product of independent random variables yields $\nu_{\text{out}} = \texttt{fan_in}\mathrm{Var}(W)(\nu_{\text{in}} + \mu_{\text{in}}^2)$. Sampling the weights W according to

$$W \sim \mathcal{N}\left(0, \frac{1}{\sqrt{\texttt{fan_in}(\nu_{\text{in}} + \mu_{\text{in}}^2)}}\right) \tag{5.6}$$

or

$$W \sim \mathcal{U}\left(-\frac{\sqrt{3}}{\sqrt{\texttt{fan_in}(\nu_{\text{in}} + \mu_{\text{in}}^2)}}, \frac{\sqrt{3}}{\sqrt{\texttt{fan_in}(\nu_{\text{in}} + \mu_{\text{in}}^2)}}\right) \tag{5.7}$$

is sufficient to ensure that

$$\mu_{\text{out}} = 0 \text{ and } \nu_{\text{out}} = 1. \tag{5.8}$$

Activation Functions The analysis in the next paragraph accounts for all activation functions in TensorFlow, including `elu`, `exponential`, `gelu`, `hard_sigmoid`, `LeakyReLU`, `linear`, `PReLU`, `ReLU`, `selu`, `sigmoid`, `softplus`, `softsign`, `swish`, `tanh`, and `ThresholdedReLU` [by 30, 31, 42, 72, 76, 99, 127, 146, 160, respectively], and in fact extends to any integrable `Activation` function f.

Let $p_{\mathcal{N}}(x; \mu, \sigma)$ denote the probability density function of a Gaussian distribution with mean μ and standard deviation σ. By the law of the unconscious statistician,

$$\mu_{\text{out}} = \int_{-\infty}^{\infty} f(x) p_{\mathcal{N}}(x; \mu_{\text{in}}, \sqrt{\nu_{\text{in}}})\, \mathrm{d}x, \tag{5.9}$$

$$\nu_{\text{out}} = \int_{-\infty}^{\infty} f(x)^2 p_{\mathcal{N}}(x; \mu_{\text{in}}, \sqrt{\nu_{\text{in}}})\, \mathrm{d}x - \mu_{\text{out}}^2. \tag{5.10}$$

These integrals are computed for an arbitrary activation function f with adaptive quadrature, a well-established numerical integration approach that approximates integrals using adaptively refined subintervals [70, 156, 190].

Dropout Layers Dropout layers randomly set `rate` percentage of their inputs to zero [175]. Therefore,

$$\mu_{\text{out}} = \mu_{\text{in}}(1 - \texttt{rate}) \text{ and } \nu_{\text{out}} = \nu_{\text{in}}(1 - \texttt{rate}). \tag{5.11}$$

However, this analysis only applies to `SpatialDropout{1D,2D,3D}` layers. For regular `Dropout` layers, TensorFlow automatically scales the values by $1/(1 - \texttt{rate})$ to avoid a mean shift towards zero.[1] Adjusting for this change gives

$$\mu_{\text{out}} = \mu_{\text{in}} \text{ and } \nu_{\text{out}} = \nu_{\text{in}}/(1 - \texttt{rate}). \tag{5.12}$$

Pooling Layers The same approach applies to all commonly used pooling layers, including `AveragePooling{1D,2D,3D}`, `MaxPooling{1D,2D,3D}`, `GlobalAveragePooling{1D,2D,3D}`, and `GlobalMaxPooling{1D,2D,3D}`.

Let $op(\cdot)$ be the average operation for an average pooling layer, and the maximum operation for a max pooling layer. Define K to be the pool size of the layer. For standard 1D, 2D, and 3D pooling layers, K would equal k, $k \times k$, and $k \times k \times k$, respectively. The global pooling layers can be seen as special cases of the standard pooling layers where the pool size is the same size as the input tensor, except along the batch and channel dimensions. Analytically, the outgoing mean and variance can be expressed as

$$\mu_{\text{out}} = \int \cdots \int_{\mathbb{R}^K} op(x_1, x_2, \ldots, x_K) \cdot \prod_{i=1}^{K} p_{\mathcal{N}}(x_i; \mu_{\text{in}}, \sqrt{\nu_{\text{in}}}) \, \mathrm{d}x_1 \, \mathrm{d}x_2 \cdots \mathrm{d}x_K, \tag{5.13}$$

$$\nu_{\text{out}} = \int \cdots \int_{\mathbb{R}^K} op(x_1, x_2, \ldots, x_K)^2 \cdot \prod_{i=1}^{K} p_{\mathcal{N}}(x_i; \mu_{\text{in}}, \sqrt{\nu_{\text{in}}}) \, \mathrm{d}x_1 \, \mathrm{d}x_2 \cdots \mathrm{d}x_K - \mu_{\text{out}}^2, \tag{5.14}$$

[1] `https://github.com/tensorflow/tensorflow/blob/v2.5.0/tensorflow/python/keras/layers/core.py#L149-L150`

where the x_i represent tensor entries within a pooling window. Unfortunately, even a modest 3×3 pooling layer requires computing nine nested integrals, which is prohibitively expensive. In this case, a Monte Carlo simulation is appropriate. Sample $x_{1_j}, x_{2_j}, \ldots x_{K_j}$ from $\mathcal{N}(\mu_{\text{in}}, \sqrt{\nu_{\text{in}}})$ for $j = 1, \ldots, S$ and return

$$\mu_{\text{out}} = \frac{1}{S} \sum_{j=1}^{S} op(x_{1_j}, x_{2_j}, \ldots, x_{K_j}), \tag{5.15}$$

$$\nu_{\text{out}} = \frac{1}{S} \sum_{j=1}^{S} op(x_{1_j}, x_{2_j}, \ldots, x_{K_j})^2 - \mu_{\text{out}}. \tag{5.16}$$

Normalization Layers `BatchNormalization`, `LayerNormalization`, and `GroupNormalization` normalize the input to have mean zero and variance one [8, 88, 198]. Thus,

$$\mu_{\text{out}} = 0 \text{ and } \nu_{\text{out}} = 1. \tag{5.17}$$

Arithmetic Operators Assume the input tensors $x_1, x_2, \ldots, x_N$ with means $\mu_{\text{in}_1}, \mu_{\text{in}_2}, \ldots, \mu_{\text{in}_N}$ and variances $\nu_{\text{in}_1}, \nu_{\text{in}_2}, \ldots, \nu_{\text{in}_N}$ are independent. The following mean and variance mapping functions are derived. For an `Add` layer,

$$\mu_{\text{out}} = \sum_{i=1}^{N} \mu_{\text{in}_i} \text{ and } \nu_{\text{out}} = \sum_{i=1}^{N} \nu_{\text{in}_i}. \tag{5.18}$$

For an `Average` layer,

$$\mu_{\text{out}} = \frac{1}{N} \sum_{i=1}^{N} \mu_{\text{in}_i} \text{ and } \nu_{\text{out}} = \frac{1}{N^2} \sum_{i=1}^{N} \nu_{\text{in}_i}. \tag{5.19}$$

For a `Subtract` layer,

$$\mu_{\text{out}} = \mu_{\text{in}_1} - \mu_{\text{in}_2} \text{ and } \nu_{\text{out}} = \nu_{\text{in}_1} + \nu_{\text{in}_2}. \tag{5.20}$$

Finally, for a `Multiply` layer,

$$\mu_{\text{out}} = \prod_{i=1}^{N} \mu_{\text{in}_i} \text{ and } \nu_{\text{out}} = \prod_{i=1}^{N} (\nu_{\text{in}_i} + \mu_{\text{in}_i}^2) - \prod_{i=1}^{N} \mu_{\text{in}_i}^2. \tag{5.21}$$

Concatenation Layers Assume the inputs $x_1, x_2, \ldots, x_N$ with means $\mu_{\text{in}_1}, \mu_{\text{in}_2}, \ldots, \mu_{\text{in}_N}$ and variances $\nu_{\text{in}_1}, \nu_{\text{in}_2}, \ldots, \nu_{\text{in}_N}$ are independent, and let input x_i have C_i elements. Then for a `Concatenate` layer,

$$\mu_{\text{out}} = \frac{1}{\sum C_i} \sum_{i=1}^{N} C_i \mu_{\text{in}_i}, \tag{5.22}$$

$$\nu_{\text{out}} = \frac{1}{\sum C_i} \sum_{i=1}^{N} C_i (\nu_{\text{in}_i} + \mu_{\text{in}_i}^2) - \mu_{\text{out}}^2. \tag{5.23}$$

Recurrent Layers A Monte Carlo simulation can be used to estimate the outgoing mean and variance for recurrent layers, including `GRU`, `LSTM`, and `SimpleRNN` [29, 78]. Recurrent layers often make use of activation functions like sigmoid and tanh that constrain the scale of the hidden states. Because of this practice, recurrent layers should be initialized with a default scheme or according to recent research in recurrent initialization [26, 54]. AutoInit will then estimate the outgoing mean and variance in order to inform appropriate weight scaling elsewhere in the network.

Padding Layers `ZeroPadding{1D,2D,3D}` layers augment the borders of the input tensor with zeros, increasing its size. Let z be the proportion of elements in the tensor that are padded zeros. Then $z = (\texttt{padded_size} - \texttt{original_size})/\texttt{padded_size}$, and

$$\mu_{\text{out}} = \mu_{\text{in}}(1 - z) \text{ and } \nu_{\text{out}} = \nu_{\text{in}}(1 - z). \tag{5.24}$$

Shape Adjustment Layers Many layers alter the size or shape of the input tensor but do not change the distribution of the data. These layers include `Flatten`, `Permute`, `Reshape`, `UpSampling{1D,2D,3D}`, and `Cropping{1D,2D,3D}` layers. The same is true of TensorFlow API calls `tf.reshape`, `tf.split`, and `tf.transpose`. For these layers,

$$\mu_{\text{out}} = \mu_{\text{in}} \text{ and } \nu_{\text{out}} = \nu_{\text{in}}. \tag{5.25}$$

Input Layers An `InputLayer` simply exposes the model to the data, therefore

$$\mu_{\text{out}} = \mu_{\text{data}} \text{ and } \nu_{\text{out}} = \nu_{\text{data}}. \tag{5.26}$$

TensorFlow allows nesting models within other models. In this use case where the `InputLayer` does not directly connect to the training data,

$$\mu_{\text{out}} = \mu_{\text{in}} \text{ and } \nu_{\text{out}} = \nu_{\text{in}}. \tag{5.27}$$

Matrix Multiplication A call to `tf.matmul` takes input tensors x_1 and x_2 of shape $\cdots \times m \times n$ and $\cdots \times n \times p$ and produces the output tensor x_{out} of shape $\cdots \times m \times p$ with entries

$$x_{\text{out}_{\cdots,i,j}} = \sum_{k=1}^{n} x_{1_{\cdots,i,k}} x_{2_{\cdots,k,j}}. \tag{5.28}$$

Assuming independent matrix entries, the output statistics can then be calculated as

$$\mu_{\text{out}} = n\mu_{\text{in}_1}\mu_{\text{in}_2}, \tag{5.29}$$

$$\nu_{\text{out}} = n\left((\nu_{\text{in}_1} + \mu_{\text{in}_1}^2)(\nu_{\text{in}_2} + \mu_{\text{in}_2}^2) - \mu_{\text{in}_1}^2\mu_{\text{in}_2}^2\right). \tag{5.30}$$

Reduction Operators A call to `tf.reduce_mean` reduces the size of the input tensor by averaging values along one or more axes. For example, averaging an input tensor of shape $128 \times 8 \times 8 \times 256$ along axes 1 and 2 would produce an output tensor of shape $128 \times 1 \times 1 \times 256$. Let D represent the product of the length of the axes being averaged over (in the example above, $8 \times 8 = 64$). The output tensor has

$$\mu_{\text{out}} = \mu_{\text{in}} \text{ and } \nu_{\text{out}} = \nu_{\text{in}}/D. \tag{5.31}$$

The function `tf.reduce_sum` performs similarly, summing entries instead of averaging them. In this case,

$$\mu_{\text{out}} = D\mu_{\text{in}} \text{ and } \nu_{\text{out}} = D\nu_{\text{in}}. \tag{5.32}$$

Maintaining Variance $\neq 1$. In Algorithm 1, AutoInit initializes weights so that the output signal at each layer has mean zero and variance one. Although signal variance $\nu = 1$ is sufficient for stable signal propagation, it is not a necessary condition. Indeed, other values for the signal variance ν could be utilized, as long as ν remains consistent throughout the network. If a different ν is desired, weights can be initialized according to Equation 5.6 or 5.7 and then multiplied by $\sqrt{\nu}$. For instance, such a modification was done for the CoAtNet model in Section 5.7, resulting in slightly improved final performance.

5.5 Hyperparameter Variation in CNNs

Experiment Setup The first experiment tests AutoInit's performance across a range of hyperparameter values for CNNs. The experiment focuses on the All-CNN-C architecture [174], which consists of convolutional layers, ReLU activation functions, dropout layers, and

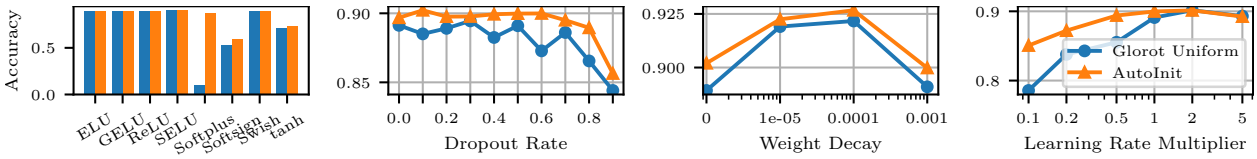


Figure 5.1: All-CNN-C test accuracy on CIFAR-10. AutoInit results in comparable or better performance with different activation functions, better performance across all dropout rates and weight decay settings, and is less sensitive to the choice of learning rate than the default initialization.

a global average pooling layer at the end of the network. This simple design helps identify performance gains that can be attributed to proper weight initialization. The network is trained on the CIFAR-10 dataset [104] using the standard setup (Appendix A.3). In particular, the baseline comparison is the "Glorot Uniform" strategy [also called Xavier initialization; 55], where weights are sampled from $\mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}, \frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}\right)$.

Hyperparameter Variation In separate experiments, the activation function, dropout rate, weight decay, and learning rate multiplier were changed. While one hyperparameter was varied, the others were fixed to the default values.

Results Figure 5.1 shows the performance of the network with the default initialization and with AutoInit in these different settings. In sum, AutoInit improved performance in every hyperparameter variation evaluated. As Figure 5.2 shows, AutoInit is adaptive. It alters the initialization to account for different activation functions and dropout rates automatically.

AutoInit is also robust. Even as other hyperparameters like learning rate and weight decay change, AutoInit still results in a higher performing network than the default initialization. The results thus suggest that AutoInit provides an improved default initialization for

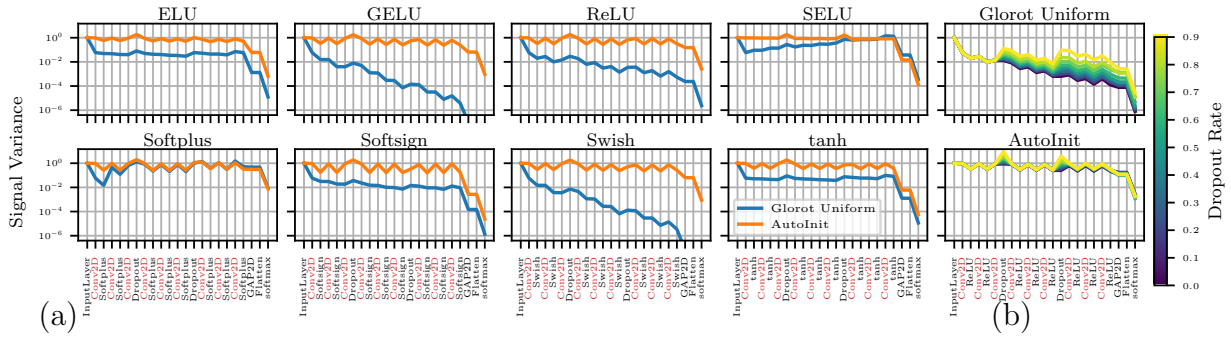


Figure 5.2: Signal propagation in All-CNN-C networks with different (a) activation functions and (b) dropout rates. With the default initialization, signals often vanish with depth, and their behavior is inconsistent across activation functions and dropout rates. With AutoInit, the variance fluctuates naturally as each layer modifies its input. At layers with weights (marked in red), AutoInit scales the weights appropriately to return the variance to approximately 1.0, stabilizing training in each case.

convolutional neural networks.

5.6 Stability in Deep ResNets

This section expands the experimental analysis of AutoInit to residual networks, focusing on preactivation residual networks of various depths [74]. The training setup is standard unless explicitly stated otherwise (Appendix A.3). In particular, the initialization is "He Normal" [72], where weights are sampled from $\mathcal{N}(0, \sqrt{2/\texttt{fan_in}})$.

Visualizing Signal Propagation Figure 5.3 shows how the signal variance changes with depth. With ResNet-56, the variance increases where the shortcut connection and residual branch meet, and the variance drops whenever ReLU is applied. Although the variance increases exponentially with the default initialization and linearly with AutoInit (note the

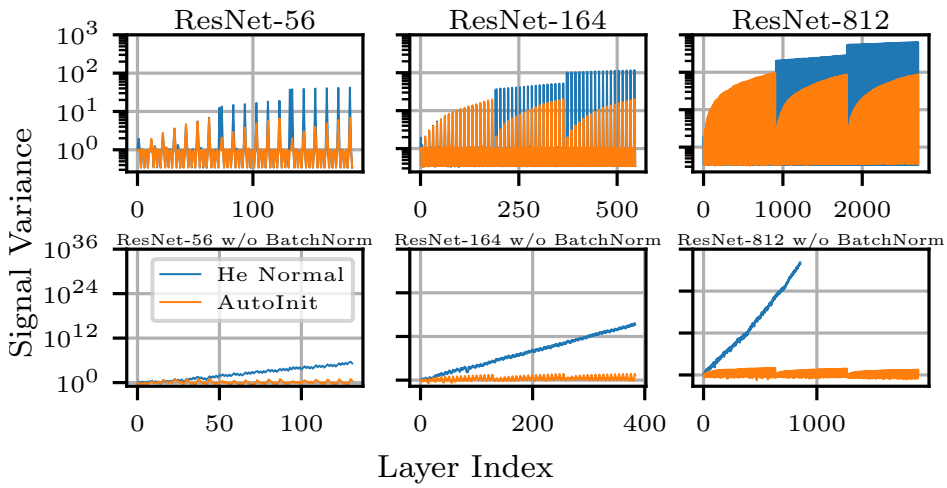


Figure 5.3: Signal propagation in residual networks. Gaussian input was fed to the networks and empirical variance computed at each layer. Since `ReLU`, `BatchNormalization`, and `Add` are counted as individual layers in this diagram, the total number of layers is different from that in the architecture name (i.e. ResNet-164 has 164 convolutional layers but over 500 total layers). The default initialization causes exploding signals, while AutoInit ensures signal propagation is stable.

log scale on the y axis), training is still stable because batch normalization layers return the signal to variance 1.0. Without batch normalization, the signal variance never stabilizes under the default initialization. In contrast, removing batch normalization is not an issue with AutoInit; the signal variance remains stable with depth.

With the deeper ResNet-164 and ResNet-812 networks, the conclusions are similar but more pronounced. In the case of ResNet-812 without batch normalization, the signals explode so severely that they exceed machine precision. AutoInit avoids this issue entirely.

Stable Initial Learning Rates Exploding or vanishing signals make optimization difficult because they result in gradient updates that are too large to be accurate or too small to be

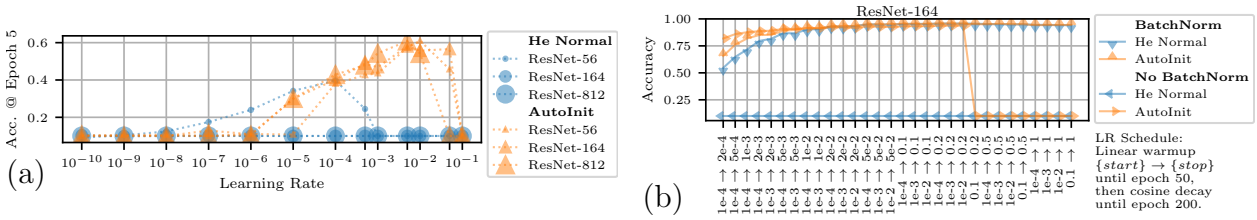


Figure 5.4: ResNet accuracy on CIFAR-10 with different settings. (a) Accuracy of unnormalized ResNet architectures after five epochs of training with different learning rates and weight initializations. While default initialization makes training difficult in ResNet-56 and impossible at greater depths, AutoInit results in consistent training at all depths. (b) Accuracy of ResNet-164 with a variety of learning rate schedules and initializations. AutoInit is comparable to or outperforms the default initialization in every case.

meaningful. This phenomenon can be observed when the network does not exceed chance accuracy. Therefore, a simple way to quantify whether a weight initialization is effective is to observe a network's performance after a few epochs.

Using this metric, AutoInit was compared against the default initialization by training unnormalized versions of ResNet-56, ResNet-164, and ResNet-812 for five epochs with a variety of learning rates. With the default initialization, ResNet-56 requires a learning rate between 10^{-8} and 0.5×10^{-3} to begin training, but training was not possible with ResNet-164 or ResNet-812 because of exploding signals (Figure 5.4a). AutoInit stabilizes training for all three networks, and its effect does not diminish with depth. The networks remain stable with higher learning rates between 10^{-4} and 0.05. Such rates speed up learning, and also correlate with better generalization performance [91, 171, 172].

Full ResNet Training In the third residual network experiment, ResNet-164 was trained to completion on CIFAR-10 with different learning rate schedules. All schedules included a linear warm-up phase followed by a decay to zero using cosine annealing [123].

Figure 5.4b displays the performance with a variety of such schedules. When the best learning rate schedules are used, ResNet-164 achieves comparable performance with the default initialization and with AutoInit. However, when a suboptimal schedule is used, performance degrades more quickly with the default initialization than it does with AutoInit. Without batch normalization, the network requires proper weight initialization for stability. In this case, ResNet-164 with the default initialization fails to train regardless of the learning rate schedule, whereas AutoInit results in high accuracy for the majority of them.

Together, the experiments in this section show that AutoInit is effective with deep networks. It prevents signals from exploding or vanishing, makes it possible to use larger learning rates, and achieves high accuracy, with and without batch normalization.

5.7 High-Resolution Images with Transformers

This section extends AutoInit to transformer architectures and applies them to high-resolution image classification. Specifically, AutoInit is applied to CoAtNet, a model that combines convolutional and attention layers [36]. The model is trained on Imagenette, a subset of 10 classes from the ImageNet dataset [41, 82]. Imagenette allows evaluating AutoInit in a high-resolution image classification task with a 132× smaller carbon footprint than the full ImageNet dataset would (Appendix B). As Table 5.1 shows, AutoInit outperforms six commonly used initialization schemes as well as the default initialization, which initializes convolutional layers from $\mathcal{N}(0, \sqrt{2/\texttt{fan_out}})$ and fully-connected layers from $\mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}, \frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}\right)$. Furthermore, AutoInit stabilizes the network even when normalization layers are removed, suggesting that it is a promising candidate towards developing normalizer-free transformer architectures. Full details are in Appendix A.3.

Table 5.1: CoAtNet top-1 accuracy on Imagenette, shown as median of three runs. The first four experiments vary the activation function, while the fifth removes all normalization layers from the architecture. A "-" indicates that training diverged. AutoInit produces the best model in three of the five settings, and remains stable even without normalization layers.

CoAtNet	w/ GELU	w/ ReLU	w/ SELU	w/ Swish	w/o Norm
Default Init.	89.38	89.22	86.09	88.69	-
Glorot Normal	91.44	91.54	87.59	90.42	**85.89**
Glorot Uniform	91.16	91.18	**88.25**	90.06	85.73
He Normal	88.48	88.05	86.11	88.36	-
He Uniform	88.66	87.87	86.37	88.41	-
LeCun Normal	91.11	90.57	87.80	90.83	-
LeCun Uniform	90.55	90.65	87.67	90.57	-
AutoInit	**92.48**	**92.15**	86.80	**92.28**	85.73

Table 5.2: ResNet-50 top-1 and top-5 validation accuracy on ImageNet. AutoInit improves performance, even with large and challenging datasets.

	top-1	top-5
Default Init.	74.33	91.60
AutoInit	**75.35**	**92.03**

5.8 Scaling up to ImageNet

In order to compliment the results from Section 5.7 and demonstrate that AutoInit can scale to more difficult tasks, ResNet-50 was trained from scratch on ImageNet with the default initialization and with AutoInit. As Table 5.2 shows, AutoInit improves top-1 and top-5 accuracy in this task as well. Full training details are in Appendix A.3.

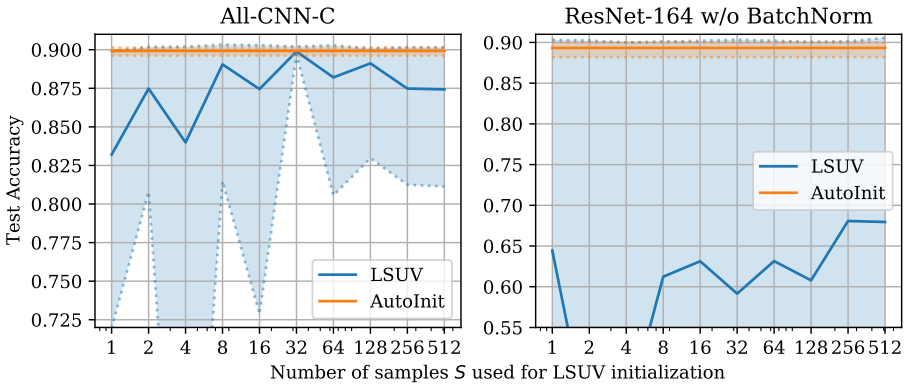


Figure 5.5: Mean CIFAR-10 test accuracy for AutoInit vs. LSUV with different numbers of samples S. Each evaluation is repeated 10 times; the shaded area shows the maximum and minimum accuracy among all trials. AutoInit is consistent, but LSUV struggles when S is small or the network is deep.

5.9 Contrast with Data-Dependent Initialization

The layer-sequential unit-variance (LSUV) algorithm is the most natural data-dependent initialization comparison to AutoInit because both approaches aim to scale the weights appropriately in an architecture-agnostic way. LSUV pre-initializes the weights with an existing approach, feeds S training samples through the network, and adjusts the scale of the weights so that each layer's output variance is approximately one [140].

Data-dependent initialization is time-consuming for large S (indeed, even $S = 1$ is used in practice [98]). However, if S is too small, the samples may not reflect the statistics of the dataset accurately, leading to poor initialization. Figure 5.5 demonstrates this phenomenon. In some training runs LSUV matches the performance of AutoInit, but in many instances the randomly selected samples do not accurately reflect the overall dataset and performance suffers. Since AutoInit is not data-dependent, it does not have this issue. Details of this

experiment are in Appendix A.3.

5.10 Enabling Neural Architecture Search

Sections 5.5 through 5.9 demonstrated that AutoInit works well for convolutional, residual, and transformer networks with a variety of hyperparameter values and depths. In this section, the results are extended to a broader variety of network topologies and types of tasks, for two reasons. First, whereas custom weight initialization may be developed by hand for the most popular machine learning benchmarks, it is unlikely to happen for a variety of architectures and tasks beyond them. Second, as new types of neural network designs are developed in the future, it will be important to initialize them properly to reduce uncertainty in their performance. This section evaluates the generality of AutoInit by applying it to the variety of networks generated in a neural architecture search process with five types of tasks.

The CoDeepNEAT Architecture Search Method Neural networks are evolved using CoDeepNEAT [114, 138]. CoDeepNEAT extends previous work on evolving network topologies and weights [144, 176] to the level of evolving deep learning architectures. Utilizing a cooperative coevolution framework [157], CoDeepNEAT evolves populations of modules and blueprints simultaneously (Figure 5.6a). Modules are small neural networks, complete with layers, connections, and hyperparameters. Blueprints are computation graphs containing only nodes and directed edges. To create a candidate neural network, CoDeepNEAT chooses a blueprint and replaces its nodes with selected modules. This mechanism makes it possible to evolve deep, complex, and recurrent structures, while taking advantage of the modularity often found in state-of-the-art models. In addition to the network structure, CoDeepNEAT

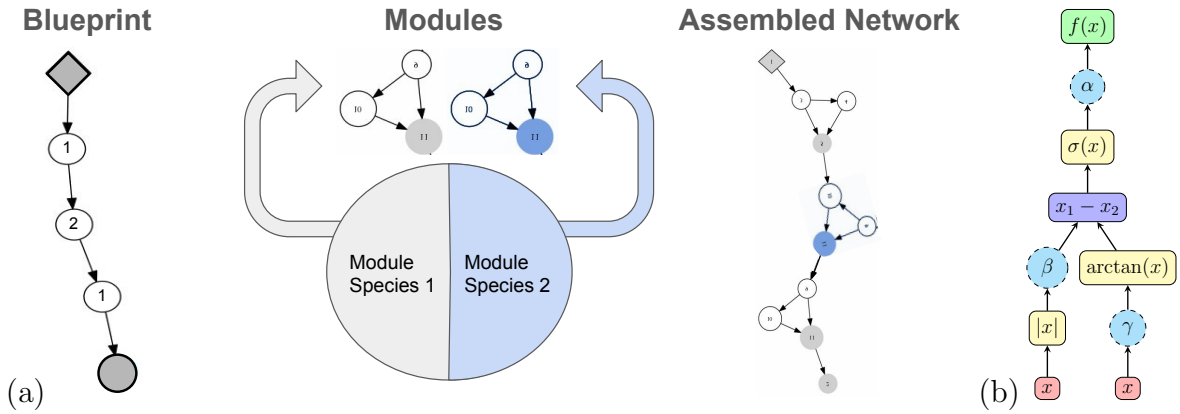


Figure 5.6: (a) The CoDeepNEAT method. Modules replace nodes in the blueprint to create a candidate neural network. (b) An example activation function created with the PANGAEA method. The computation graph represents the parametric function $f(x) = \alpha \cdot \sigma(\beta \cdot |x| - \arctan(\gamma \cdot x))$. CoDeepNEAT and PANGAEA generate a variety of architectures and activation functions that can be used to evaluate AutoInit's generality and flexibility.

evolves hyperparameters like dropout rate, kernel regularization, and learning rate. The network weights are not evolved, but instead trained with gradient descent. The generality of CoDeepNEAT helps minimize human design biases and makes it well-suited to analyzing AutoInit's performance in a variety of open-ended machine learning settings.

Tasks Using CoDeepNEAT, networks are evolved for their performance in vision (MNIST), language (Wikipedia Toxicity), tabular (PMLB Adult), multi-task (Omniglot), and transfer learning (Oxford 102 Flower) tasks (Appendix A.3.2).

Results Figure 5.7a shows how CoDeepNEAT discovers progressively better networks over time on the five tasks. Evolution often selects different weight initialization strategies for the different layers in these networks, so this scheme is already a flexible and powerful baseline.

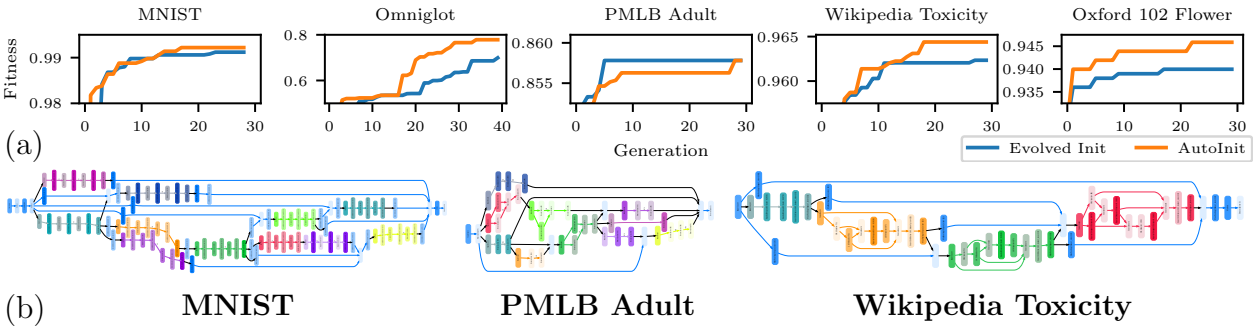


Figure 5.7: Evaluation of AutoInit with neural architecture search. (a) Performance improvement over generations in the five tasks. AutoInit outperforms the evolved initialization on four tasks and matches it on one. (b) Representative networks evolved with AutoInit. Although the networks are distinct, AutoInit initializes them properly, leading to good performance in each case.

However, by accounting for each model's unique topology and hyperparameters, AutoInit outperforms the baseline in four of the five tasks, and matches it in the fifth.

Beyond performance, three interesting phenomena can be observed. First, the mean population fitness varies greatly with the default initialization in each task, sometimes dropping significantly from one generation to the next (Figure 5.8). Though some variation is natural in a stochastic evolutionary process like CoDeepNEAT, AutoInit makes the discovery process more reliable by stabilizing the performance of the entire population.

Second, hyperparameters play a large role in the final performance of the dense networks, in particular in the "Oxford 102 Flower" task. While CoDeepNEAT discovers good models with both initialization strategies, performance is consistently higher with AutoInit. This finding agrees with Section 5.5, where AutoInit was shown to be robust to different hyperparameter values.

Third, while many networks exhibit motifs popular in existing architectures, such

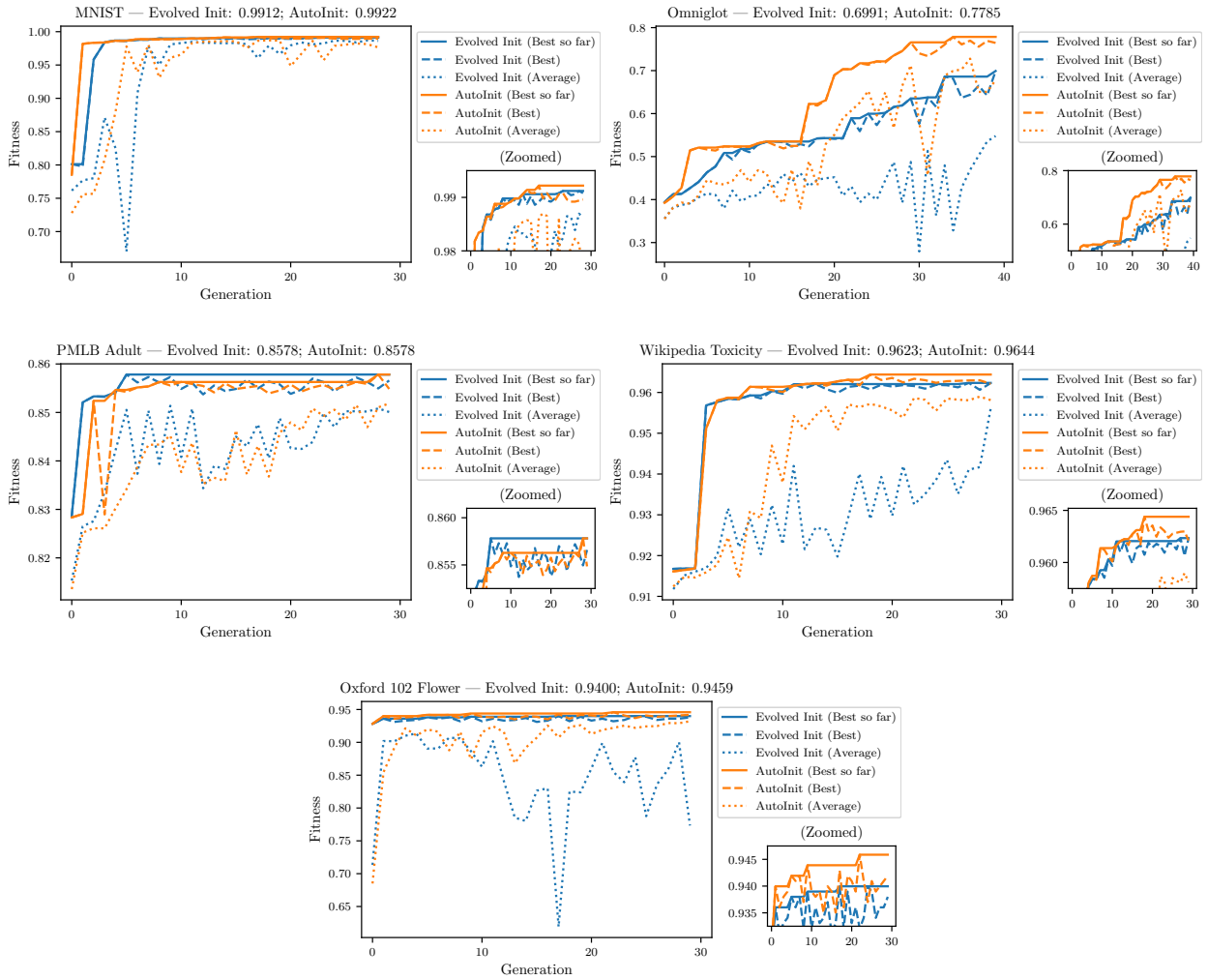


Figure 5.8: Progress of neural architecture search in the five tasks. The data is the same as that in Figure 5.7, but this plot also shows how AutoInit can stabilize mean population performance, leading to more reliable discovery of powerful models.

as alternating convolution and dropout layers and utilizing residual connections, other phenomena are less common (Figure 5.7b). For example, the networks make use of different activation functions and contain several unique information processing paths from the input to the output. Because AutoInit provides effective initialization in each of these cases, it allows for taking full advantage of unusual design choices that might otherwise hurt performance under default initialization schemes.

The results in this section suggest that AutoInit is an effective, general-purpose algorithm that provides a more effective initialization than existing approaches when searching for new models.

5.11 Enabling Activation Function Discovery

As new activation functions are developed in the future, it will be important to adjust weight initialization to maintain stable signal propagation. Since AutoInit makes this adjustment automatically, it is well-suited to the task. Indeed, Figure 5.1 confirmed that AutoInit improves performance with several existing activation functions. This section presents a more challenging task. To simulate future research in activation function design, hundreds of novel activation functions were generated as arbitrary computation graphs and trained with a CNN. AutoInit's ability to initialize each of these networks was then evaluated. The method for creating such activation functions is described first, followed by experimental details, and results on stability, performance, and generality.

Creating Novel Activation Functions An important area of automated machine learning (AutoML) is to discover new, better activation functions [11, 17, 121, 160]. Among existing

approaches, PANGAEA [15] has the most flexible search space and is therefore used to generate new functions in this section.

PANGAEA represents activation functions as computation graphs containing unary and binary operators (Figure 5.6b). Creating a novel activation function involves three steps. First, a minimal computation graph is initialized with randomly selected unary and binary operators. Second, the functional form of the activation function is modified by applying three random mutations to increase diversity. Third, the function is augmented with up to three learnable parameters. These parameters are analogous to those in other parametric activation functions, such as PReLU [72]; they are initialized to one and learned during training by gradient descent. Through this process, it is possible to understand to what extent AutoInit can improve performance with activation functions that have yet to be discovered.

Experimental Setup An important insight in this domain is that in addition to modifying the variance of the signals in a network, activation functions can induce mean shifts. Prior work encouraged stability by reparameterizing the weights to have zero empirical mean [19, 85, 158]. An alternative and more direct approach is to modify the activation function itself so that it does not cause a mean shift in the first place. Given an activation function f with Gaussian mean $\mu_f = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2/2} \, \mathrm{d}x$, this goal can be accomplished with $\tilde{f} \coloneqq f - \mu_f$, which has zero Gaussian mean. To take advantage of this idea, a version of AutoInit called AutoInit++ was created for this domain, thus extending AutoInit slightly beyond weight initialization.

Thus, three initialization strategies were compared. With the default initialization, weights were sampled from $\mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}, \frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}\right)$ [55]. With AutoInit, the

weights were sampled from $\mathcal{N}\left(0, 1/\sqrt{\texttt{fan_in}\mu_f}\right)$ to account for an arbitrary activation function f; the dropout adjustment (Section 5.5) was not used. Finally, AutoInit++ takes advantage of $\tilde{f}$ as described above, but is otherwise identical to AutoInit.

For each initialization strategy, 200 activation functions were created using the PANGAEA process. Each activation function was used with the All-CNN-C architecture on the CIFAR-10 dataset following the standard training setup. To avoid overfitting to the test set when evaluating such a large number of activation functions, the accuracy with a balanced validation set of 5000 images is reported instead.

Stability Achieving better-than-chance accuracy is a useful metric of training stability (Section 5.6). As shown in Figure 5.9a, many activation functions result in chance accuracy regardless of how the network is initialized. This phenomenon is not surprising; since the activation functions are arbitrary computation graphs, many of them will turn out to be poor. With the default initialization strategy, 149 activation functions caused training to fail in this way. With AutoInit, the number of failed activation functions dropped to 130, and with AutoInit++, it further decreased to 117. AutoInit and AutoInit++ thus make training more stable, allowing it to succeed for a greater number of activation functions.

Performance Beyond training stability, a good weight initialization should also improve performance. As a baseline, when trained with ReLU and the default initialization, All-CNN-C achieved 89.10% test accuracy. Twenty-two of 200 activation functions from the PANGAEA search space outperformed this accuracy with the default initialization. With AutoInit, this number increased to 26, and with AutoInit++, to 50—a notable improvement. Thus, with

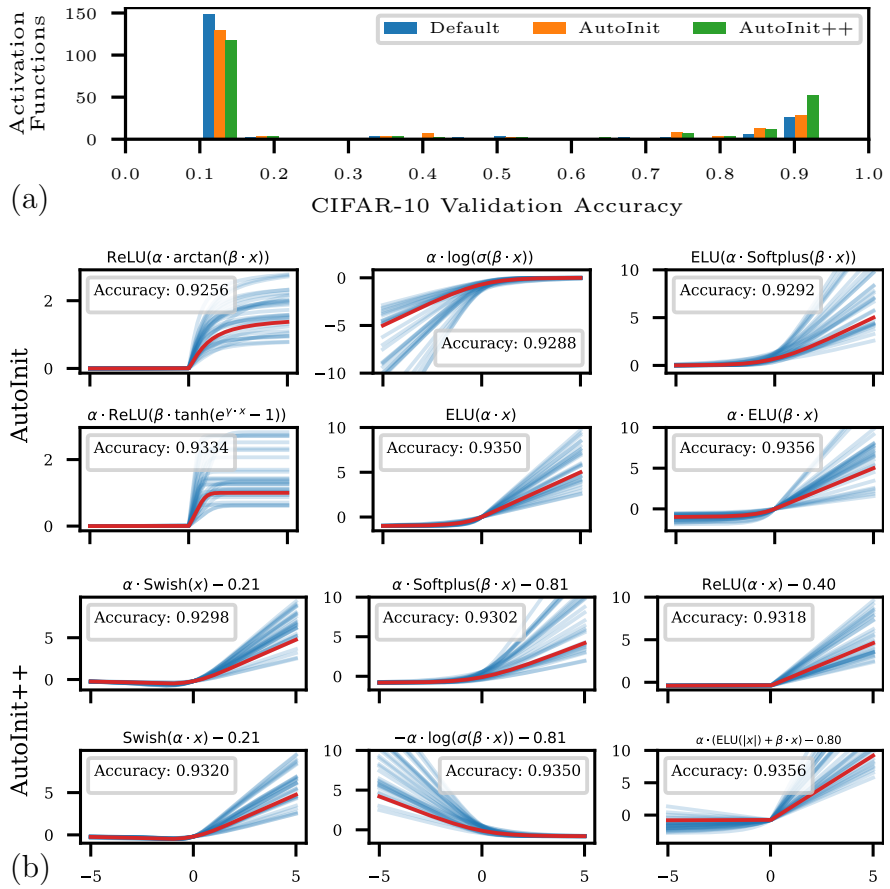


Figure 5.9: Evaluation of AutoInit with activation function discovery. (a) Distribution of accuracies achieved with 200 activation functions and different weight initialization strategies. AutoInit and AutoInit++ make training more stable and allow more high-performing activation functions to be discovered than the default initialization does. (b) High-performing activation functions. The red line shows the function at initialization, with $\alpha = \beta = \gamma = 1$. The blue lines show the shapes the activation function takes during training, created by sampling α, β, γ from $\mathcal{U}(0.5, 2.0)$. AutoInit's flexibility should turn out useful for developing new activation functions in the future.

the default initialization, one can naively create a randomly generated computation graph activation function and have roughly a one in nine chance of outperforming ReLU, but with AutoInit++, this probability increases to one in four.

Indeed, the Mann-Whitney U test [129] concludes that the distribution of accuracies induced by AutoInit++ is *stochastically larger* than that from AutoInit ($p < 0.05$) or the default initialization ($p < 0.01$). This result means that for any level of performance, it is always more probable to discover an activation function that achieves that level of performance when initializing with AutoInit++ versus AutoInit or the default initialization. The result implies that activation function researchers who properly initialize their networks are more likely to discover state-of-the-art activation functions, while staying with the default initialization may hinder that research effort. More detailed statistical significance analyses are included in Section 5.12.

Generality Figure 5.9b plots several activation functions from the PANGAEA search space. Many discovered functions have similar shapes to existing functions. However, others are nonmonotonic, have discontinuous derivatives, or saturate to nonzero values. These properties are less common in existing activation functions. This observation suggests that AutoInit is a general approach that does not depend on a specific type of activation function; it may therefore serve as a useful tool in developing new such functions in the future.

5.12 Statistical Significance of Results in Activation Function Meta-Learning

Sampling activation functions from the PANGAEA search space results in a distribution of possible models for each weight initialization strategy. Comparing the empirical distribution functions (EDFs) induced by each initialization strategy makes it possible to quantify the importance of the initialization [159].

Given n activation functions with errors $\{e_i\}$, the EDF $F(e) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}[e_i < e]$ gives the fraction of activation functions that result in error less than e. Let F_{default}, F_{AutoInit}, and $F_{\text{AutoInit++}}$ be the EDFs for the three initialization strategies. Figure 5.10 plots these EDFs along with the Kolmogorov-Smirnov test statistic $D = \sup_e |F_1(e) - F_2(e)|$, which measures the maximum vertical discrepancy between two EDFs [134]. This statistic shows that (1) AutoInit outperforms the default initialization ($D = 0.105$); (2) AutoInit++ delivers an even greater boost in performance over the default initialization ($D = 0.191$); and (3) AutoInit++ is measurably better than AutoInit ($D = 0.122$), confirming that having zero Gaussian mean is a useful property for activation functions to have.

Other ways of measuring statistical significance lead to similar conclusions. For instance, consider the null hypothesis that $F_{\text{default}} = F_{\text{AutoInit}}$. In other words, this null hypothesis states that AutoInit provides no benefit and that the accuracies obtained come from the same underlying distribution. With the Epps-Singleton test [45] this null hypothesis is rejected with $p < 0.05$. Similarly, the test rejects the null hypothesis that $F_{\text{default}} = F_{\text{AutoInit++}}$ with $p < 0.001$. Even stronger statements can be made in the case of AutoInit++. With the Mann-Whitney U test [129], the null hypothesis that $F_{\text{default}}(e) \geq F_{\text{AutoInit++}}(e)$ for some e is rejected ($p < 0.01$) in favor of the alternative that $F_{\text{default}}(e) < F_{\text{AutoInit++}}(e)$ for all e.

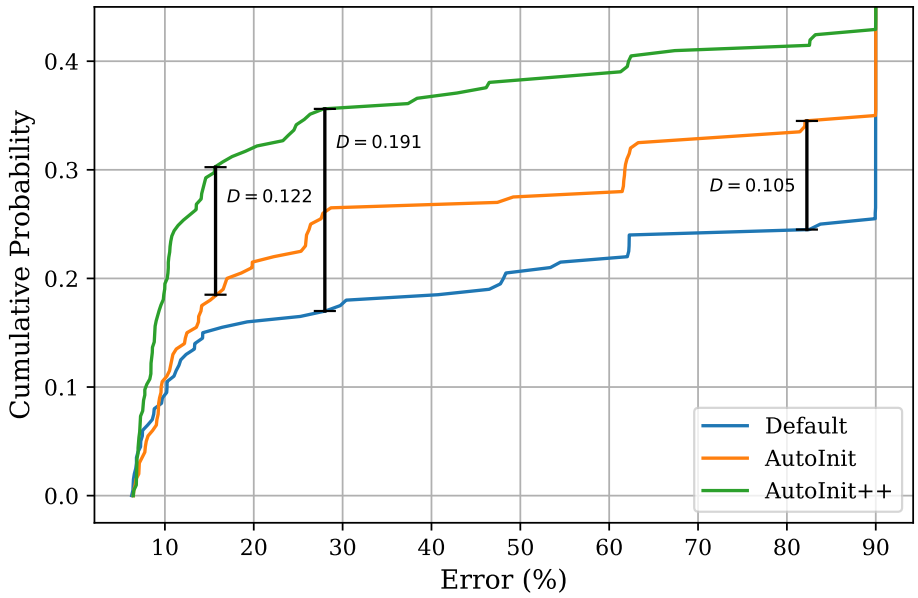


Figure 5.10: Error EDFs for PANGAEA activation functions when using different weight initialization strategies. The Kolmogorov-Smirnov statistic D quantifies the maximum vertical distance between the EDFs, and shows that proper initialization provides a measurable increase in expected performance. Notice that the x-axis shows percent error, and not accuracy as in Figure 5.9.

Similarly, the null hypothesis that $F_{\text{AutoInit}}(e) \geq F_{\text{AutoInit++}}(e)$ for some e is rejected ($p < 0.05$) in favor of the alternative that $F_{\text{AutoInit}}(e) < F_{\text{AutoInit++}}(e)$ for all e. As discussed above, this result states that the distribution of accuracies induced by AutoInit++ is stochastically larger than that from AutoInit or the default initialization.

5.13 Discussion

AutoInit is based on understanding and utilizing the training dynamics of neural networks, leading to higher and more robust performance, and facilitating further advances

in meta-learning. It can be improved and its scope broadened in several ways in the future, as outlined below.

Experiments in Other Domains The experiments in this chapter demonstrate that AutoInit can improve performance in a variety of settings, suggesting that it can be applied to other domains as well. For instance in reinforcement learning, good estimates of activation statistics are usually not available due to the online nature of the algorithm. It is not possible to stabilize training using e.g. batch normalization, but it may be possible to do it with AutoInit. Similarly, training of generative adversarial networks [63] is often unstable, and proper initialization may help. Applying AutoInit to such different domains should not only make them more reliable, but also lead to a better understanding of their training dynamics.

Accelerating Model Search In Sections 5.10 and 5.11, AutoInit was shown to facilitate the discovery of better neural network designs and activation functions. This ability is possible because AutoInit is a general method, i.e. not restricted to a single class of models, and it could similarly augment other meta-learning algorithms [e.g. those reviewed by 43, 197].

However, this finding points to an even more promising idea. As model search techniques become more prevalent in real-world applications, it will be most worthwhile to derive general principles rather than specific instantiations of those principles. For example, past weight initialization strategies improved performance with specific activation functions through manual derivation of appropriate weight scaling (Section 2.3.1). In contrast, AutoInit is a general method, leveraging Gaussian quadrature for any activation function. Similarly, AutoInit resulted in better initialization than strategies discovered by CoDeepNEAT through

evolution (Section 5.10). Further, AutoInit++ (Section 5.11), rather than producing a few high-performing activation functions, introduces the general property that activation functions with zero Gaussian mean ($\tilde{f} \coloneqq f - \mu_f$) tend to perform well. This property discovered a highly diverse set of powerful activation functions in the PANGAEA search space (Figure 5.9).

Thus, AutoInit is successful because it is not a single initialization strategy, but rather a mapping from architectures to initialization strategies. Such mappings, whether focused on initialization or some other aspect of model design, deserve increased attention in the future. They can lead to performance gains in a variety of scenarios. They also accelerate model search by focusing the search space to more promising regions. If one does not have to worry about discovering a good initialization, compute power can instead be used in other areas, like designing architectures and activation functions. Thus, general tools like AutoInit save time and resources, and lead to better models as a result.

Initial Weight Distributions AutoInit calculates appropriate weight scaling, but it does not impose a distribution from which weights are drawn (Equation 5.6). All experiments in this Chapter used a truncated normal distribution. In preliminary experiments, AutoInit also used untruncated normal, uniform, and orthogonal distributions, but no clear trends were observed. Indeed, assuming weights are scaled appropriately, whether training is stable depends only on the architecture and not the distribution from which weights are sampled [67]. However, this conclusion applies only in limited theoretical settings; in other settings, orthogonal initialization was found to be beneficial [83, 167]. Whether there is a single distribution that is optimal in every case, or whether certain distributions are better-suited

to different models, tasks, or layers, remains an open question, and a compelling direction for future research.

Variations of AutoInit Several variations of the core AutoInit algorithm can be devised that may improve its performance. For example, AutoInit stabilizes signals by analyzing the forward pass of activations from the input to the output of the network. It is possible to similarly model the backward pass of gradients from the output to the input. Indeed, past weight initialization strategies have sometimes utilized signals in both directions [7, 55]. It would be interesting to find out whether AutoInit could similarly benefit from analyzing backward-propagating signals.

Alternative objectives beyond mean and variance stabilization could also be considered. Two promising objectives are tuning the conditioning of the Fisher information matrix [153] and achieving dynamical isometry [200]. Mean field theory and nonlinear random matrix theory [154] could potentially be used to implement these objectives into AutoInit.

Support for New Layer Types AutoInit calculates outgoing mean and variance estimates for the majority of layer types available in current deep learning frameworks (Section 5.4). If AutoInit encounters an unknown layer, the default behavior is to assume that the mean and variance are not changed by that layer: $g_{\texttt{layer}}(\mu_{\text{in}}, \nu_{\text{in}}) = \mu_{\text{in}}, \nu_{\text{in}}$. This fallback mechanism tends to work well; if there are only a few unknown layers, then the variance estimation will be incorrect only by a constant factor and training can proceed. However, mean and variance estimation functions g can be derived for new types of layers as they are developed, either analytically or empirically with Monte Carlo sampling, thus taking full advantage of

AutoInit's ability to stabilize training in the future as well.

Tighter Integration with Deep Learning Frameworks Using AutoInit is simple in practice. The AutoInit package provides a wrapper around TensorFlow models. The wrapper automatically traverses the TensorFlow computation graph, calculates mean and variance estimations for each layer, and reinstantiates the model with the correct weight scaling. However, this implementation can be streamlined. The most effective approach would be to integrate AutoInit natively with deep learning frameworks like TensorFlow [1] and PyTorch [151]. Native integration would not just make AutoInit easier to use, it would also make it more accessible to general machine learning practitioners. For example, TensorFlow provides a few initialization strategies that can be leveraged by changing the `kernel_initializer` keyword in certain layers. However, implementing other weight initialization strategies requires subclassing from the `Initializer` base class, which is both time-consuming and complicated, especially for non-experts. Native integration would ensure that the benefits of smarter initialization are available immediately to the wider machine learning community.

5.14 Conclusion

This chapter introduced AutoInit, an algorithm that calculates analytic mean- and variance-preserving weight initialization for neural networks automatically. In convolutional networks, the initialization improved performance with different activation functions, dropout rates, learning rates, and weight decay settings. In residual networks, AutoInit prevented exploding signals, allowed training with higher learning rates, and improved performance with or without batch normalization. In transformers, AutoInit was scaled up to high-resolution

image classification, and improved performance with several activation functions with and without normalization. AutoInit also improved accuracy on the ImageNet dataset. The initialization is independent of data and is therefore efficient and reliable. AutoInit's generality proved instrumental in two types of AutoML. In neural architecture search, new architectures were evaluated more accurately, resulting in better networks in vision, language, tabular, multi-task, and transfer learning settings. In activation function discovery, AutoInit stabilized training and improved accuracy with a large diversity of novel activation functions. Thus, AutoInit serves to make machine learning experiments more robust and reliable, resulting in higher performance, and facilitating future research in AutoML. Although AutoInit accelerated activation function discovery, the process is still computationally expensive. To reduce this cost, the next chapter learns better activation functions in a data-driven way, improving efficiency by orders of magnitude.

Chapter 6

AQuaSurF: Efficient Activation Function Optimization through Surrogate Modeling

Activation functions are an important choice in neural network design [6, 149]. In order to realize the benefits of good activation functions, researchers often design new functions based on characteristics like smoothness, groundedness, monotonicity, and limit behavior. While these properties have proven useful, humans are ultimately limited by design biases and by the relatively small number of functions they can consider. On the other hand, automated search methods can evaluate thousands of unique functions, and as a result, often discover better activation functions than those designed by humans. However, such approaches do not usually have a theoretical justification, and instead focus only on performance. This limitation results in computationally inefficient ad hoc algorithms that may miss good solutions and may not scale to large models and datasets.

This chapter addresses these drawbacks in a data-driven way through three steps. First, in order to provide a foundation for theory and algorithm development, convolutional, residual, and vision transformer based architectures were trained from scratch with 2,913 different activation functions, resulting in three activation function benchmark datasets: `Act-Bench-CNN`, `Act-Bench-ResNet`, and `Act-Bench-ViT`. These datasets make it possible to analyze activation function properties at a large scale in order to determine which are

most predictive of performance.

The second step was to characterize the activation functions in these benchmark datasets analytically, leading to a surrogate performance measure. Exploratory data analysis revealed two activation function properties that are highly indicative of performance: (1) the spectrum of the Fisher information matrix associated with the model's predictive distribution at initialization, and (2) the activation function's output distribution. Both sets of features contribute unique information. They are both predictive of performance on their own, but they are most powerful when used in tandem. These features were combined to create a metric space where a low-dimensional representation of the activation functions could be learned. This space can then be used as a surrogate in the search for good activation functions.

In the third step, this surrogate was evaluated experimentally, first by verifying that it can discover known good functions in the benchmark datasets efficiently and reliably, and second by demonstrating that it can discover improved activation functions in new tasks on CIFAR-100 and ImageNet. The representation turned out to be so powerful that an out-of-the-box regression algorithm was able to search it effectively. The approach, called AQuaSurF (Activation Quality with a Surrogate Function), is orders of magnitude more efficient than past work. Indeed, whereas previous approaches evaluated hundreds or thousands of activation functions, AQuaSurF requires only tens of evaluations in order to discover functions that outperform a wide range of baseline activation functions in each context. Code implementing the AQuaSurF algorithm is available at `https://github.com/cognizant-ai-labs/aquasurf`.

Prior research on activation function optimization and Fisher information matrices is reviewed in Chapter 2. This work extends it in three ways. First, the benchmark

collections are made available at `https://github.com/cognizant-ai-labs/act-bench`, providing a foundation for further research on activation function optimization. Second, the low-dimensional representation of the Fisher information matrix makes it a practical surrogate measure, making it possible to apply it to not only activation function design, but potentially also to other applications in the future. Third, the already-discovered functions can be used immediately to improve performance in image processing tasks, and potentially in other tasks in the future.

6.1 Activation Function Benchmark Datasets

As the first step, this section introduces three activation function benchmark datasets: `Act-Bench-CNN`, `Act-Bench-ResNet`, and `Act-Bench-ViT`. Each dataset contains training results for 2,913 unique activation functions when paired with different architectures and tasks: All-CNN-C on CIFAR-10, ResNet-56 on CIFAR-10, and MobileViTv2-0.5 on Imagenette [74, 82, 104, 136, 174]. These functions were created using the main three-node computation graph from PANGAEA [17]. Details are in Appendix A.4.3.

Figure 6.1 shows the distribution of validation accuracies in these datasets. In all three datasets, the distribution is highly skewed towards functions that result in failed training. The plots suggest that it is difficult to design good activation functions, and explain why existing methods are often so computationally expensive. Notwithstanding this difficulty, the histograms show that many unique functions do achieve good performance. Thus, searching for new activation functions is a worthwhile task that requires a smart approach.

The scatter plots in Figure 6.2 show the same distribution of accuracies as the histograms in Figure 6.1, but through scatter plots that show how activation functions

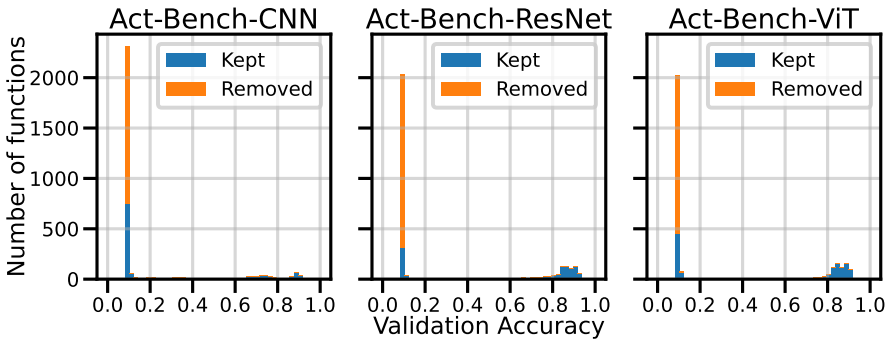


Figure 6.1: Distribution of validation accuracies with 2,913 unique activation functions from the three benchmark datasets. Many activation functions result in failed training (indicated by the chance accuracy of 0.1), suggesting that searching for activation functions is a challenging problem. However, most of these functions do not have valid FIM eigenvalues, and can thus be filtered out effectively.

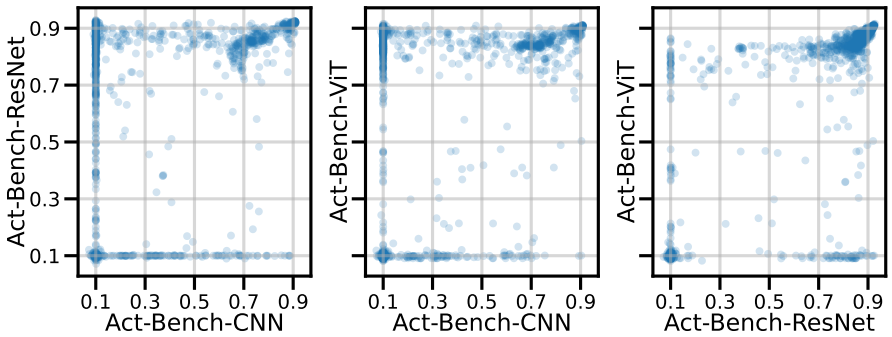


Figure 6.2: Distribution of validation accuracies across the benchmark datasets. Each point represents one activation function, and its x and y coordinates represent its performance in two of the three datasets. Some activation functions perform well on all tasks, and their improvements correlate. Others are more specialized and succeed only with one or two tasks.

perform across different tasks. Two interesting observations can be made. First, all three plots contain linearly correlated clusters of points in the upper right corner. This finding suggests that there are modifications to activation functions that make them more powerful, regardless of the task. Second, the clusters of points in the upper left and lower right corners represent activation functions that succeed in one task but fail in another, demonstrating that the best results come from discovering functions specialized to individual tasks.

The three benchmark datasets thus form a foundation for developing and evaluating methods for automated activation function design. In the next two sections, they are used to develop a measure that serves as surrogate performance metric, making it possible to scale up activation function optimization to large networks and datasets.

6.2 Fisher Information Matrix Details

In order to calculate the FIM, this chapter uses the K-FAC approach [65, 131, 132]. This technique is summarized in this section, with notation similar to that of Grosse and Martens [65].

6.2.1 Preliminaries

A feedforward neural network maps an input $\mathbf{a}_0 = \mathbf{x}$ to an output $\mathbf{a}_L = f(\mathbf{x}; \boldsymbol{\theta})$ through a series of L layers. Each layer $l \in \{1, \ldots, L\}$ is comprised of a weight matrix $\mathbf{W}_l$, a bias vector $\mathbf{b}_l$, and an element-wise activation function ϕ_l. With $\bar{\mathbf{W}}_l = \begin{pmatrix} \mathbf{b}_l & \mathbf{W}_l \end{pmatrix}$ and

$\bar{\mathbf{a}}_l = \begin{pmatrix} 1 & \mathbf{a}_l^\top \end{pmatrix}^\top$, each layer implements the transformation

$$\mathbf{s}_l = \bar{\mathbf{W}}_l \bar{\mathbf{a}}_{l-1}, \tag{6.1}$$

$$\mathbf{a}_l = \phi_l(\mathbf{s}_l). \tag{6.2}$$

Let $\boldsymbol{\theta} = \begin{pmatrix} \mathrm{vec}(\bar{\mathbf{W}}_1)^\top & \cdots & \mathrm{vec}(\bar{\mathbf{W}}_L)^\top \end{pmatrix}^\top$ represent the vector of all network parameters. Parameterized by $\boldsymbol{\theta}$ and given inputs $\mathbf{x}$ drawn from a training distribution $Q_{\mathbf{x}}$, the neural network defines the conditional distribution $R_{\mathbf{y}|f(\mathbf{x};\boldsymbol{\theta})}$. The Fisher information matrix associated with this model is

$$\mathbf{F} = \mathop{\mathrm{E}}_{\substack{\mathbf{x}\sim Q_{\mathbf{x}} \\ \mathbf{y}\sim R_{\mathbf{y}|f(\mathbf{x};\boldsymbol{\theta})}}} \left[\nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{y}, f(\mathbf{x};\boldsymbol{\theta})) \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{y}, f(\mathbf{x};\boldsymbol{\theta}))^\top \right]. \tag{6.3}$$

As usual in deep learning, the loss function $\mathcal{L}(\mathbf{y}, \mathbf{z})$ represents the negative log-likelihood associated with $R_{\mathbf{y}|f(\mathbf{x};\boldsymbol{\theta})}$ and quantifies the discrepancy between the model's prediction $\mathbf{z} = f(\mathbf{x};\boldsymbol{\theta})$ and the true label $\mathbf{y}$. The network is trained to minimize the loss by updating its parameters according to the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{y}, f(\mathbf{x};\boldsymbol{\theta}))$.

6.2.2 Approximations

For ease of notation, write $\mathcal{D}\mathbf{v} = \nabla_{\mathbf{v}} \mathcal{L}(\mathbf{y}, f(\mathbf{x};\boldsymbol{\theta}))$. Recalling that $\boldsymbol{\theta} = \begin{pmatrix} \mathrm{vec}(\bar{\mathbf{W}}_1)^\top & \cdots & \mathrm{vec}(\bar{\mathbf{W}}_L)^\top \end{pmatrix}^\top$, the FIM can be expressed as an $L \times L$ block matrix:

$$\mathbf{F} = \begin{pmatrix} \mathrm{E}\left[\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_1)\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_1)^\top\right] & \cdots & \mathrm{E}\left[\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_1)\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_L)^\top\right] \\ \vdots & \ddots & \vdots \\ \mathrm{E}\left[\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_L)\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_1)^\top\right] & \cdots & \mathrm{E}\left[\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_L)\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_L)^\top\right] \end{pmatrix}. \tag{6.4}$$

Note that $\mathcal{D}\bar{\mathbf{W}}_l = \mathcal{D}\mathbf{s}_l\bar{\mathbf{a}}_{l-1}^\top$, and recall that $\text{vec}(\mathbf{u}\mathbf{v}^\top) = \mathbf{v} \otimes \mathbf{u}$. Each block of the FIM can be written as

$$\begin{aligned}
\mathbf{F}_{i,j} &= \text{E}\left[\text{vec}(\mathcal{D}\bar{\mathbf{W}}_i)\text{vec}(\mathcal{D}\bar{\mathbf{W}}_j)^\top\right] && (6.5)\\
&= \text{E}\left[\text{vec}(\mathcal{D}\mathbf{s}_i\bar{\mathbf{a}}_{i-1}^\top)\text{vec}(\mathcal{D}\mathbf{s}_j\bar{\mathbf{a}}_{j-1}^\top)^\top\right] && (6.6)\\
&= \text{E}\left[(\bar{\mathbf{a}}_{i-1} \otimes \mathcal{D}\mathbf{s}_i)(\bar{\mathbf{a}}_{j-1} \otimes \mathcal{D}\mathbf{s}_j)^\top\right] && (6.7)\\
&= \text{E}\left[(\bar{\mathbf{a}}_{i-1} \otimes \mathcal{D}\mathbf{s}_i)(\bar{\mathbf{a}}_{j-1}^\top \otimes \mathcal{D}\mathbf{s}_j^\top)\right] && (6.8)\\
&= \text{E}\left[\bar{\mathbf{a}}_{i-1}\bar{\mathbf{a}}_{j-1}^\top \otimes \mathcal{D}\mathbf{s}_i\mathcal{D}\mathbf{s}_j^\top\right]. && (6.9)
\end{aligned}$$

Two approximations are necessary in order to make representation of the FIM practical. First, assume that different layers have uncorrelated weight derivatives. The FIM can then be approximated as a block diagonal matrix, with $\mathbf{F}_{i,j} = \mathbf{0}$ if $i \neq j$. Second, if one approximates the pre-activation derivatives $\mathcal{D}\mathbf{s}_l$ and activations $\bar{\mathbf{a}}_{l-1}^\top$ as independent, then the diagonal blocks of the FIM can be further decomposed into the Kronecker product of two smaller matrices:

$$\mathbf{F}_{l,l} = \text{E}\left[\bar{\mathbf{a}}_{l-1}\bar{\mathbf{a}}_{l-1}^\top \otimes \mathcal{D}\mathbf{s}_l\mathcal{D}\mathbf{s}_l^\top\right] \approx \text{E}\left[\bar{\mathbf{a}}_{l-1}\bar{\mathbf{a}}_{l-1}^\top\right] \otimes \text{E}\left[\mathcal{D}\mathbf{s}_l\mathcal{D}\mathbf{s}_l^\top\right]. \tag{6.10}$$

Let $\mathbf{\Omega}_l = \text{E}\left[\bar{\mathbf{a}}_l\bar{\mathbf{a}}_l^\top\right]$ and $\mathbf{\Gamma}_l = \text{E}\left[\mathcal{D}\mathbf{s}_l\mathcal{D}\mathbf{s}_l^\top\right]$. The approximate empirical FIM is then written as

$$\hat{\mathbf{F}} = \begin{pmatrix} \mathbf{\Omega}_0 \otimes \mathbf{\Gamma}_1 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \mathbf{\Omega}_{L-1} \otimes \mathbf{\Gamma}_L \end{pmatrix}. \tag{6.11}$$

6.2.3 Layer-Specific Implementation

The above example illustrates FIM approximation for a simple feedforward network. However, most modern architectures contain several different kinds of layers. Some layers

like pooling, normalization, or dropout layers do not have trainable weights, and therefore these layers are not included in the FIM [88, 175].

Each diagonal entry $\mathbf{\Omega}_{l-1} \otimes \mathbf{\Gamma}_l$ corresponds to one layer with weights. The calculation differs slightly depending on the layer type, but otherwise the example above can be straightforwardly extended to more complicated networks. Calculations for three common layer types are presented below.

Dense Layers For dense layers, the matrices $\mathbf{\Omega}_{l-1}$ and $\mathbf{\Gamma}_l$ can be readily computed with one forward and backward pass through the network using a mini-batch of data. The eigenvalues are then computed using standard techniques.

Convolutional Layers Convolutional layers require special consideration to calculate $\mathbf{\Omega}_{l-1}$ and $\mathbf{\Gamma}_l$. For a given layer, let M represent the batch size, $\mathcal{T}$ the set of spatial locations (typically a two-dimensional grid), Δ the set of spatial offsets from the center of the filter, and I and J the number of output and input maps, respectively. The activations are represented by the $M \times |\mathcal{T}| \times J$ array $\mathbf{A}_{l-1}$. The weights are represented by the $I \times |\Delta| \times J$ array $\mathbf{W}_l$ which is interpreted as an $I \times |\Delta| J$ matrix. The expansion operator $[\![\cdot]\!]$ extracts patches around each spatial location and flattens them into vectors that become the rows of a matrix: $[\![\mathbf{A}_{l-1}]\!]$ is a $M|\mathcal{T}| \times J|\Delta|$ matrix.

Similar to the feedforward networks, the bias (if used) can be prepended to the weights matrix as $\bar{\mathbf{W}}_l = \begin{pmatrix} \mathbf{b}_l & \mathbf{W}_l \end{pmatrix}$ and a homogeneous column of ones to the expanded activations

as $[\![\mathbf{A}_{l-1}]\!]_H = \begin{pmatrix}\mathbf{1} & [\![\mathbf{A}_{l-1}]\!]\end{pmatrix}$. This constructions allows the forward pass to be written as

$$\mathbf{S}_l = [\![\mathbf{A}_{l-1}]\!]_H \bar{\mathbf{W}}_l^\top, \tag{6.12}$$

$$\mathbf{A}_l = \phi\left(\mathbf{S}_l\right), \tag{6.13}$$

from which the factors are computed as

$$\mathbf{\Omega}_l = \mathrm{E}\left[[\![\mathbf{A}_l]\!]_H^\top [\![\mathbf{A}_l]\!]_H\right], \tag{6.14}$$

$$\mathbf{\Gamma}_l = \frac{1}{|\mathcal{T}|}\mathrm{E}\left[\mathcal{D}\mathbf{S}_l^\top \mathcal{D}\mathbf{S}_l\right]. \tag{6.15}$$

Depthwise Convolutional Layers Depthwise convolutional layers utilize separate kernels for each channel. In this case, $[\![\mathbf{A}_{l-1}]\!]$ is a $M|\mathcal{T}|J \times |\Delta|$ matrix. Otherwise, the factors $\mathbf{\Omega}_{l-1}$ and $\mathbf{\Gamma}_l$ are calculated in the same way as they are for standard convolutional layers.

6.2.4 Eigenvalue Calculation

Because $\hat{\mathbf{F}}$ is a block-diagonal matrix, its eigenvalues are simply the combined eigenvalues of each block: $\lambda(\hat{\mathbf{F}}) = \{\lambda(\hat{\mathbf{F}}_l)\}_{l=1}^L$. The eigenvalue calculation for one block $\hat{\mathbf{F}}_l = \mathbf{\Omega}_{l-1} \otimes \mathbf{\Gamma}_l$ is further simplified by first computing the eigenvalues $\lambda(\mathbf{\Omega}_{l-1})$ and $\lambda(\mathbf{\Gamma}_l)$ for each Kronecker factor separately and then returning all pairwise products from the two sets of eigenvalues. For numerical stability, the eigenvalues can first be log-scaled and then all pairwise sums from the two sets are returned. Calculating the eigenvalues requires one forward and backward pass through the network with a mini-batch of data. The computational cost is therefore relatively cheap, especially compared with the cost of fully training a network from scratch.

It is possible for the FIM eigenvalues to be invalid. For example, if the forward propagated activations or backward propagated gradients explode or vanish, then the diagonal

entries $\mathbf{\Omega}_{l-1} \otimes \mathbf{\Gamma}_l$ may be undefined. Such invalid values result from activation functions that are unstable. Therefore, invalid FIM eigenvalues provide a good way to filter out bad activation functions.

6.3 Features and Distance Metrics

In order to make efficient search for good activation functions possible, the surrogate space needs to be low-dimensional, represent informative features, and have an appropriate distance metric. In the second step, an approach is developed based on two kinds of features: the eigenvalues of the Fisher information matrix, and the outputs of the activation function. In this section, each feature type is motivated first, and a metric is then developed for computing distances between activation functions. They are then put together into a surrogate in the next section.

6.3.1 FIM Eigenvalues

The Fisher information matrix (FIM) is an important concept in characterizing neural network models. Viewed from various perspectives, the FIM determines a neural network's capacity for learning, ability to generalize, the robustness of the network to small perturbations of its parameters, and the geometry of the loss function near the global minimum [52, 71, 92–94, 116, 117].

The FIM has $|\boldsymbol{\theta}|$ eigenvalues. The distribution of eigenvalues can be represented by binning the eigenvalues into an m-bucket histogram, and this m-dimensional vector serves as a computational characterization of the network. Importantly, different activation functions induce different FIM eigenvalues for a given neural network. They can be calculated at

initialization and do not require training; they can thus serve as a low-dimensional feature-vector representation of the activation function. The FIM eigenvalues are immediately useful for filtering out poor activation functions; if they are invalid, the activation function is likely to fail in training (Figure 6.1). However, in order to use them as a surrogate, a distance metric needs to be defined.

Given a neural network architecture f, let f_ϕ and f_ψ be two instantiations with different activation functions ϕ and ψ. Let μ_l and ν_l represent the distributions of eigenvalues corresponding to the weights in layer l of neural networks f_ϕ and f_ψ, respectively, and let w_l be the number of weights in layer l of the networks. The distance between f_ϕ and f_ψ is then computed as a weighted layer-wise sum of 1-Wasserstein distances

$$d(f_\phi, f_\psi) = \sum_{l=1}^{L} \frac{W_1(\mu_l, \nu_l)}{w_l}. \tag{6.16}$$

With this distance metric, the FIM eigenvalue vector representations encode a low-dimensional embedding space for activation functions, making efficient search for good functions possible. Note, however, that the FIM eigenvalues incorporate multiple sources of information, including the activation function, neural network structure, data distribution, and loss function. This fact makes the eigenvalues powerful features, but also introduces noise in the prediction process. Fortunately, it is possible to combine them with another feature, activation function outputs, to address this shortcoming.

6.3.2 Activation Function Outputs

The shape of an activation function ψ can be described by a vector of n sample values $\psi(x)$. If the network's weights are appropriately initialized, the input activations to its

neurons are initially distributed as $\mathcal{N}(0,1)$ [16]. Therefore, the sampling $x \sim \mathcal{N}(0,1)$ provides an n-dimensional feature vector that represents the expected use of the activation function at initialization. A distance metric in this feature vector space can be defined naturally as the Euclidean distance

$$d(f_\phi, f_\psi) = \sqrt{\frac{\sum_{i=1}^{n}(\phi(x_i) - \psi(x_i))^2}{n}}, \quad x \sim \mathcal{N}(0,1). \tag{6.17}$$

Functions with similar shapes will have a small distance between them, while those with different shapes will have a large distance. Because these output feature vectors depend only on the activation function, they are reliable and inexpensive to compute. Most importantly, together with the FIM eigenvalues, they constitute a powerful surrogate search space, as will be demonstrated in the next section.

6.4 Using the Features as a Surrogate

In this section, the UMAP dimensionality reduction technique is used to visualize the FIM and output features across the benchmark datasets. This visualization leads to a combined surrogate space that can be used to accelerate the search for good activation functions.

6.4.1 Visualization with UMAP

The features developed above can be visualized using the UMAP algorithm [135]. UMAP is a general dimension reduction approach similar to t-SNE, but is better at scaling to large sample sizes and preserving global structure [187]. As a first demonstration, Figure 6.3 shows a 2D representation of the 2,913 activation functions in the benchmark datasets.

Each function was represented as an 80-dimensional vector of output values. Interpolating between embedded points confirms that UMAP learns a good underlying representation.

UMAP was also used to project the activation functions to nine two-dimensional spaces according to the distance metrics in Equations 6.16 and 6.17. In Figure 6.4, each column represents a different benchmark dataset (`Act-Bench-CNN`, `Act-Bench-ResNet`, or `Act-Bench-ViT`) and each row a different distance metric (FIM eigenvalues with $m = \lfloor |\boldsymbol{\theta}|/100 \rfloor$, activation function outputs with $n = 1{,}000$, or both). The plots include only activation functions that were not filtered out. Each point represents a unique activation function, and the points are colored according to their validation accuracy in the benchmark task. Note that although the performance of each activation function is already known, this information was not given to UMAP; the embeddings are entirely unsupervised.

Thus, the visualizations in Figure 6.4 illustrate how predictive each feature type is of activation function performance in each dataset. The next subsections evaluate each feature type in this role in detail, and show that utilizing both features provides better results than either feature alone. Details are presented in Section 6.5.

6.4.2 FIM Eigenvalues

The first row of Figure 6.4 shows the 2D UMAP embeddings of the FIM eigenvalue vectors associated with each activation function. There are several clusters in these plots where the points share similar colors. These regions indicate distinct activation functions with similar FIM eigenvalues. Such functions induce similar training dynamics in the neural network and eventually lead to similar performance. On the other hand, some clusters contain activation functions with a wide range of performances, and some points do not belong to

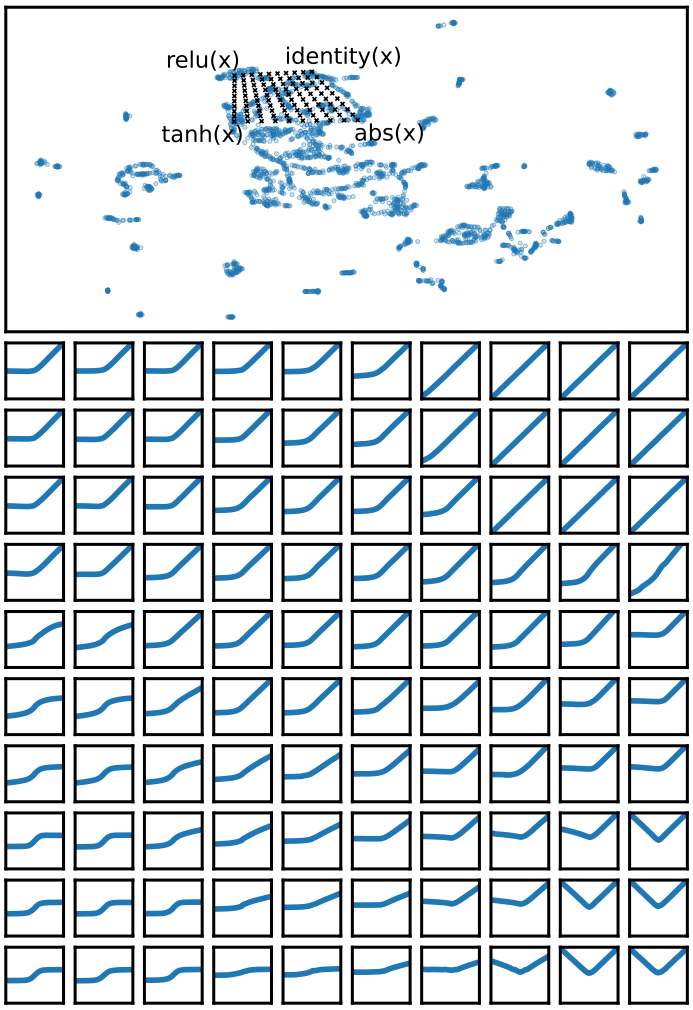


Figure 6.3: UMAP embedding of the 2,913 activation functions in the benchmark datasets. Each point stands for a unique activation function, represented by an 80-dimensional output feature vector. The embedding locations of four common activation functions are labeled. The black x's mark coordinates interpolating between these four functions, and the grid of plots on the bottom shows reconstructed activation functions at each of these points. UMAP interpolates smoothly between different kinds of functions, suggesting that it is a good approach for learning low-dimensional representations of activation functions.

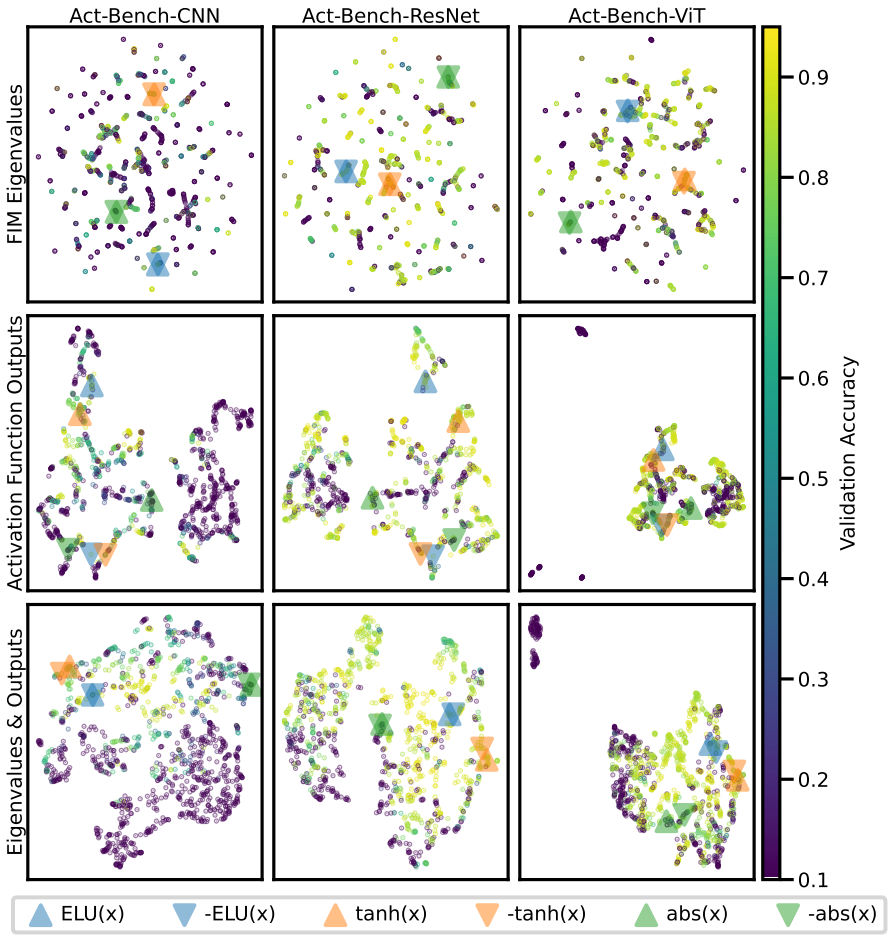


Figure 6.4: UMAP embeddings of activation functions for each dataset (column) and feature type (row). Each point represents a unique activation function, and the points are colored according to their validation accuracy on the given dataset. The colored triangles identify the locations of six well-known activation functions. The areas of similar performance are more continuous in the bottom row; that is, using both FIM eigenvalues and activation function outputs provides a better low-dimensional representation than using either feature alone.

any cluster at all. Overall, the plots suggest that FIM eigenvalues are a useful predictor of performance, but also that incorporating additional information could lead to better results.

6.4.3 Activation Function Outputs

The middle row of Figure 6.4 shows the 2D UMAP embeddings of the output vectors associated with each activation function. Points are close to each other in this space if the corresponding activation functions have similar shapes. These plots are demonstrably more informative than the plots based on the FIM eigenvalues in three ways. First, the purple points are better separated from the others. This separation means that activation functions that fail (those achieving 0.1 chance accuracy) are better separated from those that do well. Second, most points' immediate neighbors have similar colors. This similarity means that activation functions with similar shapes lead to similar accuracy, and analyzing activation function outputs on their own is more informative than analyzing the FIM eigenvalues. Third, the plots include multiple regions where there are one-dimensional manifolds that exhibit smooth transitions in accuracy, from purple to blue to green to yellow. Thus, not only does UMAP successfully embed similar activation functions near each other, but it also is able to organize the activation functions in a meaningful way.

There is one drawback to this approach: the performant activation functions (those represented by yellow dots) are often in distinct clusters. This dispersion means that a search algorithm would have to explore multiple areas of the search space in order to find all of the best functions. As the next subsection suggests, this issue can be alleviated by utilizing both FIM eigenvalues and activation function outputs.

6.4.4 Combining Features: Eigenvalues & Outputs

The UMAP algorithm uses an intermediate fuzzy topological representation to represent relationships between data points, similar to a neighborhood graph. This property makes it possible to combine multiple sources of data by taking intersections or unions of the representations in order to yield new representations [135]. The bottom row of Figure 6.4 utilizes both FIM eigenvalues and activation function outputs by taking the union of the two representations. Thus, activation functions are embedded close to each other in this space if they have similar shapes, if they induce similar FIM eigenvalues, or both.

The bottom row of Figure 6.4 shows the benefits of combining the two features. Unlike the activation function output plots, which contain multiple clusters of high-performing activation functions in different locations in the embedding space, the combined UMAP model embeds all of the best activation functions in similar regions. The combined UMAP model also places poor activation functions (purple points) in the edge of the embedding space, and brings good functions (yellow points) to the center. Thus, the embedding space is more convex, and therefore easier to optimize.

In general, activation functions with similar shapes lead to similar performances, and those with different shapes often produce different results. This property is why the middle row of Figure 6.4 appears locally smooth. However, in some cases the shape of the activation function does not tell the whole story, and additional information is needed to ascertain its performance.

For example, the colored triangles in Figure 6.4 identify the location of six activation functions in the low-dimensional space: $\mathrm{ELU}(x)$, $-\mathrm{ELU}(x)$, $\tanh(x)$, $-\tanh(x)$, $|x|$, and $-|x|$.

In the activation function output space (middle row), all of these functions are mapped to different regions of the space. The points are spread apart because an activation function and its negative have very different shapes, i.e. their output will be different for every nonzero input (Figure 6.5). In contrast, in the FIM eigenvalue space (top row of Figure 6.4), the points for these pairs of functions overlap because the FIM eigenvalues are comparable (Figure 6.5). Indeed, assuming the weights are initialized from a distribution symmetric about zero, negating an activation function does not change the training dynamics of a neural network, and they are functionally equivalent.

This issue complicates the search process in two ways. The first is that good activation functions are mapped to different regions of the embedding space, and so a search algorithm must explore multiple areas in order to find the best function. The second challenge is that distinct regions of the space may contain redundant information: if $\mathrm{ELU}(x)$ is known to be a good activation function, it is not helpful to spend compute resources evaluating $-\mathrm{ELU}(x)$ only to discover that it achieves the same performance.

Negating an activation function is a clear example of a modification that changes the shape of the activation function, but does not affect the training of a neural network. More broadly, it is likely that there exist activation functions that differ in other ways (besides just negation), but that still induce similar training dynamics in neural networks. Fortunately, utilizing FIM eigenvalues and activation function outputs together provides enough information to tease out these relationships. FIM eigenvalues take into account the activation function, the neural network architecture, the loss function, and the data distribution. The eigenvalues are more meaningful features than activation function outputs, which only depend on the shape of the function. However, as Figure 6.4 shows, the FIM

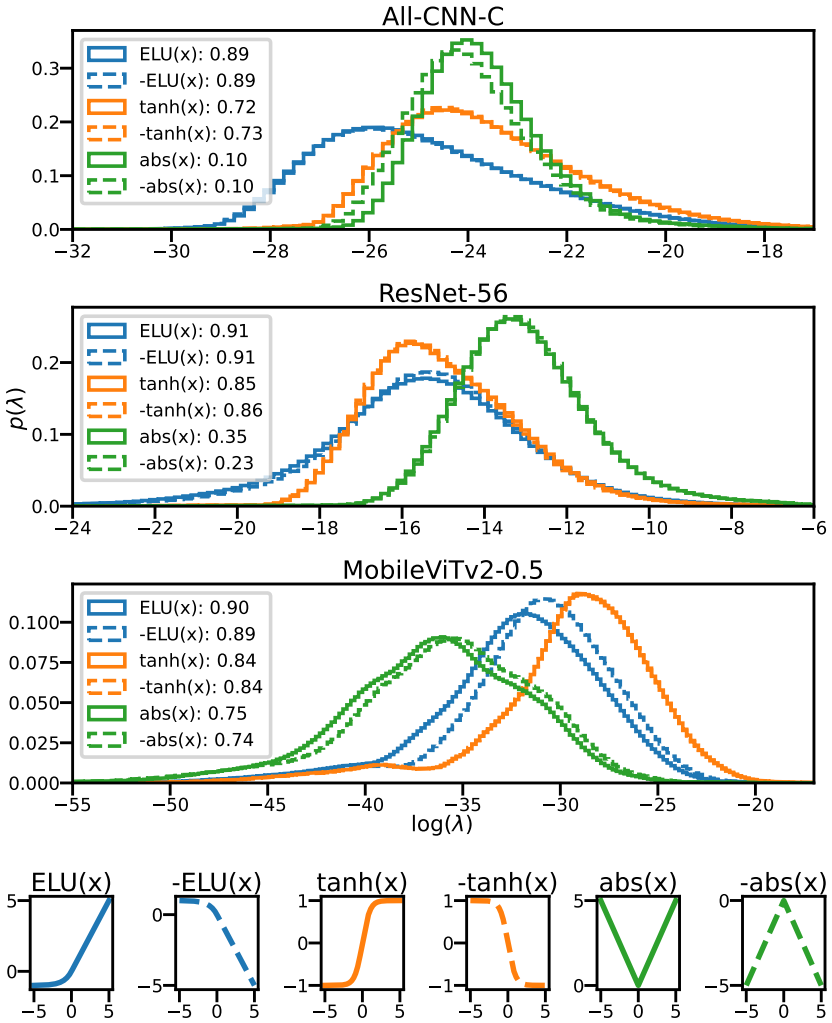


Figure 6.5: FIM eigenvalue distributions for different architectures and activation functions. The legends show the activation function and the corresponding validation accuracy in different tasks. Although negating an activation function changes its shape, it does not substantially change its behavior nor its performance. FIM eigenvalues capture this relationship between activation functions. The eigenvalues are thus useful for finding activation functions that appear different but in fact behave similarly, and these discoveries in turn improve the efficiency of activation function search.

eigenvalues are noisier features, while the activation function outputs are quite reliable. Thus, utilizing both features is a natural way to combine their strengths and address their weaknesses.

6.4.5 Constructing a Surrogate

These observations suggest an opportunity for an effective surrogate measure: The UMAP coordinates in the bottom row of Figure 6.4 have the information needed to predict how well an activation function will perform. They capture the essence of the m- and n-dimensional feature vectors, and distill it into a 2D representation that can be computed efficiently and used to guide the search for good functions. As the third step in this research, the next two sections evaluate this process experimentally, demonstrating that it is efficient and reliable, and that it scales to new and challenging datasets and search spaces.

6.5 Features and Surrogate Details

This section describes how the activation function features were implemented and how the surrogate was constructed.

Calculating FIM Eigenvalues The FIM eigenvalues were calculated for each activation function as discussed in Section 6.3. The eigenvalues were log-scaled for numerical stability. By definition, the number of eigenvalues is the same as the number of weights in the neural network. To save space, the eigenvalues were binned to histograms. For a layer l with $|\boldsymbol{\theta}_l|$ weights, $\lfloor |\boldsymbol{\theta}_l|/100 \rfloor$ equally sized bins from -100 to 100 were used. One histogram was computed for each layer in a network, and all of the histograms were concatenated together into

a single feature vector for a given activation function. In this manner, the total dimensionality was 13,692 for All-CNN-C, 16,500 for ResNet-56, and 11,013 for MobileViTv2-0.5.

Calculating Activation Function Outputs The activation function outputs $y = f(x)$ were calculated for each activation function f by sampling n =1,000 values $x \sim \mathcal{N}(0, 1)$ and truncating to the range $[-5, 5]$. The same random inputs were used for all activation functions.

Per-Layer FIM Eigenvalues In Figure 6.5, the eigenvalues for the entire network are shown for completeness. However, the UMAP representations shown in Figure 6.4 were produced by keeping the eigenvalues at each layer separate and computing a weighted distance between them (according to Equation 6.16). As pointed out above, FIM eigenvalues are informative but noisy features. In preliminary experiments, keeping the eigenvalues separate at each layer reduced some of this noise, resulting in a more informative Figure 6.4 and consequently improving the performance of the search algorithms.

UMAP Settings UMAP exposes a number of parameters that can be used to customize its behavior [135]. The `metric` parameter determines how distances are computed between points, the `n_neighbors` parameter adjusts the tradeoff between the local and global structure of the data, and the `min_dist` parameter controls the minimum distance between points in the embedding space.

The plots in Figure 6.4 were produced by computing the distances between FIM eigenvalues and activation function outputs. For the FIM eigenvalues `UMAP(metric='manhattan',`

`n_neighbors=3, min_dist=0.1)` was used, and for the activation function outputs `UMAP(metric='euclidean', n_neighbors=15, min_dist=0.1)` was used. The distance metrics were chosen to implement Equations 6.16 and 6.17.

In preliminary experiments, decreasing `n_neighbors` from the default of 15 down to 3 for the FIM eigenvalues qualitatively improved the embedding for the combined features. The combined features were visualized with a union model, i.e. `umap_combined = umap_fim_eigs + umap_fn_outputs` [135].

6.6 Searching on the Benchmark Tasks

Searching for activation functions typically requires training a neural network from scratch in order to evaluate each candidate function fully, which is often computationally expensive. With the benchmark datasets, all of the results are already precomputed. This information makes it possible to experiment with different search algorithms and conduct repeated trials to understand the statistical significance of the results. These results serve to inform both algorithm design and feature selection, as demonstrated in this section.

6.6.1 Setup

Three algorithms were evaluated: weighted k-nearest regression with $k = 3$ (KNR), random forest regression (RFR), and support vector regression (SVR). Gaussian Process Regression (GPR) was also evaluated but found to be inconsistent in preliminary experiments (Appendix A.4). Random search (RS) was included as a baseline comparison; it did not utilize the FIM eigenvalue filtering mechanism. The algorithms were used out of the box with default hyperparameters from the scikit-learn package [152]. They were provided different

activation function features in order to understand their potential to predict performance. The features included FIM eigenvalues, activation function outputs, or both. The features were preprocessed and embedded in a two-dimensional space by UMAP. These representations are visualized in Figure 6.4; the coordinates of each point correspond exactly to the information given to the regression algorithms.

The ReLU activation function is ubiquitous in machine learning. For many neural network architectures, the performance with ReLU is already known [6, 146, 149], which makes it a good starting point for search. For this reason, the search algorithms began by evaluating ReLU and seven other randomly chosen activation functions. In general, such evaluation requires training from scratch, but with the benchmark datasets, it requires only looking up the precomputed results. The algorithms then used the validation accuracy of these eight functions to predict the performance of all unevaluated functions in the dataset. The activation function with the highest predicted accuracy was then evaluated. The performance of this new function was then added to the list of known results, and this process continued until 100 activation functions had been evaluated. Each experiment comprising a different search algorithm, activation function feature set, and benchmark dataset was repeated 100 times. The full experimental details are in Appendix A.4.

6.6.2 Results

Figure 6.6 shows the results of the searches. Importantly, the curves do not depict just one search trial. Instead, they represent the average performance aggregated from 100 independent runs, which is made possible by the benchmark datasets. As indicated by the shaded confidence intervals, the results are reliable and are not simply due to chance.

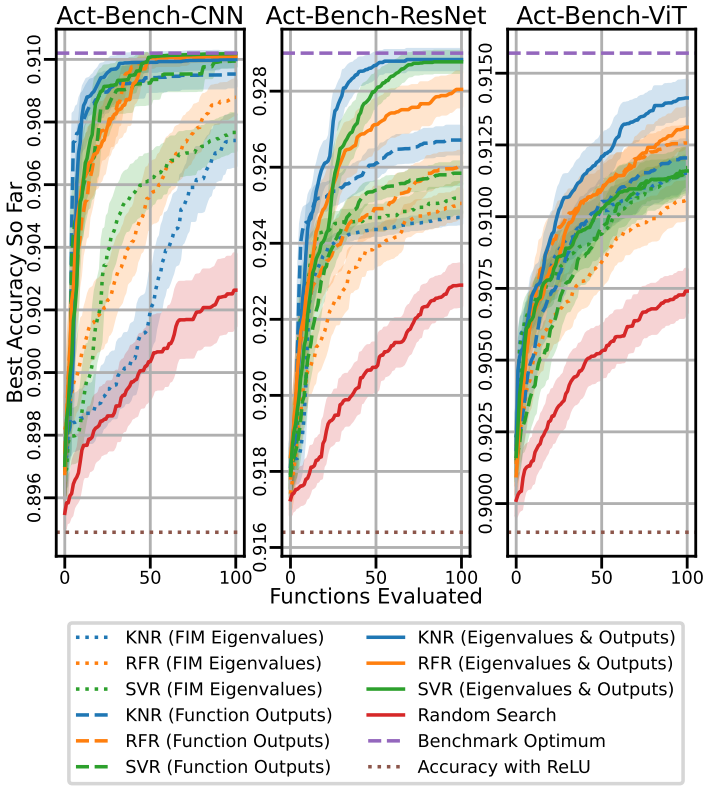


Figure 6.6: Search results on the three benchmark datasets. Each curve represents a different search algorithm (KNR, RFR, or SVR) utilizing a different UMAP feature (FIM eigenvalues, function outputs, or both; these features are visualized in Figure 6.4). The curves represent the validation accuracy of the best activation function discovered so far, averaged across 100 independent trials, and the shaded areas show the 95% confidence interval around the mean. In all cases, regression with UMAP features outperforms random search, and searching with both eigenvalues and outputs outperforms searching with either feature alone. Of the three regression algorithms, KNR performs the best, rapidly surpassing ReLU and quickly discovering near-optimal activation functions in all benchmark tasks. Thus, the features make it possible to find good activation functions efficiently and reliably even with off-the-shelf search methods; the benchmark datasets make it possible to demonstrate these conclusions with statistical reliability.

A number of conclusions can be drawn from Figure 6.6. First, all search algorithms, even random search, reliably discover activation functions that outperform ReLU. This finding is supported by previous work (reviewed in Chapter 2): Although ReLU is a good activation function that performs well in many different tasks, better performance can be achieved with novel activation functions. Therefore, continuing to use ReLU in the future is unlikely to lead to best results; The choice of the activation function should be an important part of the design, similar to the choice of the network architecture or the selection of its hyperparameters.

Second, all regression algorithms outperform random search. This finding holds across the three types of activation function features and across the three benchmark datasets. The FIM eigenvalues and activation function outputs are thus important in predicting performance of activation functions.

Third, regression algorithms trained on both FIM eigenvalues and activation function outputs outperform algorithms trained on just eigenvalues or outputs alone. This result is consistent across the regression algorithms and benchmark datasets. It suggests that the FIM eigenvalues and activation function outputs contribute complimentary pieces of information. The finding quantitatively reinforces the qualitative visualization in Figure 6.4: FIM eigenvalues are useful for matching activation functions that induce similar training dynamics in neural networks, activation function outputs enable a low-dimensional representation where search is more practical, and combining the two features results in a problem that is more convex and easier to optimize.

Fourth, the searches are efficient. Previous approaches require hundreds or thousands of evaluations to discover good activation functions [15, 17, 160]. In contrast, this chapter

leverages FIM eigenvalues and activation function outputs to reduce the problem to simple two-dimensional regression; the features are powerful enough that out-of-the-box regression algorithms can discover good functions with only tens of evaluations. This efficiency makes it possible to search for better functions directly on large datasets such as ImageNet [41], demonstrated next.

6.7 Searching on New Tasks

The experiments in Section 6.6 utilized precomputed datasets and search spaces to demonstrate that UMAP embeddings are useful in predicting activation function performance, and that KNR can find good functions based on them. In this section, these insights are extended to new datasets and search spaces, demonstrating that AQuaSurF scales up to new and more challenging tasks.

6.7.1 Setup

The experiments were scaled up in two ways. First, while the network architectures were the same, the datasets were much larger and more challenging: All-CNN-C on CIFAR-100, ResNet-56 on CIFAR-100, and MobileViTv2-0.5 on ImageNet. Second, a much larger space with 425,896 unique activation functions was searched, based on four-node computation graphs (Appendix A.4.3). This space is large, diverse, and not precomputed, putting the conclusions from the benchmark experiments to test in a production setting.

Based on the benchmark results, KNR with $k = 3$ was used as the search algorithm. The searches all begin by evaluating the same eight existing activation functions: ELU, ReLU, SELU, sigmoid, Softplus, Softsign, Swish, and tanh. From this starting point, eight

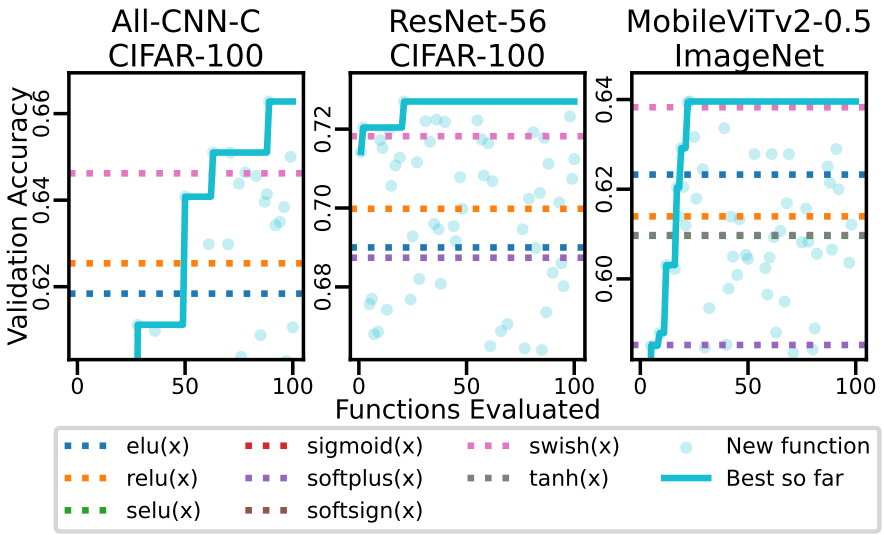


Figure 6.7: Progress of activation function searches. Each point represents the validation accuracy with a unique activation function, and the solid line indicates the performance of the best activation function found so far. AQuaSurF discovers new activation functions that outperform all baseline functions in every case.

workers operated in parallel evaluating the activation functions with the highest predicted performance. Full experimental details are in Appendix A.4.

6.7.2 Results

Figure 6.7 shows that all three searches find improved activation functions over time, and Figure 6.8 shows how the searches navigate the search space. In every experiment, new activation functions were discovered that outperform all baseline functions. Although the search space is large, the searches are efficient, requiring only tens of evaluations to improve performance. Impressively, the search with ResNet-56 on CIFAR-100 produced an activation function that outperformed all baselines on just the second evaluation.

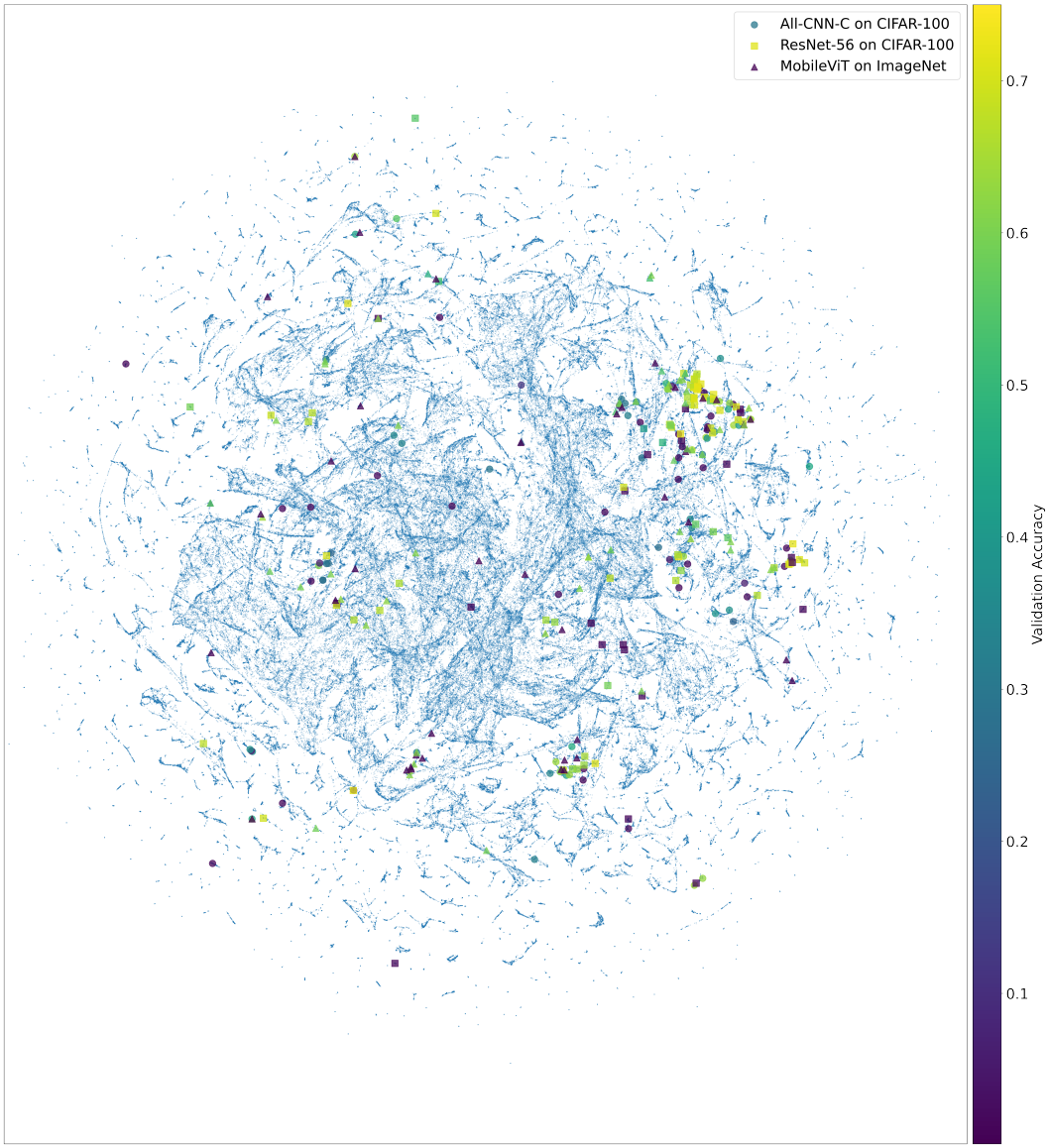


Figure 6.8: Low-dimensional UMAP representation of the 425,896 function search space. The activation functions are embedded according to their outputs; each point represents a unique function. The larger points represent activation functions that were evaluated during the searches; they are colored according to their validation accuracy. Although the space is vast, the searches require only tens of evaluations to discover good activation functions.

Table 6.1: Accuracy with different activation functions. The CIFAR-100 results show the median test accuracy from three runs, and the ImageNet results show the validation accuracy from a single run. AQuaSurF discovers novel activation functions that outperform all baselines in every case. This result demonstrates both that good functions matter, and the power of optimizing them to the task.

All-CNN-C on CIFAR-100		ResNet-56 on CIFAR-100		MobileViTv2-0.5 on ImageNet	
$\text{HardSigmoid}(\text{HardSigmoid}(x)) \cdot \text{ELU}(x)$	**0.6990**	$\text{Swish}(-2x)$	**0.7469**	$-x \cdot \sigma(x) \cdot \text{HardSigmoid}(x)$	**0.6396**
$\sigma(\text{Softsign}(x)) \cdot \text{ELU}(x)$	0.6950	$\text{SELU}(\sinh(e^{\arctan(x)} - 1))$	0.7458	$\text{ELU}(\text{Swish}(-x))$	0.6394
$\text{Swish}(x)/\text{SELU}(1)$	0.6931	$x \cdot \text{erfc}(\text{ELU}(x))$	0.7419	$\text{Swish}(x) \cdot \text{erfc}(\text{bessel_i0e}(x))$	0.6336
ELU	0.6312	ELU	0.7411	ELU	0.6233
ReLU	0.6897	ReLU	0.7348	ReLU	0.6139
SELU	0.0100	SELU	0.6967	SELU	0.6096
sigmoid	0.0100	sigmoid	0.5766	sigmoid	0.5032
Softplus	0.6563	Softplus	0.7397	Softplus	0.5853
Softsign	0.2570	Softsign	0.6624	Softsign	0.5710
Swish	0.6913	Swish	0.7401	Swish	0.6383
tanh	0.3757	tanh	0.6754	tanh	0.6098

Table 6.1 shows the final results from AQuaSurF. The results reinforce the fact that substantial gains can be obtained when using better activation functions than the default ReLU, and especially those optimized specifically for the task. The activation functions can also be transferred to new tasks to improve performance, as shown in Table 6.2.

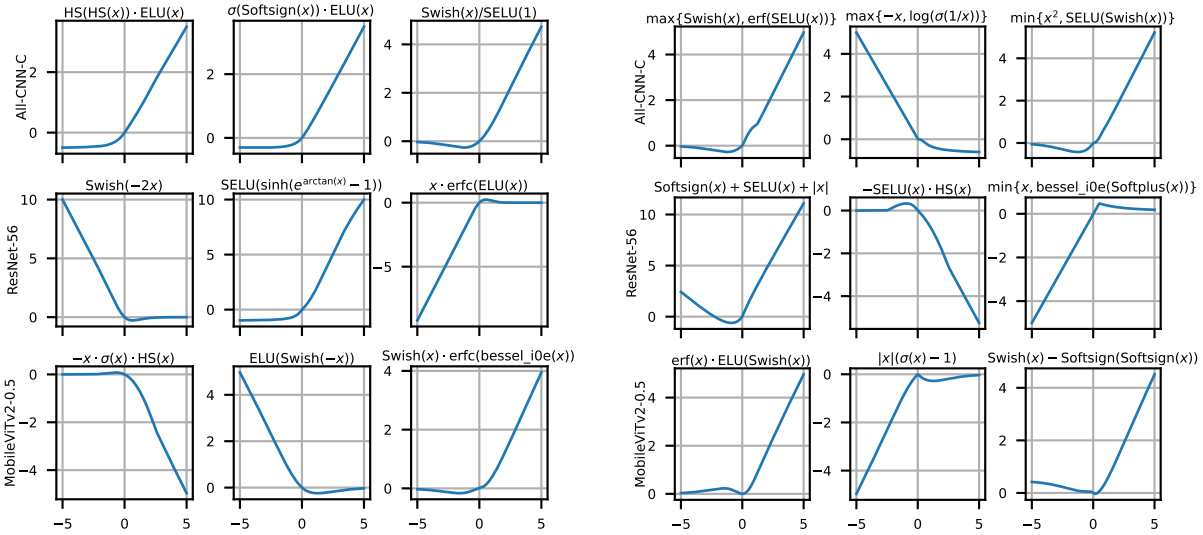
Figure 6.9 illustrates the different activation functions discovered during the searches. Visually, the best functions (shown in 6.9a) are similar to existing functions like ELU and Swish, with subtle changes in their saturation value, the slope of the positive segment, and the width and depth of the negative bump. This result is not surprising since these functions formed the starting point for the search process. Indeed, after a few good functions were discovered, much of the search process focused on refining their design (Figure 6.8). Although these refinements appear small, they were not known ahead of time and they are significant, as evidenced by the final results (Table 6.1).

Table 6.2: ResNet-50 top-1 accuracy on ImageNet. Results are the median of three runs. The best activation functions discovered in the searches (Table 6.1) successfully transfer to this new task, with eight of the nine functions outperforming ReLU.

$-x \cdot \sigma(x) \cdot \text{HardSigmoid}(x)$	**0.7776**
$\text{Swish}(x)/\text{SELU}(1)$	0.7771
$\text{Swish}(x) \cdot \text{erfc}(\text{bessel_i0e}(x))$	0.7755
$\sigma(\text{Softsign}(x)) \cdot \text{ELU}(x)$	0.7734
$\text{SELU}(\sinh(e^{\arctan(x)} - 1))$	0.7719
$\text{HardSigmoid}(\text{HardSigmoid}(x)) \cdot \text{ELU}(x)$	0.7718
$\text{ELU}(\text{Swish}(-x))$	0.7679
$\text{Swish}(-2x)$	0.7664
$x \cdot \text{erfc}(\text{ELU}(x))$	0.7635
$\text{ReLU}(x)$	0.7660

With AQuaSurF, several interesting activation functions were discovered (Figure 6.9b). While they were not the best, all of them outperformed ReLU. These functions have properties uncommon among the usual deep learning activation functions: Many of them have discontinuous derivatives at $x = 0$; some do not saturate, but diverge as $x \to \pm\infty$; in contrast to Swish, which features a negative bump, many of these functions contain positive bumps. In the future, these designs may be refined further, and perhaps produce better activation functions for specific new tasks.

Together, the plots show that AQuaSurF is capable of both exploitation (Figure 6.9a) and exploration (Figure 6.9b). In the future, it will be interesting to consider tradeoffs between these concepts. A more comprehensive discussion of this and other future research directions is in Section 6.8.



(a) (b)

Figure 6.9: Sample activation functions discovered with AQuaSurF. "HS" stands for Hard-Sigmoid. (a) The top three functions discovered in each search. Many of these functions are fine-tuned versions of existing activation functions like ELU and Swish. (b) Selected novel activation functions. All of these functions outperformed ReLU and are distinct from existing activation functions. Such designs may serve as a foundation for further improvement and specialization in new tasks.

6.8 Discussion

This chapter demonstrated that FIM eigenvalues and activation function outputs are efficient and reliable features that can predict performance of activation functions accurately. This finding enabled discovering better activation functions for various tasks, improving the state of the art in machine learning. Because the technique is efficient, it was possible to scale it up to large datasets such as ImageNet. These discoveries inspire several avenues for future research, discussed below.

New Search Spaces The PANGAEA search space was used in this chapter because it is known to work well for deep architectures [15]. In the future it will be interesting to explore search spaces with different unary, binary, and n-ary operators. Beyond computation graphs, it may also be possible to apply techniques in this chapter to optimize continuous vector representations of activation functions [2, 143].

Exploration vs. Exploitation The KNR approach was utilized to search for new activation functions because it performed well on the benchmark datasets (Section 6.6). In the future, it will be interesting to consider other algorithms and analyze their tradeoffs between exploration and exploitation. For example, in a resource-constrained environment where improvement is needed quickly, a more exploitative approach could be used to find an improved activation function in a short time. On the other hand, if substantial compute is available, an approach that focuses on exploration could be used to discover activation functions that perform well but are maximally different from functions used in modern architectures (Figure 6.9b). Novelty search [109] could serve as a suitable approach, and such discoveries could further understanding of how neural networks utilize different kinds of activation functions to learn.

Optimizing Multiple Activation Functions In a typical neural network design, the same activation function is used throughout the network. However, recent work has shown that it may be beneficial to have different activation functions at different locations, and further, that it may be useful to have different activation functions in the early and late stages of training [15]. Indeed, many hybrid architectures use Swish in convolutional layers

and ReLU in attention layers [136]. Unfortunately, it is difficult to design these strategies manually, and so practitioners often use a single activation function for simplicity.

The techniques proposed in this chapter may provide an avenue toward optimizing multiple activation functions in tandem. For example, the features for multiple candidate activation functions could be concatenated into a single feature vector, and this vector could be projected with UMAP to a low-dimensional space where performance prediction is more straightforward.

Optimizing Other Aspects of Neural Network Design By fixing the neural network architecture and varying the activation function, this chapter showed that it is possible to use FIM eigenvalues to infer future performance. As the FIM is a fundamental quantity in machine learning, it may be possible to apply a similar strategy to optimize other aspects of neural network design, such as normalization layers, loss functions, or data augmentation strategies [32, 61, 62, 121]. If a meaningful distance metric between such objects can be defined, then UMAP could be used to map them to a low-dimensional space where performance prediction is much simpler.

Reverse Engineering Activation Functions UMAP was used to project activation functions to a low-dimensional space, and regression algorithms to predict the performance of activation functions in this space, i.e. to serve as a fitness function for the search. However, it is possible that there is no activation function that maps to the optimum of this fitness landscape. Indeed, because such search spaces are finite, the activation functions do not completely fill them. For example, there are empty regions in Figure 6.4, corresponding to

activation functions outside of the predefined search space.

What should be done if an empty region of the embedding space has a higher predicted fitness than any of the candidate activation functions? In the chapter, these regions were simply ignored, and the activation function with the highest predicted fitness was used. However, in the future, it may be possible to create activation functions that map to these empty spaces, an in so doing improve performance. One approach could be based on inverse transforms: Given a coordinate in the low-dimensional embedding space, UMAP can apply an inverse transform and return an object that would have mapped to those coordinates. This technique was already used for visualization in Figure 6.3. Using this approach, UMAP could generate a hypothetical desired FIM eigenvalue distribution, or a list of activation function outputs.

There are two challenges to this approach. First, because UMAP is a dimensionality-reduction algorithm, different activation functions can map to the same location in the embedding space. Thus, the mapping from embedding space back to activation functions is not well defined. Second, even if UMAP prescribes a FIM eigenvalue distribution that is predicted to result in good performance, it may be difficult to manually design an activation function to satisfy that distribution.

However, a generated list of prescribed activation function outputs is already a good start. From this list, it is possible to construct an activation function that interpolates through these points, either in a piecewise linear fashion, with splines, or using some other standard technique. Even without the corresponding FIM eigenvalues, such an approach could potentially improve the efficiency of novel activation function discovery, and lead to better designs for activation functions in the future.

6.9 Conclusion

This chapter introduced three benchmark datasets, `Act-Bench-CNN`, `Act-Bench-ResNet`, and `Act-Bench-ViT`, to support research on activation function optimization. Experiments with these datasets showed that FIM eigenvalues and activation function outputs, and their low-dimensional UMAP embeddings, predict activation function performance accurately, and can thus be used as a surrogate for finding better functions, even with out-of-the-box regression algorithms. These conclusions extended from the benchmark datasets to new and challenging real-world tasks, where better activation functions were discovered on CIFAR-100 and ImageNet. The study reinforces the idea that activation function design is an important part of deep learning, and shows AQuaSurF is an efficient and flexible mechanism for doing it.

Chapter 7

Discussion and Future Work

This chapter reviews the main ideas that were introduced in this dissertation, and includes directions for possible future work.

7.1 Optimizing Other Aspects of Neural Network Design

While much of the literature in AutoML has focused on hyperparameter optimization and neural architecture search (Chapter 2), and this dissertation focused on activation functions (Chapters 3, 4, 6) and weight initialization (Chapter 5), in the future it will be important to similarly optimize other aspects of neural network design. This dissertation showed that automated methods can be more effective and more creative than humans, and it is likely that these benefits extend to other areas as well. Indeed, existing work has shown that automating the design of loss functions [61, 62], learning rate schedules [23], data augmentation strategies [32, 33], and optimizers [28] is promising and worthy of future research.

7.2 General and Specialized Solutions

CAFE and PANGAEA showed that general activation functions that perform well across architectures and tasks exist, but that specialized activation functions designed for

specific architectures and tasks give the best performance. To date, this idea of specialized solutions has been overlooked in AutoML, and it will be worth pursuing in the future. It is likely that similar benefits can be found by discovering specialized versions of other objects like loss functions or data augmentation strategies. Discovering specialized solutions is often more computationally expensive than finding general solutions, so it will also be important to focus on efficient search algorithms. For example, AQuaSurF was much more efficient than CAFE and PANGAEA, and made specialized activation function search practical for large tasks like ImageNet.

7.3 Joint Optimization of Multiple Components

This dissertation automated the design of activation functions and weight initialization strategies, and thus made it possible to understand the effects of optimizing multiple components of neural network design simultaneously. It turns out that doing so is beneficial: Optimizing both the activation function and weight initialization was synergistic, resulting in better performance than optimizing either component alone. It is likely joint optimization of other aspects of neural networks will lead to similar benefits. It is also possible that this practice will lead to new and surprising design choices. For example, simply changing the activation function can inspire designs for a different weight initialization [72], dropout strategy [99], normalization implementation [121], and topology [183]. All of the components in a machine learning system interact in complex ways, and the future challenge will be to exploit these interactions in order to find the combinations that lead to the best performance.

7.4 Computational Cost Considerations

AutoML approaches tend to be computationally expensive because evaluating new designs often requires training neural networks from scratch. These costs will need to be addressed in the future, especially if multiple aspects of neural networks are to be optimized at the same time. In this dissertation, AQuaSurF utilized a surrogate model to improve efficiency over CAFE and PANGAEA by orders of magnitude. AutoInit, on the other hand, identified the principle of stable signal propagation, and used this principle to define a mapping from architectures to initialization strategies. AutoInit was thus efficient by design. By evaluating different neural network designs fairly, AutoInit also accelerated neural architecture search and activation function discovery: processes that are often expensive. In the future, it will be important to focus on efficient search algorithms, low-cost evaluation proxies, surrogate models, and general principles in order to ensure that AutoML is practical to implement.

7.5 Optimizing Alternative Objectives

The techniques in this dissertation used accuracy as the primary evaluation metric, but in some scenarios other objectives may be more appropriate. Possibilities include maximizing adversarial robustness, minimizing inference time, customizing designs to run well on specific hardware, or even a multi-objective combination of these. Future approaches could optimize any of these objectives, and could conceivably present the user with a Pareto front of options that comprise tradeoffs between multiple such objectives.

7.6 Designing Better Representations

Instead of using hand-coded activation function representations like CAFE and PANGAEA, AQuaSurF learned activation function representations in a data-driven way. These representations turned out to be much more informative, and allowed AQuaSurF to discover good activation functions efficiently. This result shows that representations matter in AutoML, and also highlights that a human-interpretable representation may not be the easiest to optimize. It is likely that better encodings can similarly accelerate other AutoML algorithms. Data-driven methods like AQuaSurF are promising ways to find these representations. Although constructing benchmark datasets is initially expensive, it is a cost that is only spent once. It may then result in dramatically more efficient search algorithms that practitioners can implement more easily.

7.7 Towards Artificial General Intelligence

As it stands, AutoML approaches are not capable of recursive self-improvement, but rather execute a single step of improvement. Part of the reason for this limitation is that AutoML algorithms are typically separate from the machine learning system they are optimizing. Thus, they can only optimize the machine learning system to a limited extent before performance plateaus. In order to enable recursive self-improvement in the future, it will be important to take the lessons learned from AutoML and implement them in a system that is self-aware. Large language models (LLMs) may potentially provide a context where such an implementation is possible. There already exists a vast amount of freely available information about machine learning techniques on websites like GitHub, Stack Overflow, and arXiv. An LLM trained on these sources of data and aware of its own code could conceivably

synthesize the current state of the art in machine learning and use this knowledge to improve upon itself repeatedly, thereby achieving recursive self-improvement. Such an ability would be a major step in automated machine learning, and thus also towards artificial general intelligence.

Chapter 8

Conclusion

This chapter summarizes the main contributions of this dissertation, and concludes by reviewing the progress that this dissertation has made towards fully automated machine learning.

8.1 Contributions

Chapter 3 introduced CAFE, which demonstrated that neural network performance can be improved by evolving the design of the activation function. The chapter utilized exhaustive search in a small search space (S_1) and random search and evolution in a larger search space (S_2). It introduced novel activation functions that achieve high accuracy, outperforming both standard functions such as ReLU and novel functions such as Swish. The best activation functions successfully transferred from CIFAR-10 to CIFAR-100 and from WRN-28-10 to WRN-40-4. However, the best results were obtained by specializing functions for each architecture and dataset.

Chapter 4 presented PANGAEA, a system that extended the abilities of CAFE by being more flexible in multiple ways. The search space was extended to include deeper and more complex functional forms, including ones unlikely to be discovered by humans. Instead of fixed activation functions, parametric activation functions were evolved. This construction

allowed the activation functions to take on different shapes at different locations in the network and in different stages of training. PANGAEA used a synergy of two different optimization processes: evolutionary population-based search for the general form and gradient descent-based fine-tuning of the parameters of the activation function. PANGAEA discovered general activation functions that perform well across architectures as well as specialized functions that take advantage of a particular architecture, significantly outperforming previously proposed activation functions in both cases.

Chapter 5 described AutoInit, an algorithm that calculates analytic mean- and variance-preserving weight initialization for neural networks automatically. In convolutional networks, such an initialization improved performance with different activation functions, dropout rates, learning rates, and weight decay settings. In residual networks, AutoInit prevented exploding signals, allowed training with higher learning rates, and improved performance with or without batch normalization. In transformers, AutoInit was scaled up to high-resolution image classification, where it improved performance with several activation functions with and without normalization. AutoInit also improved accuracy on the ImageNet dataset. In neural architecture search, new architectures were evaluated more accurately, resulting in better networks in vision, language, tabular, multi-task, and transfer learning settings. In activation function discovery, AutoInit stabilized training and improved accuracy with a large diversity of novel activation functions. Thus, AutoInit constitutes an important contribution on its own, but it also demonstrates the power of optimizing multiple aspects of neural network design simultaneously.

Chapter 6 introduced AQuaSurF, as well as three benchmark datasets: `Act-Bench-CNN`, `Act-Bench-ResNet`, and `Act-Bench-ViT`. Experiments with these datasets showed that FIM

eigenvalues and activation function outputs, and their low-dimensional UMAP embeddings, predict activation function performance accurately, and can thus be used as a surrogate for finding better functions, even with out-of-the-box regression algorithms. These conclusions extended from the benchmark datasets to new and challenging real-world tasks, where better activation functions were discovered on CIFAR-100 and ImageNet. AQuaSurF demonstrated that learning better representations for activation functions allowed them to be optimized more effectively. The technique reinforces the importance of activation function design, and makes automating such designs practical for machine learning practitioners to implement.

8.2 Big Picture

The goal of AutoML is to replace biased and ad hoc human designs with ones that are creative, principled, more effective, and automatically generated. This dissertation advanced the state of the art in AutoML with automated approaches for activation function design and weight initialization. This dissertation also showed that specialized solutions give better performance than general ones, and demonstrated the power of optimizing multiple aspects of neural network design jointly. These findings will be needed in order to realize the potential of fully automated machine learning, and to achieve AGI.

Appendices

Appendix A

Training Details

This appendix contains training details for the experiments in the main text.

A.1 CAFE

This section describes implementation details for the experiments in Chapter 3.

A.1.1 Training Setup

A wide residual network [208] with depth 28 and widening factor 10 (WRN-28-10), implemented in TensorFlow [1], was trained on the CIFAR-10 and CIFAR-100 image datasets [104]. The architecture was comprised of repeated residual blocks that apply batch normalization and ReLU prior to each convolution. In the experiments, all ReLU activations were replaced with a candidate activation function. No other changes to the architecture were made. Hyperparameters were chosen to mirror those of Zagoruyko and Komodakis [208] as closely as possible. Featurewise center, horizontal flip, and ZCA whitening preprocessing were applied to the datasets. Dropout probability was 0.3, and the architecture was optimized using stochastic gradient descent with Nesterov momentum 0.9. WRN-40-4 (a deeper and thinner wide residual network architecture) was also used in some experiments for comparison.

The CIFAR-10 and CIFAR-100 datasets both have 50K training images, 10K testing

images, and no standard validation set. To prevent overfitting, balanced validation splits were created for both datasets by randomly selecting 500 images per class from the CIFAR-10 training set and 50 images per class from the CIFAR-100 training set. The test set was not modified so that the results can be compared with other work.

A.1.2 Search Implementation

To discover activation functions, a number of search strategies were used. Regardless of the strategy, the training set always consisted of 45K images while the validation set contained 5K images; the test set was never used during the search. All ReLU activations in WRN-28-10 were replaced with a candidate activation function and the architecture was trained for 50 epochs. The initial learning rate was set to 0.1, and decreased by a factor of 0.2 after epochs 25, 40, and 45. Training for only 50 epochs made it possible to evaluate many activation functions without excessive computational cost.

The top three activation functions by validation accuracy from the entire search were returned as a result. For each of these functions, a WRN-28-10 was trained from scratch for 200 epochs. The initial learning rate was set to 0.1, and decreased by a factor of 0.2 after epochs 60, 120, and 160, mirroring the work by Zagoruyko and Komodakis [208]. After training was complete, the test set accuracy was measured. The median test accuracy of five runs was reported as the final result, as is commonly done in similar work in the literature [160].

A.2 PANGAEA

This section describes the implementation details for the experiments in Chapter 4.

A.2.1 Training Details

Wide Residual Network (WRN-10-4) When measuring final performance after evolution, the standard WRN setup was used; all ReLU activations in WRN-10-4 were replaced with the evolved activation function, but no other changes to the architecture were made. The network was optimized using stochastic gradient descent with Nesterov momentum 0.9. The network was trained for 200 epochs; the initial learning rate was 0.1, and it was decreased by a factor of 0.2 after epochs 60, 120, and 160. Dropout probability was set to 0.3, and L2 regularization of 0.0005 was applied to the weights. Data augmentation included featurewise center, featurewise standard deviation normalization, horizontal flip, and random 32×32 crops of images padded with four pixels on all sides. This setup was chosen to mirror the original WRN setup [208] as closely as possible.

During evolution of activation functions, the training was compressed to save time. The network was trained for only 100 epochs; the learning rate began at 0.1 and was decreased by a factor of 0.2 after epochs 30, 60, and 80. Empirically, the accuracy achieved by this shorter schedule was sufficient to guide evolution; the computational cost saved by halving the time required to evaluate an activation function could then be used to search for additional activation functions.

Residual Network (ResNet-v1-56) As with WRN-10-4, when measuring final performance with ResNet-v1-56, the only change to the architecture was replacing the ReLU activations with an evolved activation function. The network was optimized with stochastic gradient descent and momentum 0.9. Dropout was not used, and L2 regularization of 0.0001 was applied to the weights. In the original ResNet experiments [73], an initial learning

rate of 0.01 was used for 400 iterations before increasing it to 0.1, and further decreasing it by a factor of 0.1 after 32K and 48K iterations. An iteration represents a single forward and backward pass over one training batch, while an epoch consists of training over the entire training dataset. In these experiments, the learning rate schedule was implemented by beginning with a learning rate of 0.01 for one epoch, increasing it to 0.1, and then decreasing it by a factor of 0.1 after epochs 91 and 137. (For example, (48K iterations / 45K training images) * batch size of 128 $\approx$ 137.) The network was trained for 200 epochs in total. Data augmentation included a random horizontal flip and random 32×32 crops of images padded with four pixels on all sides, as in the original setup [73].

When evolving activation functions for ResNet-v1-56, the learning rate schedule was again compressed. The network was trained for 100 epochs; the initial warmup learning rate of 0.01 still lasted one epoch, the learning rate increased to 0.1, and then decreased by a factor of 0.1 after epochs 46 and 68. When evolving activation functions, their relative performance is more important than the absolute accuracies they achieve. The shorter training schedule was therefore a cost-efficient way of discovering high-performing activation functions.

Preactivation Residual Network (ResNet-v2-56) The full training setup, data augmentation, and compressed learning rate schedule used during evolution for ResNet-v2-56 were all identical to those for ResNet-v1-56 with one exception: with ResNet-v2-56, it is not necessary to warm up training with an initial learning rate of 0.01 [74], so this step was skipped.

All-CNN-C When measuring final performance with All-CNN-C, the ReLU activation function was replaced with an evolved one, but the setup otherwise mirrored that of Springenberg et al. [174] as closely as possible. The network was optimized with stochastic gradient descent and momentum 0.9. Dropout probability was 0.5, and L2 regularization of 0.001 was applied to the weights. The data augmentation involved featurewise centering and normalizing, random horizontal flips, and random 32×32 crops of images padded with five pixels on all sides. The initial learning rate was set to 0.01, and it was decreased by a factor of 0.1 after epochs 200, 250, and 300. The network was trained for 350 epochs in total.

During evolution of activation functions, the same training setup was used. It is not necessary to compress the learning rate schedule as was done with the residual networks because All-CNN-C trains more quickly.

CIFAR-10 As with CIFAR-100, a balanced validation set was created for CIFAR-10 by randomly selecting 500 images from each class, resulting in a training/validation/test split of 45K/5K/10K images.

A.2.2 Implementing Custom Activation Functions

This section demonstrates how to implement different activation functions in a TensorFlow neural network. For example, the code to create the All-CNN-C architecture with a custom activation function is:

```
def all_cnn_c(args):
    inputs = Input((32, 32, 3))
    x = Conv2D(96, kernel_size=3, strides=(1, 1), padding='same', kernel_regularizer=l2(0.001))(inputs)
    x = CustomActivation(args)(x)
    x = Conv2D(96, kernel_size=3, strides=(1, 1), padding='same', kernel_regularizer=l2(0.001))(x)
    x = CustomActivation(args)(x)
    x = Conv2D(96, kernel_size=3, strides=(2, 2), padding='same', kernel_regularizer=l2(0.001))(x)
```

```
    x = Dropout(0.5)(x)

    x = Conv2D(192, kernel_size=3, strides=(1, 1), padding='same', kernel_regularizer=l2(0.001))(x)
    x = CustomActivation(args)(x)
    x = Conv2D(192, kernel_size=3, strides=(1, 1), padding='same', kernel_regularizer=l2(0.001))(x)
    x = CustomActivation(args)(x)
    x = Conv2D(192, kernel_size=3, strides=(2, 2), padding='same', kernel_regularizer=l2(0.001))(x)
    x = CustomActivation(args)(x)
    x = Dropout(0.5)(x)

    x = Conv2D(192, kernel_size=3, strides=(1, 1), padding='same', kernel_regularizer=l2(0.001))(x)
    x = CustomActivation(args)(x)
    x = Conv2D(192, kernel_size=1, strides=(1, 1), padding='valid', kernel_regularizer=l2(0.001))(x)
    x = CustomActivation(args)(x)
    x = Conv2D(10, kernel_size=1, strides=(1, 1), padding='valid', kernel_regularizer=l2(0.001))(x)
    x = CustomActivation(args)(x)

    x = GlobalAveragePooling2D()(x)
    x = Flatten()(x)
    outputs = Activation('softmax')(x)

    return Model(inputs=inputs, outputs=outputs)
```

The `CustomActivation` is a wrapper that resolves to different activation functions depending on the `args` parameter. After importing `from tensorflow.keras.layers import Activation`, built-in activation functions can be implemented as `Activation('relu')` or `Activation('tanh')`, for example. Activation functions that are not built in and do not contain learnable parameters can be implemented with lambda functions. For example, the Mish activation function can be implemented with `Activation(lambda x : x * tf.math.tanh(tf.keras.activations.softplus(x)))`. Finally, activation functions with learnable parameters simply require subclassing a `Layer` object. The code used to implement the PAU activation function is below; APL and SPLASH are implemented in a similar manner.

```
"""
Padé Activation Units: End-to-end Learning of Flexible Activation Functions in Deep Networks
https://arxiv.org/abs/1907.06732

PAU of degree (5, 4) initialized to approximate Leaky ReLU (0.01)
"""
import tensorflow as tf

from tensorflow.keras.layers import Layer
```

```
from tensorflow.keras.initializers import Constant

class PAU(Layer):
    def __init__(self, num_init=None, denom_init=None, param_shape='per-layer', **kwargs):
        super(PAU, self).__init__(**kwargs)
        self.num_init = num_init if num_init else [0.02979246, 0.61837738, 2.32335207, 3.05202660, 1.48548002, 0.25103717]
        self.denom_init = denom_init if denom_init else [1.14201226, 4.39322834, 0.87154450, 0.34720652]
        self.num_weights = []
        self.denom_weights = []
        self.param_shape = param_shape

    def build(self, input_shape):
        if self.param_shape == 'per-layer':
            param_shape = (1,)
        elif self.param_shape == 'per-channel':
            param_shape = list(input_shape[-1:])
        else:
            assert self.param_shape == 'per-neuron'
            param_shape = list(input_shape[1:])

        for i in range(6):
            self.num_weights.append(
                self.add_weight(
                    name=f'a{i}',
                    shape=param_shape,
                    initializer=Constant(self.num_init[i]),
                    trainable=True))
        for i in range(4):
            self.denom_weights.append(
                self.add_weight(
                    name=f'b{i+1}',
                    shape=param_shape,
                    initializer=Constant(self.denom_init[i]),
                    trainable=True))

    def call(self, inputs):
        num = tf.add_n([self.num_weights[i] * tf.math.pow(inputs, i) for i in range(6)])
        denom = 1 + tf.math.abs(tf.add_n([self.denom_weights[i] * tf.math.pow(inputs, i+1) for i in range(4)]))
        return num / denom

    def get_config(self):
        config = super(PAU, self).get_config()
        config.update({'num_init'    : self.num_init,
                       'denom_init'  : self.denom_init,
                       'param_shape' : self.param_shape})
        return config
```

A.3 AutoInit

This section describes the details of experiments from Chapter 5.

A.3.1 Convolutional, Residual, and Transformer Network Experiment Details

This section contains implementation details for the experiments in Sections 5.5-5.9.

All-CNN-C The training setup follows that of Springenberg et al. [174] as closely as possible. The network was trained with SGD and momentum 0.9. The dropout rate was 0.5 and weight decay as L2 regularization was 0.001. The data augmentation involved featurewise centering and normalizing, random horizontal flips, and random 32×32 crops of images padded with five pixels on all sides. The initial learning rate was 0.01 and was decreased by a factor of 0.1 after epochs 200, 250, and 300 until training ends at epoch 350.

Because Springenberg et al. [174] did not specify how they initialized their weights, the networks were initialized with the "Glorot Uniform" strategy [also called Xavier initialization; 55], where weights were sampled from $\mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}, \frac{\sqrt{6}}{\sqrt{\texttt{fan_in}+\texttt{fan_out}}}\right)$. This initialization is the default setting in TensorFlow,[1] and was sufficient to replicate the results reported by Springenberg et al. [174].

Residual Networks The networks were optimized with SGD and momentum 0.9. Dropout was not used, and weight decay was 0.0001. Data augmentation included a random horizontal flip and random 32×32 crops of images padded with four pixels on all sides.

CoAtNet A smaller variant of the CoAtNet architecture[2] was used in order to fit the model and data on the available GPU memory. The architecture has three convolutional blocks with 64 channels, four convolutional blocks with 128 channels, six transformer blocks with 256 channels, and three transformer blocks with 512 channels. This architecture is slightly deeper

[1] `https://github.com/tensorflow/tensorflow/blob/v2.5.0/tensorflow/python/keras/layers/convolutional.py#L608-L609`

[2] `https://github.com/leondgarse/keras_cv_attention_models/blob/v1.3.0/keras_cv_attention_models/coatnet/coatnet.py#L199`

but thinner than the original CoAtNet-0 architecture, which has two convolutional blocks with 96 channels, three convolutional blocks with 192 channels, five transformer blocks with 384 channels, and two transformer blocks with 768 channels [36]. The models are otherwise identical.

The training hyperparameters were inspired by Wightman et al. [195] and are common in the literature. Specifically, images were resized to 160×160. The learning rate schedule was increased linearly from $1e^{-4}$ to $4e^{-4}$ for six epochs and it then followed a cosine decay until epoch 105. Weight decay was set to 0.02 times the current learning rate at each epoch. The model was trained with batch size 256 and optimized with AdamW [124]. Data augmentation included RandAugment applied twice with a magnitude of six [33]. Mixup and Cutmix were also used with alpha 0.1 and 1.0, respectively [207, 209]. The training images were augmented with random resized crops [181] that were at minimum 8% of the original image; after training the model was evaluated on 95% center crops.

As discussed in Section 5.4, AutoInit maintains signal variance $\nu = 1$, but it is also possible to adjust ν if desired. In the CoAtNet experiments, $\nu = 0.01$ was found to give the best performance among $\nu = \{1, 0.1, 0.01, 0.001\}$. The experiment removing normalization layers used the default of $\nu = 1$.

ImageNet The experiment in Section 5.8 used the same training setup as the experiments with CoAtNet on Imagenette in the previous paragraph except for two changes. The batch size was 2,048 (512 per GPU across four GPUs), and the maximum learning rate was $3.2e^{-2}$.

Data-Dependent Initialization Comparison In the experiments in Section 5.9, a learning rate schedule inspired by superconvergence [170] was used to save time. The learning rate increased linearly to 0.1 during the first five epochs, and then decreased linearly for 20 epochs, after which training ended. The weight decay for All-CNN-C was also decreased by a factor of 10. This modification is common when networks are trained with superconvergence [170].

Initialization Time It is important to note that AutoInit does not incur a significant overhead. Each layer must be visited once to be initialized, so the complexity is $O(L)$ where L is the number of layers. For example: All-CNN-C, ResNet-56, and ResNet-164 took 1, 33, and 106 seconds to initialize. The costs are hardware-dependent, but only spent once, and are small compared to the cost of training.

A.3.2 Neural Architecture Search Experiment Details

This appendix contains implementation details for the experiments in Section 5.10. Table A.1 contains the training hyperparameters, neural network layers, evolutionary hyperparameters, and mutation probabilities used in each of the five tasks. The five tasks used with CoDeepNEAT are:

Vision The MNIST dataset contains 28x28 grayscale images of handwritten digits 0-9. There are 60,000 training images (5,000 of which were used for validation) and 10,000 test images [107]. MNIST was used under the Creative Commons Attribution-Share Alike 3.0 license.

Table A.1: Configuration of neural architecture search experiments in the five tasks. Entries with a (B) or (M) suffix apply to CoDeepNEAT blueprints or modules, respectively. These values were found to work well in preliminary experiments. When AutoInit is applied to an evolved network, it replaces the weight initialization method selected by evolution, but the setup otherwise remains unchanged. The neural architecture search experiments were designed to show that AutoInit improves performance in a wide variety of settings, including those with different data modalities, network topologies, computational complexities, and hyperparameter configurations.

	MNIST	**Omniglot**	**PMLB Adult**	**Wikipedia Toxicity**	**Oxford 102 Flower**
Training Hyperparameters					
Activation	All domains: {ReLU, Linear, ELU, SELU}				
Batch Size	128	1000 iterations	32	128	8
Dropout Rate	$[0.0, 0.7]$	$[0.0, 0.7]$	$[0.0, 0.9]$	$[0.0, 0.5]$	$[0.0, 0.9]$
Epochs	5	3	25	3	30
Filters/Units	[16, 96]	[16, 96]	[64, 192]	[64, 192]	[64, 192]
Kernel Reg.	L2: $[10^{-9}, 10^{-3}]$	L2: $[10^{-9}, 10^{-3}]$	{L1, L2}: $[10^{-9}, 10^{-3}]$	{L1, L2}: $[10^{-9}, 10^{-3}]$	{L1, L2}: $[10^{-7}, 10^{-3}]$
Kernel Size	{1, 3}	{1, 3}	N/A	{1, 3, 5, 7}	N/A
Learning Rate	$[10^{-4}, 10^{-2}]$	$[10^{-4}, 10^{-3}]$	$[10^{-4}, 10^{-2}]$	$[10^{-4}, 10^{-2}]$	$[10^{-4}, 10^{-1}]$
Optimizer	Adam	Adam	Adam	Adam	SGD (Nesterov=0.9)
Weight Init.	All domains: {Glorot Normal, Glorot Uniform, He Normal, He Uniform} or **AutoInit**				
Neural Network Layers					
Add	✓		✓	✓	
Concatenate			✓	✓	
Conv1D				✓	
Conv2D	✓	✓			
Dense			✓		✓
Dropout	✓	✓	✓		✓
GRU				✓	
LSTM				✓	
MaxPooling1D				✓	
MaxPooling2D	✓	✓			
SpatialDropout1D				✓	
WeightedSum		✓			
Evolutionary Hyperparameters					
Elitism (B)	0.4	0.4	0.4	0.2	0.1
Elitism (M)	0.4	0.4	0.4	0.2	0.4
Evaluations (B)	4	4	4	4	1
Generations	30	40	30	30	30
Population Size (B)	22	22	22	22	20
Population Size (M)	56	56	56	56	20
Preserved Networks	12	12	12	12	1
Species (B)	1	1	1	1	1
Species (M)	4	4	4	4	2
Mutation Probabilities					
Change Hyperparam.	0.25	0.25	0.25	0.5	0.5
New Connection (B)	0.12	0.12	0.12	0.2	0.12
New Connection (M)	0.08	0.08	0.08	0.2	0.08
New Layer (M)	0.08	0.08	0.08	0.2	0.08
New Node (B)	0.16	0.16	0.16	0.2	0.16

Language In the Wikipedia Toxicity dataset, the task is to classify English Wikipedia comments as toxic or healthy contributions [199]. The dataset contains 92,835, 31,227, and 30,953 comments in the training, validation, and test sets, respectively.

Tabular In the Adult dataset [101] from the Penn Machine Learning Benchmarks repository [PMLB; 150]) the task is to predict whether an individual makes over $50K per year based on 14 features. Out of 48,842 total instances, 20% were randomly separated to create a test set. The dataset was used under the MIT License.

Multi-Task The Omniglot dataset contains handwritten characters in 50 different alphabets [106]; classifying characters in each alphabet is a natural multi-task problem. The characters are 105×105 grayscale images, and there are 20 instances of each character. To save compute resources, 20 of the 50 alphabets were randomly selected for experiments. A fixed training, validation, and testing split of 50%, 20%, and 30% was used with each task. The learning rate decayed as $\texttt{learning_rate} = 0.1^{\texttt{epoch}/10} * \texttt{initial_learning_rate}$ during training. The dataset was used under the MIT License.

Transfer Learning A DenseNet-121 network was first pretrained on the ImageNet dataset [41, 84]. Models were then evolved to utilize its embeddings to classify images in the Oxford 102 Flower dataset, consisting of 102 types of flowers found in the United Kingdom [148]. Each class has between 40 and 258 images; the training and validation sets have 10 images per class, and the test set contains the remaining images from the dataset. During training, the weight decay (L2 loss) was scaled by the current learning rate. Images were also augmented

to improve generalization performance. Images were randomly flipped horizontally, rotated up to 40 degrees, shifted up/down and left/right up to 20%, and shear intensity and zoom range varied up to 20%.

A.4 AQuaSurF

This section specifies the details for the experiments in Chapter 6.

A.4.1 Implementation Details

Training Details For CIFAR-10 and CIFAR-100, balanced validation sets were created by sampling 5,000 images from the training set. Full training details and hyperparameters are listed in Table A.2.

Search Implementation In order to predict performance for an unevaluated activation function, the function outputs and FIM eigenvalues must first be computed. Thus, the searches in Section 6.7 were implemented in three steps. First, activation function outputs for all 425,896 activation functions in the search space were calculated. This computation is inexpensive and easily parallelizable. Second, eight workers operated in parallel to sample activation functions uniformly at random from the search space and calculate their FIM eigenvalues. Third, once the number of activation functions with FIM eigenvalues calculated reached 5,000, seven of the workers began the search by evaluating the functions with the highest predicted performance. The eighth worker continued calculating FIM eigenvalues for new functions so that their performance could be predicted during the search. This setup allowed taking best advantage of the available compute for the regression-type search

Table A.2: Training details and hyperparameter values used in the experiments in Chapter 6.

All-CNN-C on CIFAR-10 and CIFAR-100	
Batch Size	128
Dropout	0.5
Epochs	25 for `Act-Bench-CNN` and search (Figure 6.7), 50 for full evaluation (Table 6.1)
Image Size	32×32
Learning Rate	Linear warmup to 0.1 for five epochs, then linear decay
Mean/Std. Normalization	Yes
Momentum	0.9
Optimizer	SGD
Random Crops	32×32 crops of images padded with four pixels on all sides
Random Flips	Yes
Weight Decay	$1e^{-4}$
Weight Initialization	AutoInit [16]

ResNet-56 on CIFAR-10 and CIFAR-100	
Batch Size	128
Dropout	0.0
Epochs	25 for `Act-Bench-ResNet` and search (Figure 6.7), 50 for full evaluation (Table 6.1)
Image Size	32×32
Learning Rate	Linear warmup to 0.1 for five epochs, then linear decay
Mean/Std. Normalization	No
Momentum	0.9
Optimizer	SGD
Random Crops	32×32 crops of images padded with five pixels on all sides
Random Flips	Yes
Weight Decay	$1e^{-4}$
Weight Initialization	AutoInit [16]

MobileViTv2-0.5 on Imagenette and ImageNet	
Batch Size	256
CutMix Alpha [207]	1.0
Epochs	105
Evaluation Center Crop	95%
Image Size	160×160
Learning Rate	Linear warmup from $1e^{-4}$ to $4e^{-3}$ for five epochs, then cosine decay to $1e^{-6}$
Mixup Alpha [209]	0.1
Optimizer	AdamW [124]
RandAugment [33]	Magnitude six, applied twice
Random Resized Crop [181]	Minimum 8% of the original image
Weight Decay	$0.02\times$ current learning rate

ResNet-50 on ImageNet	
Batch Size	256
CutMix Alpha [207]	1.0
Epochs	105
Evaluation Center Crop	95%
Image Size	160×160
Learning Rate	Linear warmup from $1e^{-4}$ to $2e^{-3}$ for five epochs, then cosine decay to $1e^{-6}$
Mixup Alpha [209]	0.1
Optimizer	AdamW [124]
RandAugment [33]	Magnitude six, applied twice
Random Resized Crop [181]	Minimum 8% of the original image
Weight Decay	$0.02\times$ current learning rate
Weight Initialization	AutoInit [16]

methods.

The experiments on ImageNet required substantially more compute than the experiments on CIFAR-100. For this reason, all eight workers evaluated activation functions once the number of functions with FIM eigenvalues reached 7,000.

Computing FIM eigenvalues took approximately 26 seconds, 84 seconds, and 37 seconds per activation function for All-CNN-C, ResNet-56, and MobileViTv2-0.5, respectively. This cost is not trivial, but it is well worth it, as the experiments in Chapter 6 show.

Unique Activation Functions Different computation graphs can result in the same activation function (e.g. $\max\{x, 0\}$ and $\max\{0, x\}$). In the benchmark dataset and in the larger search space of Section 6.7, repeated activation functions were filtered out. A total of 1,000 inputs were sampled $\mathcal{N}(0, 1)$ and truncated to $[-5, 5]$. Two activation functions were considered the same if their outputs were identical.

A.4.2 Preliminary Experiments

Several variations to the approach presented in the main text were also evaluated in preliminary experiments. The approach turned out to be robust to most of them, but the results also justify the choices used for the main experiments.

Improving the Combined UMAP Projection Figure A.1 displays a projection of FIM eigenvalues using default UMAP hyperparameters. The plots show the eigenvalues organized in multiple distinct one-dimensional manifolds. Again, FIM eigenvalues are noisy features; there are some clusters of activation functions achieving similar performance, but there are

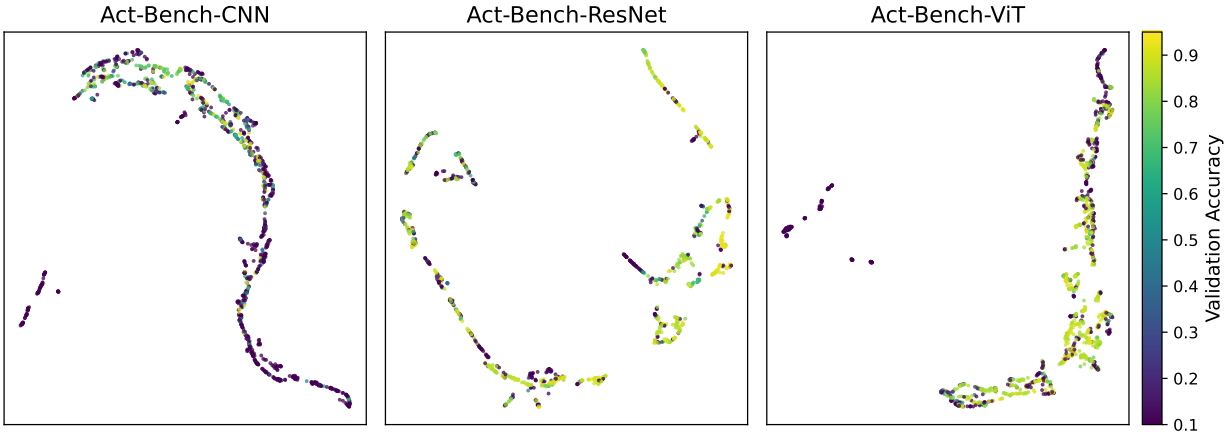


Figure A.1: UMAP projections of FIM eigenvalues using the default hyperparameter of `n_neighbors=15`. The embedding is informative but also noisy. Using `n_neighbors=3`, as shown in the main text, improved performance.

also regions where performance varies widely. As mentioned in the main text, this issue was addressed by reducing the UMAP parameter `n_neighbors` to three. This change reduced the connectivity of the low-dimensional FIM eigenvalue representation, resulting in a space with many distinct clusters (as seen in Figure 6.4).

On its own, this setting did not improve the search on the benchmark datasets. However, it did improve performance when the FIM eigenvalues were combined with activation function outputs (as was discussed in Section 6.4). The reason is that the UMAP model for the activation function outputs did not decrease `n_neighbors`, and so the combined UMAP model relied more on the activation function outputs than it did on the FIM eigenvalues. As Figure 6.4 shows, the activation function outputs are reliable but sometimes project good activation functions to distinct regions in the search space. Introducing extra connectivity into the fuzzy topological representation via the FIM eigenvalues was sufficient to address this issue, bringing good activation functions to common regions of the space.

Increasing the Dimension of the UMAP Projections The UMAP plots show two-dimensional projections of FIM eigenvalues and activation function outputs. Regression algorithms were also trained on five and 10-dimensional projections. These runs resulted in comparable or worse performance. Therefore, the two-dimensional projections were selected for simplicity and for consistency between the algorithm implementation and figure visualizations.

Gaussian Process Regression As an alternative search method, Gaussian process regression (GPR) was evaluated in activation function search. Several different acquisition mechanisms were used, including expected improvement, probability of improvement, maximum predicted value, and upper confidence bound. The approach worked well, but the results were inconsistent across the different acquisition mechanisms. GPR was also more expensive to run compared to the algorithms in the main text (KNR, RFR, SVR), and so those algorithms were used instead for simplicity and efficiency.

Adjusting k in KNR The initial experiments with the KNR algorithm used $k = 3$. Experimenting with $k = \{1, 5, 8\}$ did not reliably improve performance, so $k = 3$ was kept.

Uniformly Spaced Inputs for Activation Function Outputs In an alternative implementation, equally spaced inputs from -5 to 5 were given to the activation functions instead of normally distributed inputs. This variation did not noticeably change the quality of the embeddings nor the performance of the search algorithms. Therefore, normal inputs were used for consistency with Equation 6.17. Figure 6.3 is the only exception; it used 80 inputs

equally spaced from -5 to 5 and increased the UMAP parameter `min_dist` to 0.5. These settings improved the quality of the reconstructed activation functions in the plot.

A.4.3 Activation Function Search Spaces

The activation functions in this chapter were implemented as computation graphs from the PANGAEA search space [15]. The space includes unary and binary operators, in addition to existing activation functions [30, 42, 99, 146, 160]. This approach allows specifying families of functions in a compact manner. It is thus possible to focus the search on a space where good functions are likely to be located, and also to search it comprehensively.

Benchmark Datasets The benchmark datasets introduced in Section 6.1 contain every activation function of the three-node form `binary(unary(`x`),unary(`x`))` using the operators in Table A.3. The result is 5,103 activation functions, of which 2,913 are unique. This space is visualized in Figure 6.4.

For `Act-Bench-CNN` and `Act-Bench-ResNet`, the accuracies are the median from three runs. For `Act-Bench-ViT`, the results are from single runs due to computational costs.

New Tasks The experiments in Section 6.7 utilized a larger search space. Specifically, it was based on the following four-node computation graphs: `binary(unary(unary(`x`)),unary(`x`))`, `binary(unary(`x`),unary(unary(`x`)))`, `n-ary(unary(`x`),unary(`x`),unary(`x`))`, `unary(binary(unary(`x`),unary(`x`)))`, and `unary(unary(unary(unary(`x`))))`. The unary and binary nodes used the operators in Table A.3, and the n-ary node used the sum, product, maximum, and minimum operators. Together, these computation graphs create a search

Table A.3: Activation function search spaces were defined through computation graphs consisting of basic unary and binary operators as well as existing activation functions [15].

Unary			**Binary**
0	$\mathrm{erf}(x)$	$\mathrm{ReLU}(x)$	$x_1 + x_2$
1	$\mathrm{erfc}(x)$	$\mathrm{ELU}(x)$	$x_1 - x_2$
x	$\sinh(x)$	$\mathrm{SELU}(x)$	$x_1 \cdot x_2$
$-x$	$\cosh(x)$	$\mathrm{Swish}(x)$	x_1/x_2
$\lvert x \rvert$	$\tanh(x)$	$\mathrm{Softplus}(x)$	$x_1^{x_2}$
x^{-1}	$\mathrm{arcsinh}(x)$	$\mathrm{Softsign}(x)$	$\max\{x_1, x_2\}$
x^2	$\arctan(x)$	$\mathrm{HardSigmoid}(x)$	$\min\{x_1, x_2\}$
e^x	$e^x - 1$	$\mathrm{bessel_i0e}(x)$	
$\sigma(x)$	$\log(\sigma(x))$	$\mathrm{bessel_i1e}(x)$	

space with 1,023,516 functions, of which 425,896 are unique. This space is visualized in Figure 6.8.

Appendix B

Compute Infrastructure and Cost

This appendix describes the infrastructure used to run the experiments in the main text, and the computational cost incurred from running them.

B.1 Compute Infrastructure

Four main systems were used to run experiments for this dissertation.

HTCondor In the first system, HTCondor [185] was used for scheduling jobs. Jobs were placed on NVIDIA GeForce GTX 1080 and 1080 Ti GPUs.

Slurm The second system used the Slurm workload manager, and also used 1080 and 1080 Ti GPUs.

StudioML In the third system, StudioML software [145, 211] was used to place jobs on machines with NVIDIA GeForce GTX 1080 Ti and RTX 2080 Ti GPUs.

AWS The fourth system was an AWS `g5.48xlarge` instance with eight NVIDIA A10G GPUs. The instance ran in Oregon (`us-west-2`) and was powered by renewable energy, so the experiments with this system contributed no carbon emissions.

B.2 Cost

The overall cost of each system depends on the hardware used. With this detail in mind, the costs for the four systems in this dissertation in terms of GPU hours are estimated below.

B.2.1 CAFE

Ten generations of evolution took approximately 2,000 GPU hours using the HTCondor system.

B.2.2 PANGAEA

PANGAEA experiments used both the HTCondor and Slurm systems. When a job began executing, a parent activation function was selected by sampling $S = 16$ functions from the $P = 64$ most recently evaluated activation functions. This is a minor difference from the original regularized evolution [162], which is based on a strict sliding window of size P. This approach may haven given extra influence to some activation functions, depending on how quickly or slowly jobs were executed in each of the clusters. In practice the method was highly effective; it allowed evolution to progress quickly by taking advantage of extra compute when demand on the clusters was low.

It is difficult to know ahead of time how computationally expensive the evolutionary search will be. Some activation functions immediately resulted in an undefined loss, causing training to end. In that case only a few seconds had been spent and another activation function could immediately be evaluated. Other activation functions trained successfully, but their complicated expressions resulted in longer-than-usual training times. In these

experiments, evolution for WRN-10-4 took 2,314 GPU hours, evolution for ResNet-v1-56 took 1,594 GPU hours, and evolution for ResNet-v2-56 took 2,175 GPU hours. These numbers do not include costs for reranking and repeated runs in the final experiments.

B.2.3 AutoInit

AutoInit experiments were run using the StudioML system, while the CoAtNet experiments (Section 5.7) used the AWS system. Training CoAtNet on Imagenette required an average of 0.91 GPU hours per run. Training CoAtNet on ImageNet once took 119.89 GPU hours. Training on Imagenette instead of ImageNet therefore required $119.89/0.91 \approx 132$ times less compute.

B.2.4 AQuaSurF

AQuaSurF was implemented using the AWS system. The total compute cost for the search experiments in Section 6.7 was 14.49 GPU-hours for All-CNN-C on CIFAR-100, 21.67 GPU-hours for ResNet-56 on CIFAR-100, and 196.25 GPU-days for MobileViTv2-0.5 on ImageNet. This cost includes the time to train the eight baseline activation functions and then to evaluate 100 additional functions.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi. Learning activation functions to improve deep neural networks. *arXiv:1412.6830*, 2014.

[3] M. Alber, I. Bello, B. Zoph, P.-J. Kindermans, P. Ramachandran, and Q. Le. Backprop evolution. *arXiv:1808.02822*, 2018.

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

[5] A. Antoniou, H. Edwards, and A. Storkey. How to train your maml. In *International Conference on Learning Representations*, 2018.

[6] A. Apicella, F. Donnarumma, F. Isgrò, and R. Prevete. A survey on modern trainable activation functions. *Neural Networks*, 2021.

[7] D. Arpit, V. Campos, and Y. Bengio. How to initialize your network? robust initialization for weightnorm & resnets. *Advances in Neural Information Processing Systems*, 32, 2019.

[8] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv:1607.06450*, 2016.

[9] T. Bachlechner, B. P. Majumder, H. Mao, G. Cottrell, and J. McAuley. Rezero is all you need: Fast convergence at large depth. In *Uncertainty in Artificial Intelligence*, pages 1352–1361. PMLR, 2021.

[10] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv:1611.02167*, 2016.

[11] M. Basirat and P. M. Roth. The quest for the golden activation function. In *Proceedings of the ARW & OAGM Workshop*, 2019.

[12] I. Bello, B. Zoph, V. Vasudevan, and Q. V. Le. Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, pages 459–468. PMLR, 2017.

[13] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[14] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[15] G. Bingham and R. Miikkulainen. Discovering parametric activation functions. *Neural Networks*, 148:48–65, 2022.

[16] G. Bingham and R. Miikkulainen. Autoinit: Analytic signal-preserving weight initialization for neural networks. In *Thirty-Seventh AAAI Conference on Artificial Intelligence*, 2023.

[17] G. Bingham, W. Macke, and R. Miikkulainen. Evolutionary optimization of deep learning activation functions. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancún, Mexico*, 2020.

[18] N. Bostrom. Superintelligence: paths, dangers, strategies. 2014.

[19] A. Brock, S. De, and S. L. Smith. Characterizing signal propagation to close the performance gap in unnormalized resnets. *arXiv:2101.08692*, 2021.

[20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[21] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang. Efficient architecture search by network transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[22] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu. Path-level network transformation for efficient architecture search. In *International Conference on Machine Learning*, pages 678–687. PMLR, 2018.

[23] P. Carvalho, N. Lourenço, F. Assunção, and P. Machado. Autolr: an evolutionary approach to learning rate policies. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 672–680, 2020.

[24] F.-C. Chen. Back-propagation neural networks for nonlinear self-tuning adaptive control. *IEEE control systems Magazine*, 10(3):44–48, 1990.

[25] L. Chen, M. Collins, Y. Zhu, G. Papandreou, B. Zoph, F. Schroff, H. Adam, and J. Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *Advances in Neural Information Processing Systems*, pages 8699–8710, 2018.

[26] M. Chen, J. Pennington, and S. Schoenholz. Dynamical isometry and a mean field theory of rnns: Gating enables signal propagation in recurrent neural networks. In *International Conference on Machine Learning*, pages 873–882. PMLR, 2018.

[27] X. Chen, L. Xie, J. Wu, and Q. Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1294–1303, 2019.

[28] X. Chen, C. Liang, D. Huang, E. Real, K. Wang, Y. Liu, H. Pham, X. Dong, T. Luong, C.-J. Hsieh, et al. Symbolic discovery of optimization algorithms. *arXiv:2302.06675*, 2023.

[29] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[30] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289, 2015.

[31] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Neural information processing systems*, pages 3123–3131, 2015.

[32] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation policies from data. *arXiv:1805.09501*, 2018.

[33] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 702–703, 2020.

[34] X. Cui, W. Zhang, Z. Tüske, and M. Picheny. Evolutionary stochastic gradient descent for optimization of deep neural networks. In *Advances in neural information processing systems*, pages 6048–6058, 2018.

[35] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[36] Z. Dai, H. Liu, Q. V. Le, and M. Tan. Coatnet: Marrying convolution and attention for all data sizes. *Advances in Neural Information Processing Systems*, 34:3965–3977, 2021.

[37] Y. N. Dauphin and S. Schoenholz. Metainit: Initializing learning by learning to initialize. *Advances in Neural Information Processing Systems*, 32, 2019.

[38] S. De and S. Smith. Batch normalization biases residual blocks towards the identity function in deep networks. *Neural Information Processing Systems*, 33, 2020.

[39] A. Defazio and K. Mishchenko. Learning-rate-free learning by d-adaptation. *arXiv preprint arXiv:2301.07733*, 2023.

[40] G. Denevi, C. Ciliberto, D. Stamos, and M. Pontil. Learning to learn around a common mean. In *Advances in Neural Information Processing Systems*, pages 10169–10179, 2018.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[42] S. Elfwing, E. Uchibe, and K. Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, 107:3–11, 2018.

[43] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[44] A. F. Emery and A. V. Nenarokomov. Optimal experiment design. *Measurement Science and Technology*, 9(6):864, 1998.

[45] T. Epps and K. J. Singleton. An omnibus test for the two-sample problem using the empirical characteristic function. *Journal of Statistical Computation and Simulation*, 26(3-4):177–203, 1986.

[46] M. Feurer and F. Hutter. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer, Cham, 2019.

[47] M. Feurer, J. T. Springenberg, and F. Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[48] C. Finn and S. Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. *arXiv:1710.11622*, 2017.

[49] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[50] L. Franceschi, P. Frasconi, S. Salzo, R. Grazzi, and M. Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *ICML*, volume 80, pages 1563–1572. PMLR (Proceedings of Machine Learning Research), 2018.

[51] Y. Furusho and K. Ikeda. Effects of skip-connection in resnet and batch-normalization on fisher information matrix. In *INNS Big Data and Deep Learning Conference*, pages 341–348. Springer, 2019.

[52] Y. Furusho and K. Ikeda. Theoretical analysis of skip connections and batch normalization from generalization and optimization perspectives. *APSIPA Transactions on Signal and Information Processing*, 9, 2020.

[53] Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu. Graphnas: Graph neural architecture search with reinforcement learning. *arXiv:1904.09981*, 2019.

[54] D. Gilboa, B. Chang, M. Chen, G. Yang, S. S. Schoenholz, H. Chi, and J. Pennington. The dynamics of signal propagation in gated recurrent neural networks. 2019.

[55] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[56] L. B. Godfrey. An evaluation of parametric activation functions for deep learning. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 3006–3011. IEEE, 2019.

[57] L. B. Godfrey and M. S. Gashler. A parameterized activation function for learning fuzzy logic operations in deep neural networks. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 740–745. IEEE, 2017.

[58] F. Gomez and R. Miikkulainen. Active guidance for a finless rocket using neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 2084–2095, 2003.

[59] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May): 937–965, 2008.

[60] X. Gong, S. Chang, Y. Jiang, and Z. Wang. Autogan: Neural architecture search for generative adversarial networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3224–3234, 2019.

[61] S. Gonzalez and R. Miikkulainen. Evolving loss functions with multivariate taylor polynomial parameterizations. *arXiv:2002.00059*, 2020.

[62] S. Gonzalez and R. Miikkulainen. Improved training speed, accuracy, and data utilization through loss function optimization. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2020. doi: 10.1109/CEC48606.2020.9185777.

[63] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Neural information processing systems*, 27, 2014.

[64] M. Goyal, R. Goyal, and B. Lall. Learning activation functions: A new paradigm of understanding neural networks. *arXiv:1906.09529*, 2019.

[65] R. Grosse and J. Martens. A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582. PMLR, 2016.

[66] A. Hagg, M. Mensing, and A. Asteroth. Evolving parsimonious networks by mixing activation functions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 425–432, 2017.

[67] B. Hanin. Which neural net architectures give rise to exploding and vanishing gradients? In *32nd International Conference on Neural Information Processing Systems*, pages 580–589, 2018.

[68] B. Hanin and D. Rolnick. How to start training: The effect of initialization and architecture. *Advances in Neural Information Processing Systems*, 31, 2018.

[69] B. Hanin and D. Rolnick. Deep relu networks have surprisingly few activation patterns. In *Advances in Neural Information Processing Systems*, pages 361–370, 2019.

[70] C. R. Harris, K. J. Millman, S. J. van der Walt, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

[71] T. Hayase and R. Karakida. The spectrum of fisher information of deep networks achieving dynamical isometry. In *International Conference on Artificial Intelligence and Statistics*, pages 334–342. PMLR, 2021.

[72] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[73] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[74] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[75] D. Hendrycks and K. Gimpel. Adjusting for dropout variance in batch normalization and weight initialization. *arXiv:1607.02488*, 2016.

[76] D. Hendrycks and K. Gimpel. Gaussian error linear units (gelus). *arXiv:1606.08415*, 2016.

[77] S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

[78] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

[79] S. Hochreiter, A. S. Younger, and P. R. Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[80] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):5149–5169, 2021.

[81] R. Houthooft, Y. Chen, P. Isola, B. Stadie, F. Wolski, O. J. Ho, and P. Abbeel. Evolved policy gradients. In *Advances in Neural Information Processing Systems*, pages 5400–5409, 2018.

[82] J. Howard. Imagenette. `https://github.com/fastai/imagenette`, 2019. Accessed: 2022-07-27.

[83] W. Hu, L. Xiao, and J. Pennington. Provable benefit of orthogonal initialization in optimizing deep linear networks. *arXiv:2001.05992*, 2020.

[84] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[85] L. Huang, X. Liu, Y. Liu, B. Lang, and D. Tao. Centered weight normalization in accelerating training of deep neural networks. In *IEEE International Conference on Computer Vision*, pages 2803–2811, 2017.

[86] L. Huang, J. Qin, L. Liu, F. Zhu, and L. Shao. Layer-wise conditioning analysis in exploring the learning dynamics of dnns. In *European Conference on Computer Vision*, pages 384–401. Springer, 2020.

[87] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[88] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[89] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, et al. Population based training of neural networks. *arXiv:1711.09846*, 2017.

[90] S. Jadon and A. A. Srinivasan. Improving siamese networks for one-shot learning using kernel-based activation functions. In *Data Management, Analytics and Innovation: Proceedings of ICDMAI 2020, Volume 2*, pages 353–367. Springer, 2021.

[91] S. Jastrzebski, Z. Kenton, D. Arpit, N. Ballas, A. Fischer, Y. Bengio, and A. Storkey. Three factors influencing minima in sgd. *arXiv:1711.04623*, 2017.

[92] S. Jastrzebski, D. Arpit, O. Astrand, G. B. Kerg, H. Wang, C. Xiong, R. Socher, K. Cho, and K. J. Geras. Catastrophic fisher explosion: Early phase fisher matrix impacts generalization. In *International Conference on Machine Learning*, pages 4772–4784. PMLR, 2021.

[93] R. Karakida, S. Akaho, and S.-i. Amari. Universal statistics of fisher information in deep neural networks: Mean field approach. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1032–1041. PMLR, 2019.

[94] R. Karakida, S. Akaho, and S.-i. Amari. Pathological spectra of the fisher information metric and its variants in deep neural networks. *Neural Computation*, 33(8):2274–2307, 2021.

[95] B. Karlik and A. V. Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

[96] J. D. M.-W. C. Kenton and L. K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186, 2019.

[97] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[98] D. P. Kingma and P. Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *Advances in neural information processing systems*, 31, 2018.

[99] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[100] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536. PMLR, 2017.

[101] R. Kohavi. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *Kdd*, volume 96, pages 202–207, 1996.

[102] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[103] P. Krähenbühl, C. Doersch, J. Donahue, and T. Darrell. Data-dependent initializations of convolutional neural networks. *arXiv:1511.06856*, 2015.

[104] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[105] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

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

[107] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[108] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

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

[110] J. Lehman, J. Clune, D. Misevic, C. Adami, L. Altenberg, J. Beaulieu, P. J. Bentley, S. Bernard, G. Beslon, D. M. Bryson, et al. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *Artificial life*, 26(2):274–306, 2020.

[111] L. Li and A. Talwalkar. Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, pages 367–377. PMLR, 2020.

[112] Y. Li, C. Wei, and T. Ma. Towards explaining the regularization effect of initial large learning rate in training neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[113] Z. Li, F. Zhou, F. Chen, and H. Li. Meta-sgd: Learning to learn quickly for few-shot learning. *arXiv:1707.09835*, 2017.

[114] J. Liang, E. Meyerson, B. Hodjat, D. Fink, K. Mutch, and R. Miikkulainen. Evolutionary neural automl for deep learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 401–409, 2019.

[115] J. Liang, S. Gonzalez, and R. Miikkulainen. Population-based training for loss function optimization. *arXiv:2002.04225*, 2020.

[116] T. Liang, T. Poggio, A. Rakhlin, and J. Stokes. Fisher-rao metric, geometry, and complexity of neural networks. In *The 22nd international conference on artificial intelligence and statistics*, pages 888–896. PMLR, 2019.

[117] Z. Liao, T. Drummond, I. Reid, and G. Carneiro. Approximate fisher information matrix to characterize the training of deep neural networks. *IEEE transactions on pattern analysis and machine intelligence*, 42(1):15–26, 2018.

[118] S. Lim, I. Kim, T. Kim, C. Kim, and S. Kim. Fast autoaugment. In *Advances in Neural Information Processing Systems*, pages 6665–6675, 2019.

[119] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv:1711.00436*, 2017.

[120] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. In *International Conference on Learning Representations*, 2018.

[121] H. Liu, A. Brock, K. Simonyan, and Q. Le. Evolving normalization-activation layers. *Advances in Neural Information Processing Systems*, 33:13539–13550, 2020.

[122] I. Loshchilov and F. Hutter. Cma-es for hyperparameter optimization of deep neural networks. *arXiv:1604.07269*, 2016.

[123] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv:1608.03983*, 2016.

[124] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv:1711.05101*, 2017.

[125] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[126] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. In *Advances in neural information processing systems*, pages 7816–7827, 2018.

[127] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, page 3, 2013.

[128] D. Maclaurin, D. Duvenaud, and R. Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[129] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[130] A. Marchisio, M. Hanif, S. Rehman, M. Martina, and M. Shafique. A methodology for automatic selection of activation functions to design hybrid deep neural networks. *arXiv:1811.03980*, 2018.

[131] J. Martens and R. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[132] J. Martens, J. Ba, and M. Johnson. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018.

[133] A. D. Martinez, J. Del Ser, E. Villar-Rodriguez, E. Osaba, J. Poyatos, S. Tabik, D. Molina, and F. Herrera. Lights and shadows in evolutionary deep learning: Taxonomy, critical methodological analysis, cases of study, learned lessons, recommendations and challenges. *Information Fusion*, 2020.

[134] F. J. Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[135] L. McInnes, J. Healy, N. Saul, and L. Großberger. Umap: Uniform manifold approximation and projection. *Journal of Open Source Software*, 3(29):861, 2018.

[136] S. Mehta and M. Rastegari. Separable self-attention for mobile vision transformers. *arXiv:2206.02680*, 2022.

[137] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley. Neural architecture search without training. In *International Conference on Machine Learning*, pages 7588–7598. PMLR, 2021.

[138] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[139] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv:1301.3781*, 2013.

[140] D. Mishkin and J. Matas. All you need is a good init. *arXiv:1511.06422*, 2015.

[141] N. Mishra, M. Rohaninejad, X. Chen, and P. Abbeel. A simple neural attentive meta-learner. In *International Conference on Learning Representations*, 2018.

[142] D. Misra. Mish: A self regularized non-monotonic neural activation function. *arXiv:1908.08681*, 2019.

[143] A. Molina, P. Schramowski, and K. Kersting. Padé activation units: End-to-end learning of flexible activation functions in deep networks. *arXiv:1907.06732*, 2019.

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

[145] K. Mutch. Studio go runner. `https://hub.docker.com/repository/docker/leafai/studio-go-runner`, 2017–2021. URL `https://github.com/leaf-ai/studio-go-runner/tree/0.14.2`.

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

[147] R. Negrinho and G. Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv:1704.08792*, 2017.

[148] M.-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *The Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.

[149] C. E. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: comparison of trends in practice and research for deep learning. In *2nd International Conference on Computational Sciences and Technology*, pages 124–133, 2021.

[150] R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, and J. H. Moore. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, Dec 2017. ISSN 1756-0381. doi: 10.1186/s13040-017-0154-4. URL `https://doi.org/10.1186/s13040-017-0154-4`.

[151] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Neural information processing systems*, 32:8026–8037, 2019.

[152] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[153] J. Pennington and P. Worah. The spectrum of the fisher information matrix of a single-hidden-layer neural network. In *NeurIPS*, pages 5415–5424, 2018.

[154] J. Pennington and P. Worah. Nonlinear random matrix theory for deep learning. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124005, 2019.

[155] J. Pennington, S. S. Schoenholz, and S. Ganguli. Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. In *31st International Conference on Neural Information Processing Systems*, pages 4788–4798, 2017.

[156] R. Piessens, E. de Doncker-Kapenga, C. W. Überhuber, and D. K. Kahaner. *QUADPACK: A subroutine package for automatic integration*, volume 1. Springer Science & Business Media, 2012.

[157] M. A. Potter and K. A. D. Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary computation*, 8(1):1–29, 2000.

[158] S. Qiao, H. Wang, C. Liu, W. Shen, and A. Yuille. Micro-batch training with batch-channel normalization and weight standardization. *arXiv:1903.10520*, 2019.

[159] I. Radosavovic, J. Johnson, S. Xie, W.-Y. Lo, and P. Dollár. On network design spaces for visual recognition. In *IEEE/CVF International Conference on Computer Vision*, pages 1882–1890, 2019.

[160] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, 2018.

[161] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org, 2017.

[162] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[163] E. Real, C. Liang, D. So, and Q. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*, pages 8007–8019. PMLR, 2020.

[164] S. Risi, J. Lehman, and K. O. Stanley. Evolving the placement and density of neurons in the hyperneat substrate. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 563–570, 2010.

[165] T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to

accelerate training of deep neural networks. *Neural information processing systems*, 29: 901–909, 2016.

[166] G. Salton, A. Wong, and C.-S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[167] A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv:1312.6120*, 2013.

[168] Y. Shen, Y. Li, J. Zheng, W. Zhang, P. Yao, J. Li, S. Yang, J. Liu, and B. Cui. Proxybo: Accelerating neural architecture search via bayesian optimization with zero-cost proxies. *arXiv:2110.10423*, 2021.

[169] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33:7462–7473, 2020.

[170] L. N. Smith and N. Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial intelligence and machine learning for multi-domain operations applications*, volume 11006, page 1100612. International Society for Optics and Photonics, 2019.

[171] S. L. Smith and Q. V. Le. A bayesian perspective on generalization and stochastic gradient descent. In *International Conference on Learning Representations*, 2018.

[172] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018.

[173] D. So, Q. Le, and C. Liang. The evolved transformer. In *International conference on machine learning*, pages 5877–5886. PMLR, 2019.

[174] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. In *ICLR (workshop track)*, 2015.

[175] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

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

[177] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[178] M. Suganuma, S. Shirakawa, and T. Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the genetic and evolutionary computation conference*, pages 497–504, 2017.

[179] M. Suganuma, M. Ozay, and T. Okatani. Exploiting the potential of standard convolutional autoencoders for image restoration by evolutionary search. In *International Conference on Machine Learning*, pages 4771–4780. PMLR, 2018.

[180] R. Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13(1), 2019.

[181] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[182] M. Taki. Deep residual networks and weight initialization. *arXiv:1709.02956*, 2017.

[183] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019.

[184] M. Tavakoli, F. Agostinelli, and P. Baldi. Splash: Learnable activation functions for improving accuracy and adversarial robustness. *Neural Networks*, 140:1–12, 2021.

[185] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4): 323–356, 2005.

[186] S. Urban. *Neural Network Architectures and Activation Functions: A Gaussian Process Approach*. PhD thesis, Technische Universität München, 2018.

[187] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[188] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Neural information processing systems*, pages 5998–6008, 2017.

[189] K. Vijayaprabakaran and K. Sathiyamurthy. Towards activation function search for long short-term model network: A differential evolution based approach. *Journal of King Saud University-Computer and Information Sciences*, 2020.

[190] P. Virtanen, R. Gommers, T. E. Oliphant, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[191] L. Wang, L. Xie, K. Zhao, J. Guo, and Q. Tian. Scalable nas with factorizable architectural parameters. *Neurocomputing*, 506:25–36, 2022.

[192] C. White, A. Zela, R. Ru, Y. Liu, and F. Hutter. How powerful are performance predictors in neural architecture search? *Advances in Neural Information Processing Systems*, 34:28454–28469, 2021.

[193] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[194] D. Whitley, K. Mathias, and P. Fitzhorn. Delta-Coding: An iterative search strategy for genetic algorithms. In *Proceedings of the International Conference on Genetic Algorithms*, pages 77–84, 1991.

[195] R. Wightman, H. Touvron, and H. Jégou. Resnet strikes back: An improved training procedure in timm. *arXiv:2110.00476*, 2021.

[196] M. Wistuba. Deep learning architecture search by neuro-cell-based evolution with function-preserving mutations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 243–258. Springer, 2018.

[197] M. Wistuba, A. Rawat, and T. Pedapati. A survey on neural architecture search. *arXiv:1905.01392*, 2019.

[198] Y. Wu and K. He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[199] E. Wulczyn, N. Thain, and L. Dixon. Ex machina: Personal attacks seen at scale. In *26th international conference on world wide web*, pages 1391–1399, 2017.

[200] L. Xiao, Y. Bahri, J. Sohl-Dickstein, S. Schoenholz, and J. Pennington. Dynamical isometry and a mean field theory of cnns: How to train 10,000-layer vanilla convolutional neural networks. In *International Conference on Machine Learning*, pages 5393–5402. PMLR, 2018.

[201] C. Xie, M. Tan, B. Gong, A. Yuille, and Q. V. Le. Smooth adversarial training. *arXiv:2006.14536*, 2020.

[202] L. Xie and A. Yuille. Genetic cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1379–1388, 2017.

[203] L. Yang and A. Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.

[204] Z. Yang, K. Tang, and X. Yao. Large scale evolutionary optimization using cooperative coevolution. *Information sciences*, 178(15):2985–2999, 2008.

[205] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE transactions on neural networks*, 8(3):694–713, 1997.

[206] E. Yudkowsky. Levels of organization in general intelligence. *Artificial general intelligence*, pages 389–501, 2007.

[207] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6023–6032, 2019.

[208] S. Zagoruyko and N. Komodakis. Wide residual networks. In *British Machine Vision Conference 2016*. British Machine Vision Association, 2016.

[209] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv:1710.09412*, 2017.

[210] H. Zhang, Y. N. Dauphin, and T. Ma. Fixup initialization: Residual learning without normalization. *arXiv:1901.09321*, 2019.

[211] P. Zhokhov, A. Denissov, and K. Mutch. Studio. `https://studio.ml`, 2017–2021. URL `https://github.com/studioml/studio/tree/0.0.48`.

[212] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2423–2432, 2018.

[213] C. Zhu, R. Ni, Z. Xu, K. Kong, W. R. Huang, and T. Goldstein. Gradinit: Learning to initialize neural networks for stable and efficient training. *Advances in Neural Information Processing Systems*, 34:16410–16422, 2021.

[214] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016.

[215] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

Vita

Garrett Joseph Bingham was born in Provo, Utah in 1996. He received the Bachelor of Science degree in Computer Science & Mathematics from Yale University in May 2019 and enrolled at the University of Texas at Austin in August 2019.

Permanent address: binghamgarrett@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.