Discovering Parametric Activation Functions

Garrett Bingham[a,b,*], Risto Miikkulainen[a,b]

[a] *The University of Texas at Austin, Austin, Texas, 78712, USA*
[b] *Cognizant AI Labs, 649 Front St., San Francisco, California, 94111, USA*

Abstract

Recent studies have shown that the choice of activation function can significantly affect the performance of deep learning networks. However, the benefits of novel activation functions have been inconsistent and task dependent, and therefore the rectified linear unit (ReLU) is still the most commonly used. This paper proposes a technique for customizing activation functions automatically, resulting in reliable improvements in performance. Evolutionary search is used to discover the general form of the function, and gradient descent to optimize its parameters for different parts of the network and over the learning process. Experiments with four different neural network architectures on the CIFAR-10 and CIFAR-100 image classification datasets show that this approach is effective. It discovers both general activation functions and specialized functions for different architectures, consistently improving accuracy over ReLU and other activation functions by significant margins. The approach can therefore be used as an automated optimization step in applying deep learning to new tasks.

Keywords: Activation Functions, Evolutionary Computation, Gradient Descent, AutoML, Deep Learning

1. Introduction

The rectified linear unit ($\text{ReLU}(x) = \max\{x, 0\}$) is the most commonly used activation function in modern deep learning architectures [28]. When introduced, it offered substantial improvements over the previously popular tanh and sigmoid activation functions. Because ReLU is unbounded as $x \to \infty$, it is less susceptible to vanishing gradients than tanh and sigmoid are. It is also simple to calculate, which leads to faster training times.

Activation function design continues to be an active area of research, and a number of novel activation functions have been introduced since ReLU, each with different properties [29]. In certain settings, these novel activation functions lead to substantial improvements in accuracy over ReLU, but the gains are often inconsistent across tasks. Because of this inconsistency, ReLU is still the most commonly used: it is reliable, even though it may be suboptimal.

The improvements and inconsistencies are due to a gradually evolving understanding of what makes an activation function effective. For example, Leaky ReLU [25] allows a small amount of gradient information to flow when the input is negative. It was introduced to prevent ReLU from

[*] Corresponding author
Email addresses: `bingham@cs.utexas.edu` (Garrett Bingham), `risto@cs.utexas.edu` (Risto Miikkulainen)

creating dead neurons, i.e. those that are stuck at always outputting zero. On the other hand, the ELU activation function [5] contains a negative saturation regime to control the forward propagated variance. These two very different activation functions have seemingly contradicting properties, yet each has proven more effective than ReLU in various tasks.

There are also often complex interactions between an activation function and other neural network design choices, adding to the difficulty of selecting an appropriate activation function for a given task. For example, Ramachandran et al. [30] warned that the scale parameter in batch normalization [18] should be set when training with the Swish activation function; Hendrycks and Gimpel [16] suggested using an optimizer with momentum when using GELU; Klambauer et al. [19] introduced a modification of dropout [17] called alpha dropout to be used with SELU. These results suggest that significant gains are possible by designing the activation function properly for a network and task, but that it is difficult to do so manually.

This paper presents an approach to automatic activation function design. The approach is inspired by genetic programming [20], which describes techniques for evolving computer programs to solve a particular task. In contrast with previous studies [3, 4, 24, 30], this paper focuses on automatically discovering activation functions that are parametric. Evolution discovers the general form of the function, while gradient descent optimizes the parameters of the function during training. The approach, called PANGAEA (Parametric ActivatioN functions Generated Automatically by an Evolutionary Algorithm), 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.

2. Related Work

Prior work in automatic activation function discovery includes approaches based on either reinforcement learning (RL), evolutionary computation, or gradient descent. In contrast, PANGAEA combines evolutionary computation and gradient descent into a single optimization process. PANGAEA achieves better performance than previous work, and therefore is a promising approach.

2.1. Reinforcement Learning

Ramachandran et al. [30] 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.

2.2. Evolutionary Computation

Bingham et al. [4] used evolution to discover novel activation functions. Whereas their functions had a fixed graph structure, PANGAEA utilizes a flexible search space that implements activation functions as arbitrary computation graphs. PANGAEA also includes more powerful mutation operations, and a function parameterization approach that makes it possible to further refine functions through gradient descent.

Liu et al. [24] evolved normalization-activation layers. They searched for a computation graph that replaced both batch normalization and ReLU in multiple neural networks. They argued that the inherent nonlinearity of the discovered layers precluded the need for any explicit activation function. However, experiments in this paper show that carefully designed parametric activation functions can in fact be a powerful augmentation to existing deep learning models.

Basirat and Roth [3] used a genetic algorithm to discover task-specific piecewise activation functions. They showed that different functions are optimal for different tasks. However, the discovered activation functions did not outperform ELiSH and HardELiSH, two hand-designed activation functions proposed in the same paper [3]. The larger search space in PANGAEA affords evolution extra flexibility in designing activation functions, while the trainable parameters give customizability to the network itself, leading to consistent, significant improvement.

2.3. Gradient Descent

Learnable activation functions (LAFs) encode functions with general functional forms such as polynomial, rational, or piecewise linear, and utilize gradient descent to discover optimal parameterizations during training [2, 12, 27, 35]. The general forms allow most LAFs to approximate arbitrary continuous functions. In theory, a LAF could approximate any activation function discovered with PANGAEA. However, just because a LAF can represent an activation function does not guarantee that the optimal activation function will be discovered by gradient descent. Indeed, by synergizing evolutionary search and gradient descent, PANGAEA is able to outperform existing LAF approaches.

3. 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.

3.1. Representing and Modifying Activation Functions

Activation functions are represented as computation graphs in which each node is a unary or a binary operator (Table 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 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 Appendix E).

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 1. Both forms are equally likely, and the unary and binary operators are also selected uniformly at random. The computation graphs in Figure 1 represent the simplest non-trivial computation graphs with and without a binary operator. This design choice is inspired by previous work in neuroevolution, which demonstrated the power of starting from simple structures and gradually complexifying them [33].

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.

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

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

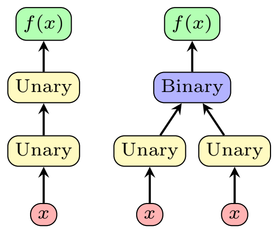

Figure 1: Random activation function initialization. The initial population consists of random samples of two kinds of computation graphs, randomly initialized with the operators in Table 1. In this manner, the search starts with simple graphs and gradually expands to more complex forms.

Given a parent activation function, a child activation function is created by applying one of four possible mutations (Figure 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 paper. 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.

3.1.1. 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 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.

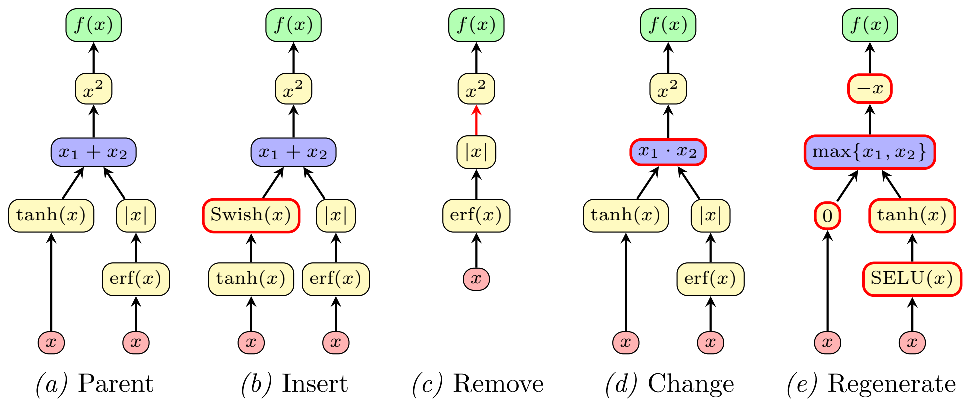

Figure 2: Evolutionary operations on activation functions. In an 'Insert' mutation, a new operator is inserted in one of the edges of the computation graph, like the Swish(x) in *(b)*. In a 'Remove' mutation, a node in the computation graph is deleted, like the addition in *(c)*. In a 'Change' mutation, 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.

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

3.1.3. 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 2*d* shows how changing addition to multiplication produces the activation function $(\tanh(x) \cdot |\mathrm{erf}(x)|)^2$.

3.1.4. 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 [9, 37]. Figure 2*e* shows the child activation function $-\max\{0, \tanh(\mathrm{SELU}(x))\}$, which is quite different from the parent function in Figure 2*a*.

3.1.5. Parameterization of Activation Functions

After mutation (or random initialization), activation functions are parameterized (Figure 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

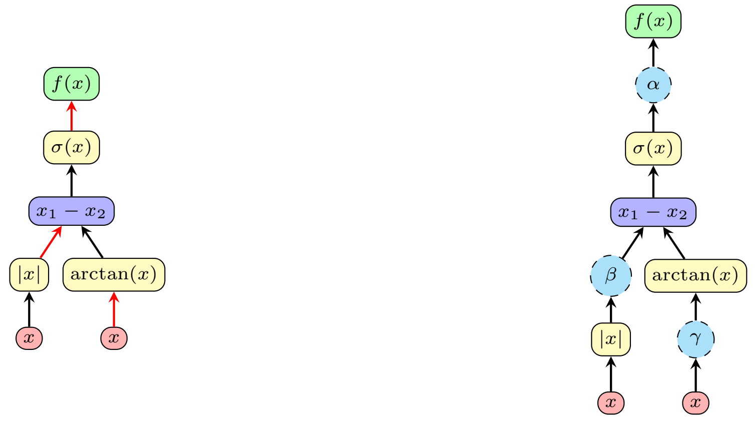

Figure 3: Parameterization of activation functions. In this example, parameters are added to $k = 3$ random edges, yielding the parametric activation function $\alpha\sigma(\beta|x| - \arctan(\gamma x))$.

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.

3.2. Discovering Activation Functions with Evolution

Activation functions are discovered by regularized evolution [31]. 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. Datasets and Architectures

The experiments in this paper focus primarily on the CIFAR-100 image classification dataset [21]. 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 for training details and Appendix D for code that shows how to train with custom activation functions).

Wide Residual Network [WRN-10-4; 40] 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; 14], 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; 15] 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.

5. Main Results

5.1. 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 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 Appendix B for implementation and compute details).

Table 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 Appendix C for baseline activation function details and Appendix G for additional results with learnable baseline functions). In sum, PANGAEA discovered the best activation function for each of the three architectures.

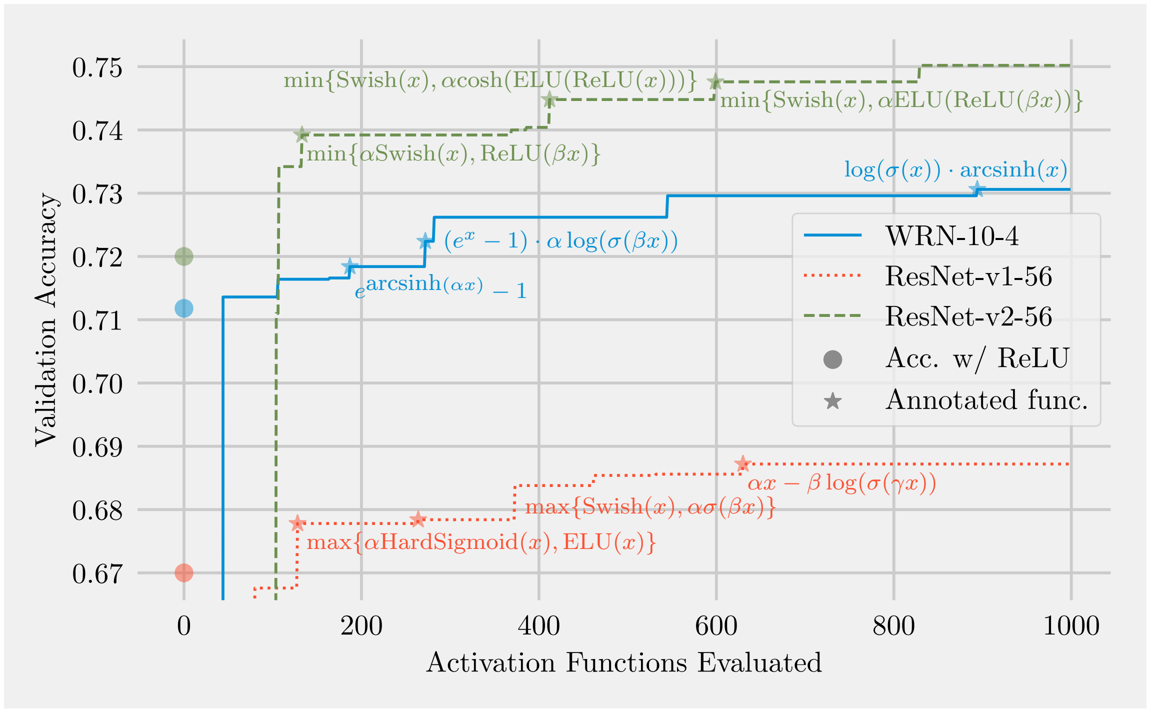

Figure 4: Progress of PANGAEA with three different neural networks. The plots show the best accuracy achieved among all activation functions evaluated so far. The stars on the plot specify the time when notable activation functions were discovered during evolution; the expression for each such function 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 2 lists test set accuracy.

5.2. 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 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 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

Table 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 Appendix C.

	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	$72.16_{\pm 0.81}$ *	$01.00_{\pm 0.00}$	$73.45_{\pm 0.43}$

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 meta-learning not considered by previous work [3, 4, 30].

5.3. 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.

5.4. 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. [30], which requires binary operators. They also do not exist in the S_1 or S_2 search spaces proposed by Bingham et al. [4], 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 Appendix F 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 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.

6. Diving Deeper: Experiments on Ablations, Variations, and Training Dynamics

PANGAEA is a method with many moving parts. The main results from Section 5 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 6.1). The top activation functions benefitted from their learnable parameters (Section 6.2), but baseline functions did not (Section 6.3), showing how PANGAEA discovered functional forms well-suited to parameterization.

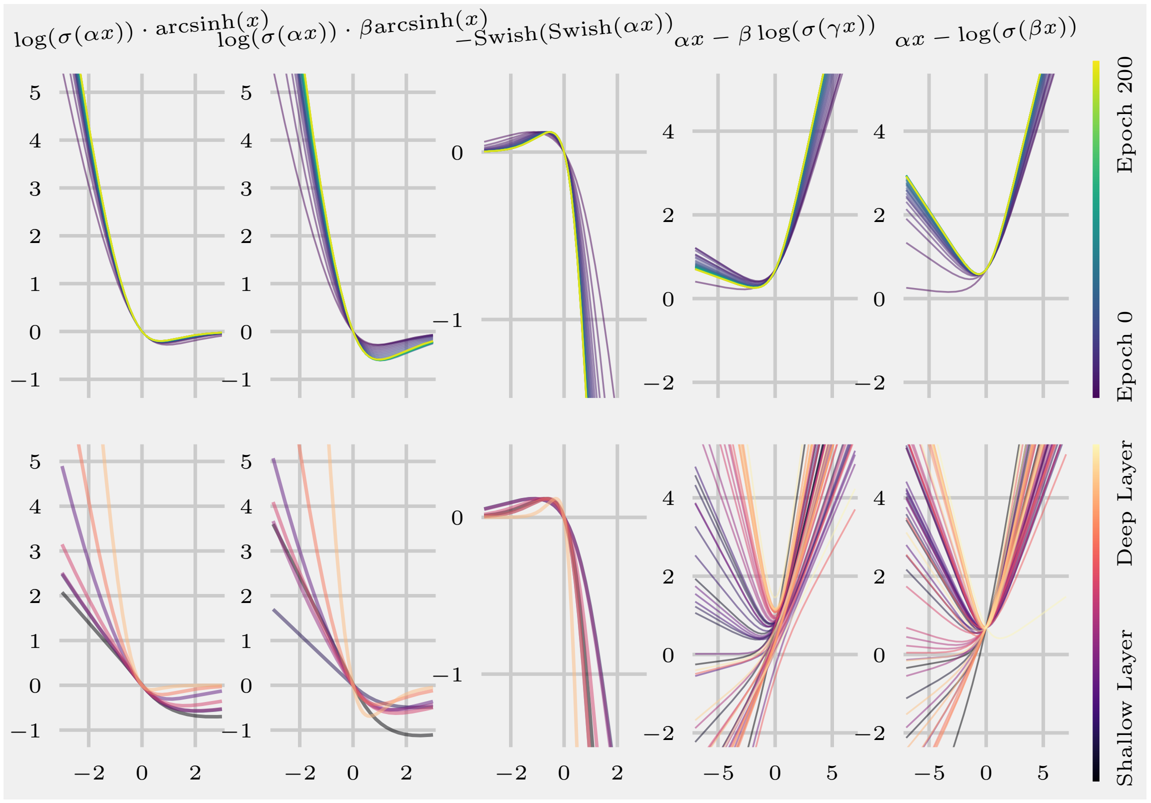

Figure 5: Adaptation of parametric activation functions over time and space. **Top:** The parameters change during training, resulting in different activation functions in the early and late stages. The plots were created by averaging the values of α, β, and γ across the entire network at different training 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.

PANGAEA is robust: the activation functions it discovers transfer to larger networks (Section 6.4) and PANGAEA achieves impressive results with a new dataset and architecture (Section 6.5). The activation functions discovered by PANGAEA improve accuracy by easing optimization and implicitly regularizing the network (Section 6.6).

6.1. 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 3). Otherwise, both baselines follow the same setup as PANGAEA, including evaluating $C = 1{,}000$ candidate functions and reranking the most promising ones (Section 3.2).

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

Table 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.

6.2. Effect of Parameterization

To understand the effect that parameterizing activation functions has on their performance, the specialized functions (Table 2) were trained without them. As Table 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 2). These results confirm that the learnable parameters contributed to the success of PANGAEA.

6.3. Adding Parameters to Fixed Baseline Activation Functions

As demonstrated in Tables 2 and 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 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.

Table 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	
$\log(\sigma(\alpha x)) \cdot \beta\text{arcsinh}(x)$	**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$_{\pm 0.30}$
$-\text{Swish}(\text{Swish}(\alpha x))$	**72.49**$_{\pm 0.55}$
$-\text{Swish}(\text{Swish}(x))$	71.97$_{\pm 0.22}$
ResNet-v1-56	
$\alpha x - \beta \log(\sigma(\gamma x))$	**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$_{\pm 0.39}$
$\log(\sigma(x))$	**75.53**$_{\pm 0.37}$

Table 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			
$\log(\sigma(\alpha x)) \cdot \beta\text{arcsinh}(x)$	**73.20**$_{\pm 0.37}$		
$\alpha x - \beta \log(\sigma(\gamma x))$		**71.01**$_{\pm 0.64}$	
Softplus(ELU(x))			**75.61**$_{\pm 0.42}$
Parameterized Functions			
αReLU(βx)	71.96$_{\pm 0.31}$	68.93$_{\pm 0.22}$	73.52$_{\pm 0.37}$
αELiSH(βx)	01.00$_{\pm 0.00}$	01.00$_{\pm 0.00}$	73.94$_{\pm 0.33}$
αELU(βx)	71.98$_{\pm 0.24}$	69.06$_{\pm 0.37}$	73.97$_{\pm 0.45}$
αGELU(βx)	71.96$_{\pm 0.34}$	69.39$_{\pm 0.35}$	73.83$_{\pm 0.24}$
αHardSigmoid(βx)	66.70$_{\pm 0.64}$	34.33$_{\pm 6.53}$	65.10$_{\pm 0.40}$
αLeaky ReLU(βx)	71.74$_{\pm 0.39}$	69.11$_{\pm 0.47}$	73.44$_{\pm 0.29}$
αMish(βx)	72.11$_{\pm 0.31}$	69.51$_{\pm 0.67}$	73.72$_{\pm 0.32}$
αSELU(βx)	71.07$_{\pm 0.33}$	68.05$_{\pm 0.39}$	73.37$_{\pm 0.38}$
αsigmoid(βx)	66.98$_{\pm 0.66}$	44.40$_{\pm 2.62}$	66.98$_{\pm 0.85}$
αSoftplus(βx)	71.73$_{\pm 0.31}$	68.84$_{\pm 0.30}$	73.95$_{\pm 0.37}$
αSoftsign(βx)	62.12$_{\pm 0.83}$	09.18$_{\pm 13.75}$	68.87$_{\pm 0.38}$
αSwish(βx)	72.26$_{\pm 0.29}$	69.25$_{\pm 0.28}$	73.93$_{\pm 0.22}$
αtanh(βx)	63.55$_{\pm 0.56}$	02.92$_{\pm 6.07}$	69.55$_{\pm 0.62}$

Table 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	
$\log(\sigma(\alpha x)) \cdot \beta \text{arcsinh}(x)$	**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	**71.23**$\pm_{0.25}$
ResNet-v2-110	
$\text{Softplus}(\text{ELU}(x))$	**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$\pm_{0.11}$

6.4. Scaling Up to Larger Networks

PANGAEA discovered specialized activation functions for WRN-10-4, ResNet-v1-56, and ResNet-v2-56. Table 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. [15]).

Evolving novel activation functions can be computationally expensive. The results in Table 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.

6.5. 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 [32] 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 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% reduction in the

Table 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.

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

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

6.6. 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 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 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 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 [22].

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 6c). Others have curves that start out lower than those from ReLU, but surpass it within a few dozen epochs (Figure 6d). Such diverse behavior suggests that these mechanisms can be combined in complex ways. Thus, the plots in Figure 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.

7. 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

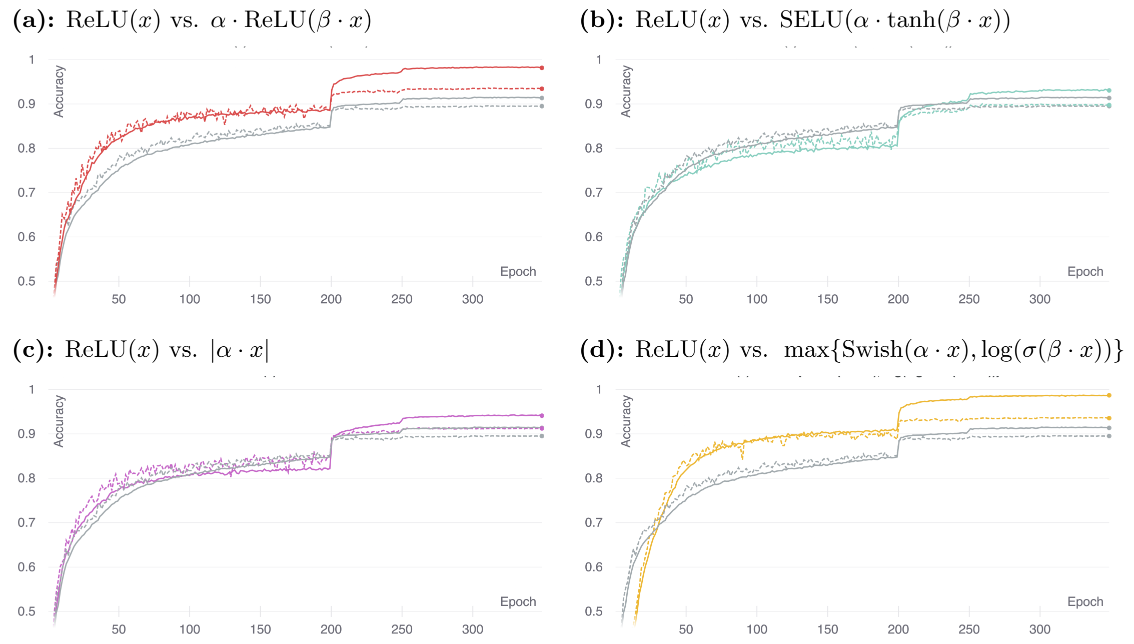

Figure 6: Training curves of All-CNN-C on CIFAR-10 with different activation functions. Accuracy with ReLU is shown in gray, and accuracy with the discovered functions in different colors. The training accuracy is shown as a solid line, and validation accuracy as a dashed line. All of these discovered 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.

with per-channel parameters. The results show that PANGAEA is effective with all three variations. Third, statistics from activation functions 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.

7.1. 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 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 8 shows that both PANGAEA runs discovered multiple good activation functions. The accuracies

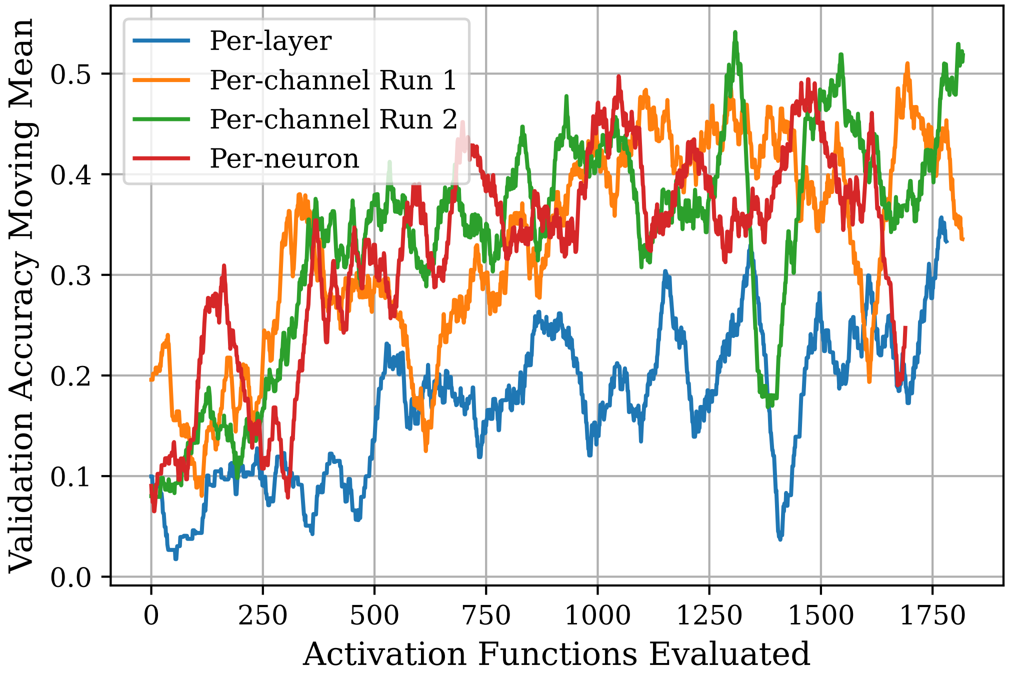

Figure 7: Average population fitness across time for four independent PANGAEA runs. The plots show the average validation accuracy achieved with the 64 most recently evaluated activation functions at any given time in the search process. All four PANGAEA runs gradually discover better activation functions 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.

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 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 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 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.

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.

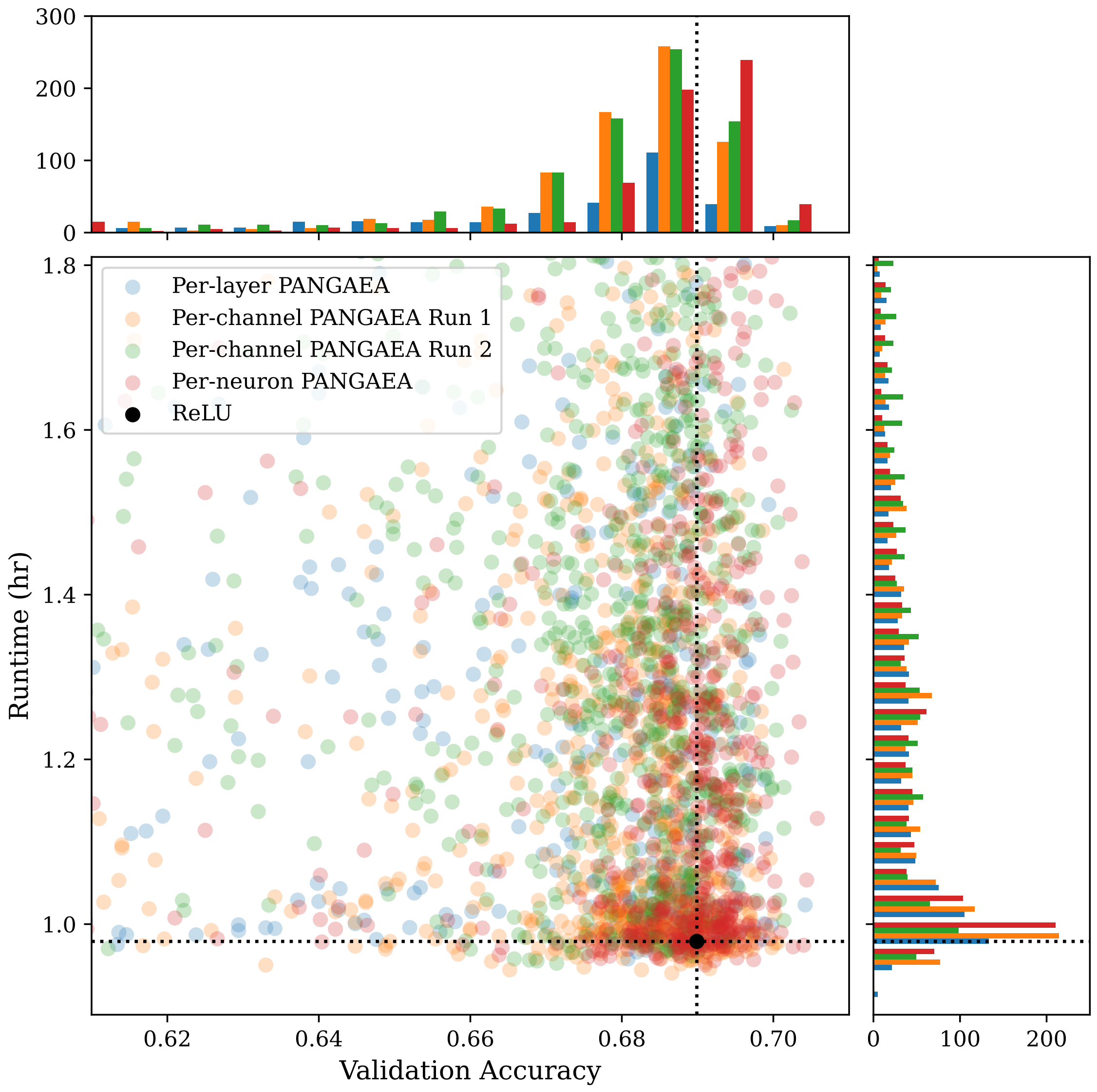

Figure 8: Fitness (validation accuracy) and compute cost (runtime in hours) among all activation functions considered in four independent PANGAEA processes on ResNet-v1-56. Each point represents a different activation function. The distribution of fitness and compute cost are shown in histograms on the top and right, respectively. All PANGAEA variants explore activation functions of similar complexity and reliably discover many novel functions that outperform ReLU.

7.2. 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 [13]. However, He et al. [13] also experimented with per-channel and per-layer implementations. Further preliminary experiments for this paper (Table 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 run from scratch on ResNet-v1-56. Both of these PANGAEA runs produced good activation functions that outperformed ReLU substantially (Table 8). Interestingly, although per-layer PReLU outperformed per-neuron PReLU with ResNet-v1-56 (Table 9), PANGAEA performed the best in the per-neuron setting (Table 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 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 1, and then mutated randomly three times as shown in Figure 2). The best activation functions from these random samples are included in Table 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.

7.3. 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 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

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	$69.64_{\pm 0.65}$

Table 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.

	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	$\mathbf{72.23}_{\pm 0.37}$	$69.77_{\pm 0.40}$	$\mathbf{75.10}_{\pm 0.53}$

Table 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.

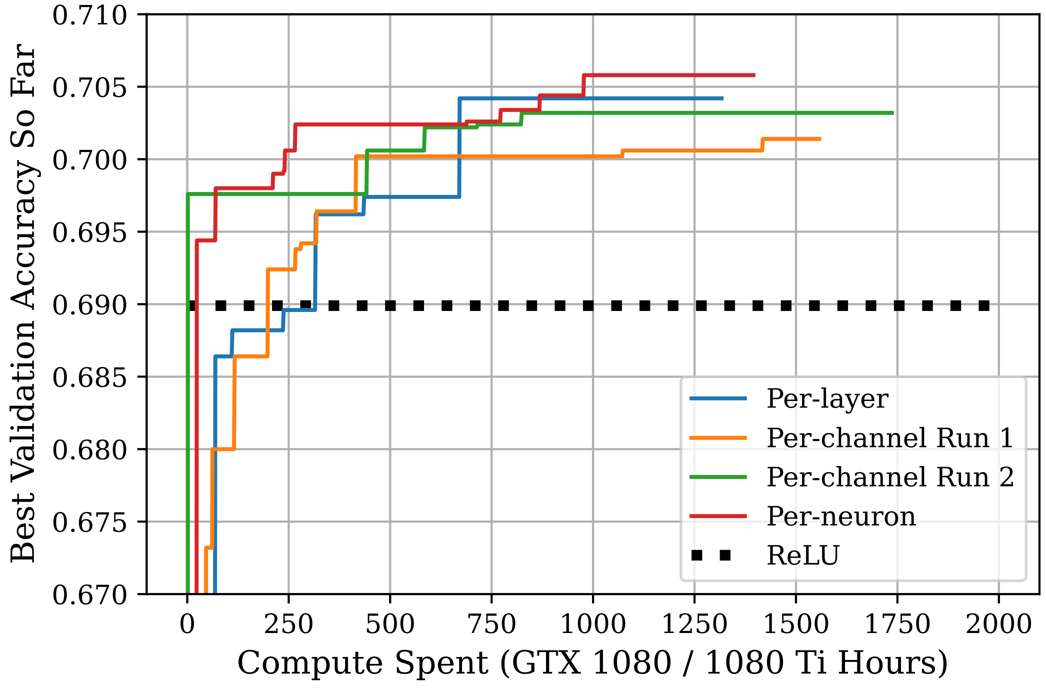

Figure 9: Computational efficiency of PANGAEA. The plot shows the performance of the best activation function discovered so far (y-axis) after a given amount of compute was spent (x-axis). All four PANGAEA runs discover activation functions that outperform ReLU with relatively little compute, demonstrating the efficiency of PANGAEA. If even better performance is needed, additional compute can be spent.

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 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 8 shows the distribution of accuracy and compute cost of all activation 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.

8. Future Work

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 [8, 31, 38]. In related work, evolution was used discover novel loss functions automatically [10, 11, 23], 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 [34], which achieved state-of-the-art accuracy for ImageNet [6] using the Swish activation function [7, 30]. Coevolution 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 8), improve adversarial robustness [39], stabilize training [19], or meet other objectives like easing optimization or providing implicit regularization [22]. 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.

9. Conclusion

This paper 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, 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. 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.

Acknowledgements

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

[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. Basirat and P. M. Roth. The quest for the golden activation function. *arXiv:1808.00783*, 2018.

[4] 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.

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

[6] 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.

[7] 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.

[8] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[9] 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.

[10] S. Gonzalez and R. Miikkulainen. Improved training speed, accuracy, and data utilization through loss function optimization. *arXiv:1905.11528*, 2019.

[11] S. Gonzalez and R. Miikkulainen. Evolving loss functions with multivariate taylor polynomial parameterizations. *arXiv:2002.00059*, 2020.

[12] M. Goyal, R. Goyal, and B. Lall. Learning activation functions: A new paradigm of understanding neural networks. *arXiv:1906.09529*, 2019.

[13] 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.

[14] 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.

[15] 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.

[16] D. Hendrycks and K. Gimpel. Gaussian error linear units (gelus). *arXiv:1606.08415*, 2016.

[17] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012.

[18] 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.

[19] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[20] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[21] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[22] Y. Li, C. Wei, and T. Ma. Towards explaining the regularization effect of initial large learning rate in training neural networks. *arXiv:1907.04595*, 2019.

[23] J. Liang, S. Gonzalez, and R. Miikkulainen. Population-based training for loss function optimization. *arXiv:2002.04225*, 2020.

[24] H. Liu, A. Brock, K. Simonyan, and Q. V. Le. Evolving normalization-activation layers. *arXiv:2004.02967*, 2020.

[25] 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.

[26] D. Misra. Mish: A self regularized non-monotonic neural activation function. *arXiv:1908.08681*, 2019.

[27] A. Molina, P. Schramowski, and K. Kersting. Pad\'e activation units: End-to-end learning of flexible activation functions in deep networks. *arXiv:1907.06732*, 2019.

[28] 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.

[29] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv:1811.03378*, 2018.

[30] 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.

[31] 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.

[32] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. In *ICLR (workshop track)*, 2015.

[33] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[34] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019.

[35] M. Tavakoli, F. Agostinelli, and P. Baldi. Splash: Learnable activation functions for improving accuracy and adversarial robustness. *arXiv:2006.08947*, 2020.

[36] 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.

[37] 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.

[38] M. Wistuba, A. Rawat, and T. Pedapati. A survey on neural architecture search. *arXiv:1905.01392*, 2019.

[39] C. Xie, M. Tan, B. Gong, A. Yuille, and Q. V. Le. Smooth adversarial training. *arXiv:2006.14536*, 2020.

[40] S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv:1605.07146*, 2016.

Appendix A. Training Details

Appendix A.1. Wide Residual Network (WRN-10-4)

When measuring final performance after evolution, the standard WRN setup is used; all ReLU activations in WRN-10-4 are replaced with the evolved activation function, but no other changes to the architecture are made. The network is optimized using stochastic gradient descent with Nesterov momentum 0.9. The network is trained for 200 epochs; the initial learning rate is 0.1, and it is decreased by a factor of 0.2 after epochs 60, 120, and 160. Dropout probability is set to 0.3, and L2 regularization of 0.0005 is applied to the weights. Data augmentation includes 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 [40] as closely as possible.

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

Appendix A.2. Residual Network (ResNet-v1-56)

As with WRN-10-4, when measuring final performance with ResNet-v1-56, the only change to the architecture is replacing the ReLU activations with an evolved activation function. The network is optimized with stochastic gradient descent and momentum 0.9. Dropout is not used, and L2 regularization of 0.0001 is applied to the weights. In the original ResNet experiments [14], 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 this paper, the learning rate schedule is 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 is trained for 200 epochs in total. Data augmentation includes a random horizontal flip and random 32×32 crops of images padded with four pixels on all sides, as in the original setup [14].

When evolving activation functions for ResNet-v1-56, the learning rate schedule is again compressed. The network is trained for 100 epochs; the initial warmup learning rate of 0.01 still lasts one epoch, the learning rate increases to 0.1, and then decreases 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 is therefore a cost-efficient way of discovering high-performing activation functions.

Appendix A.3. Preactivation Residual Network (ResNet-v2-56)

The full training setup, data augmentation, and compressed learning rate schedule used during evolution for ResNet-v2-56 are 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 [15], so this step is skipped.

Appendix A.4. All-CNN-C

When measuring final performance with All-CNN-C, the ReLU activation function is replaced with an evolved one, but the setup otherwise mirrors that of Springenberg et al. [32] as closely as possible. The network is optimized with stochastic gradient descent and momentum 0.9. Dropout probability is 0.5, and L2 regularization of 0.001 is applied to the weights. The data augmentation involves 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 is set to 0.01, and it is decreased by a factor of 0.1 after epochs 200, 250, and 300. The network is 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.

Appendix A.5. 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.

Appendix B. Implementation and Compute Requirements

High-performance computing in two clusters is utilized for the experiments. One cluster uses HTCondor [36] for scheduling jobs, while the other uses the Slurm workload manager. Training is executed on GeForce GTX 1080 and 1080 Ti GPUs on both clusters. When a job begins executing, a parent activation function is 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 [31], which is based on a strict sliding window of size P. This approach may give extra influence to some activation functions, depending on how quickly or slowly jobs are executed in each of the clusters. In practice the method is highly effective; it allows evolution to progress quickly by taking advantage of extra compute when demand on the clusters is low.

It is difficult to know ahead of time how computationally expensive the evolutionary search will be. Some activation functions immediately result in an undefined loss, causing training to end. In that case only a few seconds have been spent and another activation function can immediately be evaluated. Other activation functions train successfully, but their complicated expressions result in longer-than-usual training times. Although substantial, the computational cost is negligible compared to the cost in human labor in designing activation functions. Evolution of parametric activation functions requires minimal manual setup and delivers automatic improvements in accuracy.

Appendix C. Baseline Activation Function Details

The following activation functions were used as baseline comparisons in Table 2. Some functions were also utilized in the search space (Table 1).

- ReLU $= \max\{x, 0\}$ [28].
- ELiSH $= \frac{x}{1+e^{-x}}$ `if` $x \geq 0$ `else` $\frac{e^x-1}{1+e^{-x}}$ [3].
- ELU $= x$ `if` $x \geq 0$ `else` $\alpha(e^x - 1)$, with $\alpha = 1$ [5].
- GELU $= x\Phi(x)$, with $\Phi(x) = P(X \leq x), X \sim \mathcal{N}(0, 1)$, approximated as $0.5x(1+\tanh[\sqrt{2/\pi}(x+0.044715x^3)])$ [16].
- HardSigmoid $= \max\{0, \min\{1, 0.2x + 0.5\}\}$.
- Leaky ReLU $= x$ `if` $x \geq 0$ `else` $0.01x$ [25].
- Mish $= x \cdot \tanh(\text{Softplus}(x))$ [26].
- SELU $= \lambda x$ `if` $x \geq 0$ `else` $\lambda\alpha(e^x - 1)$, with $\lambda = 1.05070098$, $\alpha = 1.67326324$ [19].
- sigmoid $= (1 + e^{-x})^{-1}$.
- Softplus $= \log(e^x + 1)$.
- Softsign $= x/(|x| + 1)$.
- Swish $= x \cdot \sigma(x)$, with $\sigma(x) = (1 + e^{-x})^{-1}$ [7, 30].
- 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 [13].
- PSwish $= x \cdot \sigma(\beta x)$, where β is a per-channel learnable parameter [30].
- 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 [27].
- 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 $a = 0$ [35].

Appendix D. Training with 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)
```

Here, `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
```

Appendix E. 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 1. In the main text, 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

$$f(x) = \sum_{n=0}^{\infty} a_n (x - x_0)^n, \tag{E.1}$$

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 E.1 are included in Table 1, and so $f \in \mathcal{G}_\infty$. □

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

$$\mathbf{1}_{(-\infty,b)}(x) = \begin{cases} 1 & x < b \\ 0 & x \geq b \end{cases} \tag{E.2}$$

$$\mathbf{1}_{(a,\infty)}(x) = \begin{cases} 1 & x > a \\ 0 & x \leq a \end{cases} \tag{E.3}$$

$$\mathbf{1}_{(a,b)}(x) = \begin{cases} 1 & x \in (a,b) \\ 0 & x \notin (a,b) \end{cases} \tag{E.4}$$

$$\mathbf{1}_{a}(x) = \begin{cases} 1 & x = a \\ 0 & x \neq a \end{cases} \tag{E.5}$$

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

$$\mathbf{1}_{(-\infty,b)}(x) = \frac{\max\{b - x, 0\}}{b - x}. \tag{E.6}$$

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,

$$\mathbf{1}_{(a,\infty)}(x) = \frac{\min\{a - x, 0\}}{a - x}, \tag{E.7}$$

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

$$\mathbf{1}_{(a,b)} = \mathbf{1}_{(-\infty,b)} \mathbf{1}_{(a,\infty)} \tag{E.8}$$

and

$$\mathbf{1}_a = (1 - \mathbf{1}_{(-\infty,a)})(1 - \mathbf{1}_{(a,\infty)}). \tag{E.9}$$

All operators in Equations E.6-E.9 (maximum, minimum, subtraction, multiplication, division, zero, one) are in the PANGAEA search space in Table 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

$$f(x) = \begin{cases} f_0(x) & x \in (-\infty, k_1) \\ K_1 & x = k_1 \\ f_1(x) & x \in (k_1, k_2) \\ K_2 & x = k_2 \\ & \vdots \\ f_{n-1}(x) & x \in (k_{n-1}, k_n) \\ K_n & x = k_n \\ f_n(x) & x \in (k_n, \infty) \end{cases}, \tag{E.10}$$

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:

$$\begin{aligned} f(x) = \mathbf{1}_{(-\infty,k_1)}(x) f_0(x) + \mathbf{1}_{k_1}(x) K_1 + \mathbf{1}_{(k_1,k_2)}(x) f_1(x) + \mathbf{1}_{k_2}(x) K_2 + \cdots \\ + \mathbf{1}_{(k_{n-1},k_n)}(x) f_{n-1}(x) + \mathbf{1}_{k_n}(x) K_n + \mathbf{1}_{(k_n,\infty)}(x) f_n(x). \end{aligned} \tag{E.11}$$

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 E.11 utilizes only addition and multiplication, both of which are operators included in Table 1. Therefore, $f \in \mathcal{G}_\infty$. □

Appendix F. Size of the PANGAEA Search Space

This section analyzes the size of the PANGAEA search space as implemented in experiments in the main text. 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 $f(x) = \tanh(x) + \operatorname{erf}(x)$, 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 1 in the main text).

	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	

Table F.10: 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.

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 the set to be

$$|\mathcal{F}_g| = U^{u_g} \cdot B^{b_g} \cdot \sum_{i=0}^{E} \binom{e_g}{i}. \tag{F.1}$$

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

$$\mathcal{G}_3 = \{g(x) = \texttt{unary1}(\texttt{unary2}(\texttt{unary3}(x))), g(x) = \texttt{binary}(\texttt{unary1}(x), \texttt{unary2}(x))\}. \tag{F.2}$$

Table F.10 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

$$g(x) = \texttt{binary1}(\texttt{binary2}(\texttt{unary1}(x), \texttt{unary2}(x)), \texttt{unary3}(x)) \tag{F.3}$$

or

$$g(x) = \texttt{binary1}(\texttt{unary1}(x), \texttt{binary2}(\texttt{unary2}(x), \texttt{unary3}(x))). \tag{F.4}$$

The number of activation functions in PANGAEA is therefore

$$\sum_{j=1}^{7} \sum_{g \in \mathcal{G}_j} |\mathcal{F}_g| = 10{,}170{,}042{,}948{,}450. \tag{F.5}$$

	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

Table G.11: 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.

Naturally there exist duplicates within this space. The functions $f(x) = \text{ReLU}(x)$ 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 [3, 4, 30], and yet PANGAEA consistently discovers functions that outperform ReLU and other baseline functions.

Appendix G. Additional Results with Learnable Activation Functions

PAU and SPLASH achieved worse-than-expected performance in Table 2, so additional experiments were run to investigate their behavior.

PAU. Molina et al. [27] 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 appendix 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 [27]. 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 Leaky ReLU with a slope of 0.01.

Table G.11 shows the performance of WRN-10-4 and ResNet-v2-56 using these discovered hyperparameters and the specialized training setup from Molina et al. [27]. 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 paper for all baseline comparisons. The reason is that there are dozens of such comparisons in this paper, 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.

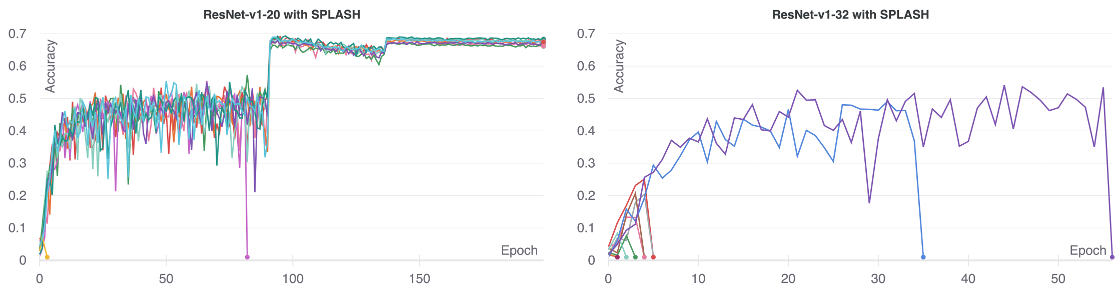

Figure G.10: ResNet-v1-20 and ResNet-v1-32 test accuracy on CIFAR-100 with the SPLASH activation function. SPLASH units work well in shallow networks, but become unstable with increased depth. This result explains the success of SPLASH in the original work by Tavakoli et al. [35], and also shows why SPLASH units fail with the architectures considered in this work.

SPLASH. In addition to ResNet-v1-56 in the main text, 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 G.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. [35]. However, two of the ten runs failed in the middle of training because the loss became undefined (Figure G.10), suggesting that SPLASH units can be unstable. Progressing to the deeper ResNet-v1-32, the effect was more pronounced. As shown in Figure G.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. [35]. 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 paper.