Neuron-Level Activation Learning in Neural Networks: A Self-Configuring AI Approach

Abstract: Neural networks traditionally use fixed activation functions (ReLU, tanh, etc.) for all neurons, which may limit representational flexibility. We propose Self-Activating Neural Networks (SANN), a novel architecture in which each neuron learns its own activation function. In the proposed approach, every neuron's activation is a parametrized piecewise-linear function learned via gradient descent alongside network weights. This per-neuron adaptive activation enables the network to learn the optimal nonlinearity for each neuron from data, rather than relying on a one-size-fits-all activation. We detail the architecture and learning algorithm for these self-activating units, including the mathematical formulation of the piecewise-linear functions and their gradients. Experiments on benchmark tasks (MNIST and CIFAR-10) demonstrate that SANN can outperform networks with fixed activations (e.g., ReLU), yielding faster convergence and higher accuracy. We present training and validation curves, showing that SANN maintains good generalization without overfitting. Theoretical analysis indicates that SANN significantly enhances expressiveness, as each neuron can implement a flexible continuous piecewise-linear mapping, potentially reducing the depth or width needed to approximate complex functions. We discuss practical implementation details, regularization methods to prevent overfitting, and the broader implications of making activation functions learnable. This work opens the door to a new class of neural networks that learn both weights and activation shapes, potentially leading to more compact and adaptive models. Future directions include extending self-activating units to recurrent, graph, and transformer architectures. We conclude that learning activation functions per neuron can be a powerful approach to increase neural network flexibility and performance, moving beyond the constraints of fixed activation functions.

Introduction

Deep neural networks owe much of their success to the choice of activation functions that introduce nonlinearity into neurons' outputs. Historically, activation functions such as the sigmoid and hyperbolic tangent were popular but suffered from saturation at extreme inputs, causing vanishing gradients in deep networks. The advent of the Rectified Linear Unit (ReLU) addressed this issue by using a simple piecewise-linear function $f(x)=\max(0,x)$. The ReLU significantly improved training dynamics by alleviating vanishing gradients and enabling deep networks to learn effectively (Glorot et al., 2011). ReLU-based networks (e.g., Krizhevsky et al., 2012) became easier to optimize and achieved state-of-the-art results in image recognition. Due to its simplicity and effectiveness, ReLU has become the de facto default activation in modern deep learning. Nevertheless, fixed activations like ReLU may not be optimal for all tasks or all neurons. Using the same activation shape (such as ReLU's linear-for-positive, flat-for-negative response) uniformly across all neurons can constrain the network's expressive power.

Recent research has explored alternatives to ReLU, including leaky ReLUs (Maas et al., 2013), parametric ReLUs (PReLU) (He et al., 2015), exponential linear units (ELU) (Clevert et al., 2016), scaled ELU (SELU) (Klambauer et al., 2017), and swish (Ramachandran et al., 2017). Many of these provide modest performance gains in certain settings, but no single

fixed function has emerged as universally superior to ReLU. Moreover, these functions are still globally applied—every unit in a layer (or even the whole network) uses the same functional form (with perhaps a parameter tweak as in PReLU). This global uniformity might be suboptimal: different neurons, at different layers or representing different features, could benefit from distinct nonlinear response shapes.

A more powerful approach is to learn the activation functions themselves from data. Early efforts in this direction date back to evolutionary algorithms that attempted to evolve or select an activation function for each neuron from a predefined set (Yao, 1999). For example, genetic algorithms have been used to choose among sigmoid, Gaussian, or linear activations for each unit. These methods demonstrated the potential benefits of heterogeneous activations but were limited by the discrete nature of selection and the computational cost of evolutionary search. Turner & Miller (2014) introduced an approach combining evolution with learned scaling parameters, allowing slight tuning of each chosen activation. However, such methods still did not fully leverage continuous activation shapes with gradient-based learning.

In this paper, we propose a novel neural network architecture called Self-Activating Neural Networks (SANN), in which each neuron learns its own activation function during training. Each activation is parameterized in a flexible form and updated by gradient descent simultaneously with the network's weights. Unlike prior fixed or globally parameterized activations, SANN endows each neuron with the capacity to shape its input-output relationship to best fit the data. Our specific instantiation of SANN uses parametric piecewise-linear functions as neuron-wise activation functions. Piecewise-linear functions are a natural choice because they are universal approximators for continuous functions and are simple, efficient, and compatible with ReLU-based optimization techniques. Notably, our approach is inspired by the adaptive piecewise-linear unit (APL) of Agostinelli et al. (2015), which first demonstrated that learning a piecewise-linear activation per neuron can yield state-of-the-art performance on image classification benchmarks. We extend and formalize this concept under the SANN framework, providing a detailed formulation, theoretical analysis, and new experiments and insights.

The **novelty** of this work lies in proposing a fully trainable per-neuron activation architecture and thoroughly exploring its implications. Our contributions are summarized as follows:

- **Learning Activation Functions per Neuron:** We introduce the SANN architecture in which each neuron has its own learnable activation function. To our knowledge, this represents a significant step beyond traditional networks that use one or a few fixed activation types globally. We formulate a flexible piecewise-linear parameterization that can represent a wide range of shapes (convex, concave, non-monotonic), allowing neurons to adapt their response during training.
- **Parametrized Piecewise-Linear Units:** We propose a practical parameterization for per-neuron activations using a sum of hinge-shaped linear pieces. This yields a **continuous piecewise-linear function** for each neuron, with only a small number of extra parameters per neuron. We provide the mathematical formulation, derivation of gradients, and a discussion of how these "activation parameters" are learned alongside normal weights. The approach is designed to be easily implemented in modern deep learning frameworks with negligible computational overhead.
- **Theoretical Analysis of Expressiveness:** We analyze the expressive power of SANNs. We show that a single neuron in our model can approximate any continuous

piecewise-linear function on $\mathbb{R}$ (under mild conditions) given enough segments, which we formally state and discuss. This implies that networks with self-activating neurons can represent functions that would otherwise require deeper or wider architectures with fixed activations. We discuss how learning activations per neuron can reduce the network size needed for a given task, and we examine possible risks such as overfitting due to the increased flexibility. Strategies for regularizing the learned activation functions are also presented.

- **Empirical Evaluation:** We conduct experiments on benchmark datasets (the MNIST handwritten digits and CIFAR-10 image classification tasks) to evaluate the performance of SANN against standard networks using ReLU or other fixed activations. Using identical network architectures (apart from activation mechanism), we compare training dynamics, convergence speed, and final generalization performance. Our results show that SANN consistently matches or outperforms the baseline networks. For example, on MNIST we achieve slightly higher test accuracy than a ReLU network, with faster convergence in terms of epochs. On CIFAR-10, a convolutional SANN model achieves a notable accuracy improvement over the ReLU counterpart, demonstrating the benefit of learnable per-neuron activations in a more complex setting. We include detailed training curves (loss and accuracy) to illustrate learning behavior, and we analyze the learned activation functions from trained models to gain insight into how they differ from fixed functions (Fig. 1).
- **Practical Considerations:** We detail how SANN can be implemented and trained in practice. We describe initialization schemes (e.g., start with all neurons using ReLU-like behavior), techniques to constrain or regularize the activation parameters to prevent pathological shapes, and the computational cost. We show that the overhead in parameter count is very small (on the order of a few parameters per neuron, which is negligible compared to weights) and that inference with learned piecewise-linear activations is efficient. We also discuss the integration of SANN with other architectural components like batch normalization and dropout. Our implementation is straightforward and leverages standard automatic differentiation for training the activation parameters.

The remainder of this paper is structured as follows. In **Section 2 (Related Work)**, we review prior activation functions and approaches to learn or adapt them, positioning our contribution in context. **Section 3 (Proposed Methodology)** describes the SANN architecture and the parametrized piecewise-linear activation functions in detail, including mathematical formulation and learning algorithm. **Section 4** covers **implementation details** and practical considerations for training SANN models. In **Section 5 (Experiments)**, we present empirical results on two benchmark tasks, with comparative evaluations, ablation of parameter choices, and visualizations of learned activations. **Section 6 (Theoretical Analysis)** discusses the expressive power of self-activating neurons and considerations regarding capacity and regularization. **Section 7 (Discussion)** elaborates on the implications, benefits, and potential challenges of the approach, and **Section 8 (Future Work)** outlines possible extensions of this research to other network architectures and activation function forms. Finally, **Section 9 (Conclusion)** summarizes our findings and contributions. Full references are provided at the end of the paper.

Related Work

Fixed Activation Functions: The choice of activation function has long been recognized as crucial for neural network performance (Hornik, 1989). Early neural networks predominantly

used **sigmoid** or **tanh** activation functions, which introduce smooth nonlinearity but suffer from gradient saturation for large magnitude inputs. As networks became deeper, these saturating activations led to vanishing gradients, hampering training. The introduction of the **ReLU** (Jarrett et al., 2009; Glorot et al., 2011) addressed this problem by using a piecewise-linear function that is identity for positive inputs and zero for negative inputs. By not squashing large inputs, ReLU allows gradients to propagate well and encourages sparse activations. ReLU-based deep networks (e.g., AlexNet by *Krizhevsky et al.*, 2012) demonstrated unprecedented success in image classification, and ReLU quickly became the default activation in deep learning. Variants of ReLU were subsequently proposed to improve upon it. **Leaky ReLU** (LReLU) introduces a small slope for negative inputs instead of a hard zero, typically $f(x)=\max(0,x) + \alpha \min(0,x)$ with a fixed α (e.g. 0.01) to allow a small negative gradient (Maas et al., 2013). This addresses ReLU's "dying neuron" problem (where a neuron stuck with negative inputs never updates). **Parametric ReLU (PReLU)** extends this idea by making the negative slope α a learnable parameter (He et al., 2015). In PReLU, each channel (or neuron) can adapt its negative slope during training, giving a slight increase in flexibility over a fixed leaky ReLU. He et al. reported that PReLU improved model fitting with negligible computational cost and did not overfit, and using PReLUs helped their networks achieve superior results on ImageNet. Other modifications of ReLU include **Randomized ReLU (RReLU)** (Xu et al., 2015), which samples the negative slope from a distribution during training (and uses the average at test time) as a form of regularization, and **exponential linear units (ELU)** (Clevert et al., 2016), which replace the negative linear part with an exponential curve to force outputs to have mean zero, purportedly speeding up learning. The **Scaled ELU (SELU)** (Klambauer et al., 2017) further scales ELU outputs to preserve variance and was key in self-normalizing networks. Each of these innovations fixed certain drawbacks of ReLU or aimed to automatically ensure good initialization and normalization, but they still represent a fixed functional form (with at most one or two global/trainable parameters like the slope).

Learnable Activation Functions (Global): Rather than having a rigid form, some activation functions introduce one or more trainable parameters *shared across all neurons using that activation*. PReLU, mentioned above, can be seen as learnable but typically the slope parameter is often shared per layer or channel (to avoid too many parameters) – though one can have per-neuron slopes, that becomes many parameters without changing the functional form (it's still a "linearity plus a slope for negatives"). The **Swish** activation is a notable recent example discovered via automated search (Ramachandran et al., 2017). Swish is defined as $f(x) = x \cdot \sigma(\beta x)$, where σ is the sigmoid function and β is a constant or trainable scalar. In the simplest case $\beta=1$, $f(x)=x\sigma(x)$ is a smooth, non-monotonic function (it increases overall but has a slight bump for negative x). Ramachandran et al. found Swish by searching a space of possible activation formulas, and showed that replacing ReLUs with Swish units in deep networks often improves performance. For example, Swish outperformed ReLU by about 0.6–0.9% top-1 accuracy in certain ImageNet models. Swish can also be viewed as a smooth blend of identity and sigmoid, and it has a trainable parameter β that controls the shape (though in practice β is sometimes fixed to 1 or allowed to vary per layer). Another trainable activation is the **Parametric Softplus** (PSwish or β-Swish), which generalizes the softplus $f(x)=\log(1+e^x)$ by a slope parameter. **Mish** (Misra, 2019) is a newer self-regularized activation defined as $x \tanh(\log(1+e^x))$, which is fixed-form but was shown to give improvements on some tasks. Overall, these trainable or searched activations provide *one* global function with slight tunable aspects, which is an advance over purely fixed functions, but they do not fully customize the activation at the level of individual neurons.

Learnable Activation Functions (Per-Neuron / Adaptive): A different line of research has aimed to increase flexibility by allowing activation functions to vary across neurons and to be learned from data. One simple approach is to approximate an arbitrary activation via a linear combination of basis functions. For example, Chen et al. (2015) proposed activation functions represented as linear combinations of sigmoid or Gaussians with learnable coefficients (a form of basis expansion). An influential approach in this category is the **Maxout** neuron (Goodfellow et al., 2013). A maxout layer has no fixed activation; instead, each "neuron" computes $\max(z_1, z_2, ..., z_k)$ for a set of k linear pre-activations $z_i = \mathbf{w}_i^\top \mathbf{x} + b_i$. By taking the maximum of several affine functions, a maxout unit effectively produces a piecewise-linear convex function as its activation (with at most k pieces). Maxout units can approximate any convex function and were shown to achieve excellent performance on various benchmarks (Goodfellow et al. reported state-of-the-art results on MNIST, CIFAR-10, SVHN, etc.). However, maxout increases the number of parameters (each neuron has k sets of weights) and computations, and if one wanted different nonlinearities at different locations in the feature map (in CNNs) it would be parameter-intensive. Lin et al. (2014) took a related approach in the **Network-in-Network (NiN)** architecture, where they replace the scalar activation function with a learned micro-network (a multilayer perceptron) applied at each location of a convolutional feature map. For example, a 1×1 convolution followed by a nonlinearity can act as a learned activation function (NiN used a 1×1 conv with ReLU, essentially allowing a learned linear combination of channels before applying ReLU). NiN and Maxout demonstrate the advantages of trainable nonlinear transformations, but they drastically increase model complexity (each introduces many additional parameters). As a result, applying Maxout or NiN-style activations per neuron or per spatial location can become impractical in large networks.

A more parameter-efficient strategy is to learn a parameterized function for each neuron. The **Adaptive Piecewise Linear Unit (APL)** introduced by Agostinelli et al. (2015) is a seminal work in this direction. An APL unit's activation function is a piecewise-linear graph composed of multiple linear segments with learnable slopes and breakpoints. In their formulation, each neuron's activation is expressed as a sum of "hinge" functions (ReLU-like components) placed at various locations. Specifically, APL defines the activation for neuron i as:

hi(x)=max⁡(0,x) + ∑s=1Sai,s max⁡(0, bi,s−x) ,h_i(x) = \max(0, x) \;+\; \sum_{s=1}^{S} a_{i,s} \, \max(0,\, b_{i,s} - x)\,,hi(x)=max(0,x)+∑s=1Sai,smax(0,bi,s−x),

where S is a fixed number of additional piecewise components (hinges), and $a_{i,s}$ and $b_{i,s}$ are learnable parameters that determine the slope and the location of the s-th hinge for neuron i. The first term $\max(0,x)$ is just a ReLU (slope 1 for $x>0$, 0 for $x<0$), ensuring that as $x \to +\infty$, $h_i(x) \sim x$. Each additional term $a_{i,s}\max(0, b_{i,s}-x)$ contributes a piecewise-linear "bump" or adjustment that activates when x is below the corresponding breakpoint $b_{i,s}$. Intuitively, these terms allow the function to deviate from the identity line in the negative region or mid-range: for $x < b_{i,s}$, the term adds a line of slope $-a_{i,s}$ (since $b_{i,s}-x>0$), and for $x > b_{i,s}$ the term is inactive (0 contribution). By learning appropriate a and b values, the neuron can obtain a rich variety of shapes, including non-monotonic functions. Agostinelli et al. showed that with even a small number of hinges (e.g. $S=2$ or 3), APL units significantly improve accuracy on image classification benchmarks compared to ReLU units, without overfitting. For instance, a deep CNN with APL units achieved **7.51%** error on

CIFAR-10 and **30.83%** error on CIFAR-100, which at the time were state-of-the-art results, improving on standard ReLU networks. Notably, they achieved this with a relatively small increase in parameters: if there are N_h hidden neurons and each has S hinges, the total extra parameters is $N_h \times 2S$ (each hinge has an a and a b), which is minor compared to millions of weight parameters in a typical deep network. APL's success demonstrated the feasibility of learning activation functions per neuron with gradient descent.

Several other works have since built on the idea of learnable piecewise or polynomial activations. Zhou et al. (2021) proposed the **Piecewise Linear Unit (PWLU)**, which similarly learns piecewise-linear activation functions but with a formulation aimed at easier optimization and adaptivity. They argued that prior search-based methods for activations (like Swish discovery) were inefficient, and instead one can **learn a specialized activation function for each model and dataset** using PWLU. In their experiments, replacing Swish or ReLU with PWLU led to new state-of-the-art results on ImageNet and COCO detection tasks. For example, on ImageNet classification, integrating PWLU improved top-1 accuracy by **0.5–1.7%** across a variety of architectures (ResNet-50, MobileNet-V3, EfficientNet-B0, etc.) compared to the Swish activation. This is a remarkable gain at the model level purely by tuning the activation functions, underlining the importance of activation function flexibility. An extension called **non-uniform PWLU** was also explored to allocate segments more efficiently (more segments where the function changes rapidly). Other recent innovations include **SPLINES** or B-spline-based activations (Jagtap et al., 2020) where a smooth spline is learned, and **Padé Activation Units (PAU)** (Molina et al., 2020) which use learnable rational functions to approximate activation curves. While these methods differ in parameterization (splines, rational functions, etc.), they share the core idea of increasing the activation function's flexibility and allowing it to be learned from data.

Our approach falls squarely in the category of *per-neuron learnable activations* and is most closely related to APL and PWLU. We build on the piecewise-linear formulation, given its simplicity and proven effectiveness, and we propose a streamlined version with some modifications for stability and ease of use. Unlike some prior works that share activation parameters across a layer or restrict the flexibility, we allow **each neuron to have its own activation shape**, which maximizes expressiveness. At the same time, we keep the number of learnable parameters minimal and identical for each neuron (a fixed small S), ensuring the approach is scalable. Our contributions relative to existing work include a thorough theoretical and experimental exploration of such self-activating networks, demonstrations of their advantages on standard tasks, and discussions on how to manage and interpret the additional flexibility. In summary, this work takes the next step in the evolution of activation functions: from fixed, to globally parameterized, to *locally parameterized per neuron*, advancing the capacity of neural networks to shape their internal nonlinearities to the data at hand.

Proposed Methodology

Overview of Self-Activating Neural Networks

In a **Self-Activating Neural Network (SANN)**, each neuron is endowed with its own activation function, which is learned during training. This is achieved by giving each neuron a set of internal parameters that define a flexible function mapping its pre-activation input (the weighted sum before nonlinearity) to its output. These internal activation parameters are updated via backpropagation in

the same way as the weight parameters. The result is that, over the course of training, each neuron **adapts its activation shape** to better fit the overall model to the data. Neurons in different layers, or even different neurons in the same layer, can develop very different activation behaviors, if beneficial. This mechanism contrasts with a standard network where a fixed $f(\cdot)$ (e.g. ReLU or tanh) is applied uniformly.

We choose a **piecewise-linear parameterization** for the activation functions, both for its universal approximation capabilities and for computational convenience. Piecewise-linear functions are essentially what standard ReLUs already produce (with one linear piece for negative inputs – flat zero – and one linear piece of slope 1 for positive inputs). By introducing additional linear pieces, a neuron can approximate more complex functions. The key insight is that **any continuous piecewise-linear function** can be constructed by combining a sufficient number of ReLU-like hinges. We retain the ReLU's advantageous property of linearity at extremes: as input $x \to +\infty$ or $x \to -\infty$, we constrain the activation to grow linearly (or at most linearly) so that it does not diverge faster than its input, which helps preserve stability (this is achieved implicitly by our parameterization, as discussed below). Each neuron's activation is thus a polyline with a small number of segments that is learned. Because the function is linear in segments, computing it is as efficient as a series of ReLU operations; and because it is defined by just a few parameters, learning it is tractable with gradient methods.

Formally, consider a neuron i with pre-activation input $x_i = \mathbf{w}_i^\top \mathbf{x} + b_i$ (the weighted sum of outputs from the previous layer plus bias). In a conventional network, the output of neuron i would be $y_i = f(x_i)$ for some fixed f like ReLU. In SANN, the neuron has parameters defining its own activation $f_i(\cdot)$, and the output is $y_i = f_i(x_i)$, where f_i is learned. We represent f_i as a piecewise-linear function with $S+1$ linear pieces. One convenient representation (following Agostinelli et al. and our Equation above) is to use a base ReLU plus S additional "negative" ReLUs:

- **Base segment:** $h_i^{(0)}(x) = \max(0, x)$, which has slope 0 for $x<0$ and slope 1 for $x>0$. This gives a baseline of a ReLU.
- **Additional segments:** For each $s = 1,2,\dots,S$, we have a parameterized hinge of the form $h_i^{(s)}(x) = a_{i,s} , \max(0,, b_{i,s} - x)$. Here $a_{i,s}$ and $b_{i,s}$ are learnable parameters. The term $\max(0,, b_{i,s} - x)$ is essentially a ReLU that activates when $x < b_{i,s}$ (note it is zero for $x > b_{i,s}$ and linear with slope -1 for $x<b_{i,s}$). The coefficient $a_{i,s}$ scales this hinge's contribution.

The overall activation function for neuron i is then:

fi(x) = max⁡(0,x) + ∑s=1Sai,s max⁡(0, bi,s−x) .(1)f_i(x) \;=\; \max(0, x) \;+\; \sum_{s=1}^{S} a_{i,s}\,\max(0,\, b_{i,s} - x)\,. \tag{1}fi(x)=max(0,x)+s=1∑Sai,smax(0,bi,s−x).(1)

This can be viewed as the **defining equation** of a self-activating neuron's function (with S learned hinges). For clarity, let's interpret this formula. For input values **much larger** than all $b_{i,s}$, every $\max(0, b_{i,s}-x)$ will be 0 (since x is greater than each $b_{i,s}$), so all hinge terms drop out and $f_i(x) \approx \max(0,x)$. At sufficiently large x, $\max(0,x) = x$, so asymptotically $f_i(x) \sim x$ (a line of slope 1). This ensures the neuron's activation grows roughly linearly for very large positive inputs, preventing unbounded exponential growth and preserving the notion that the neuron will not "blow up" for large signals. For input values **much smaller** (more negative) than all

$b_{i,s}$, specifically if $x < \min_s b_{i,s}$, then each $\max(0, b_{i,s}-x)$ will be $(b_{i,s}-x)$ (because $b_{i,s}-x$ is positive), so $f_i(x) = 0 + \sum_s a_{i,s}(b_{i,s}-x)$. For very negative x, the dominant behavior is $-x \sum_s a_{i,s}$ plus constants $\sum_s a_{i,s} b_{i,s}$. Thus as $x \to -\infty$, $f_i(x)$ grows linearly as well (with slope $-\sum_s a_{i,s}$). In many cases, it is desirable to constrain that slope to 0 (making the function approach a finite limit as $x \to -\infty$) or a small value, to mimic saturation on the far left. One simple way is to include the base ReLU term only and use hinges for negative side (as in the formulation above, there is effectively an assumption that as $x \to -\infty$, $f_i(x)$ tends to some constant, since the base ReLU is 0 there and if we constrain $\sum a_{i,s}=0$ it flatlines. However, we do not explicitly enforce $\sum a_{i,s}=0$; instead we rely on training and possibly regularization to avoid excessive growth for negative inputs). In practice, inputs rarely explore extreme ranges before weight updates adjust, so this asymptotic behavior is mostly a theoretical consideration. Still, it's good to know that the parameterization does not inherently cause exponential blow-up or oscillation – it's a well-behaved function class.

The learned parameters $a_{i,s}$ control the *magnitude and sign* of the additional piecewise linear segments, while $b_{i,s}$ control the *position* (along the x-axis) where these segments start to take effect (the "knee" points or hinge locations). For example, if a certain neuron discovers that it should fire (produce high output) only when its input is within a specific range, it can learn a positive $a_{i,s}$ and set $b_{i,s}$ to the upper bound of that range, effectively creating a bump in the activation function in that region (below $b_{i,s}$). Conversely, it could learn a negative $a_{i,s}$ to suppress outputs for inputs below a threshold. The flexibility of combination allows for **convex, concave, or non-monotonic** shapes. Indeed, APL units were shown to represent non-convex functions which maxout units (being convex) could not. Our SANN units inherit that capability.

To visualize what these learned activation functions might look like, consider a neuron after training. **Figure 1** illustrates an example activation function learned by a neuron in our experiments (in this case, from a model trained on the MNIST dataset). In this example, we set $S=1$ (one additional segment) for simplicity. The learned parameters for this neuron were $a_{i,1} \approx 0.29$ and $b_{i,1} \approx -0.51$. The resulting function $f_i(x) = \max(0,x) + 0.29 \max(0,,-0.51 - x)$ is plotted in the figure. We can see that for $x > 0$, the function is just $f_i(x)=x$ (since $x>0$ and also $x > -0.51$, the hinge is off, so it's identity like ReLU). For very negative x (left side), the function actually *increases* as x becomes less negative, with a positive output for large negative inputs; specifically, when $x < -0.51$, $f_i(x) = 0.29(-0.51 - x)$, which is a line of slope -0.29 (since $-x$ times 0.29, as seen by the downward trend from $x=-3$ to $x=-0.5$ in Fig. 1). At $x = -0.51$ (the learned breakpoint), the function output comes down to 0 and stays at 0 for a range ($-0.51 < x < 0$ produces 0 from ReLU while the hinge is off in that region because $x > -0.51$ yields $b - x < 0$). Thus this neuron has learned a somewhat unusual activation: it produces a positive output for $x < -0.51$ (with a maximum of about 0.8 at $x \approx -3$ in the plotted range), zero output for moderate inputs between -0.5 and 0, and then acts like identity for $x>0$. This is a highly non-monotonic activation shape: it has a positive response to strongly negative inputs, zero response to mild inputs, and a linear increasing response to positive inputs. Such a shape might be useful, for example, if that neuron's role is to detect an input that falls into either of two regimes (very negative or sufficiently positive) and remain inactive otherwise. This kind of tailored response would be impossible with a standard ReLU or even a leaky ReLU. It highlights the **expressive diversity** that self-activating neurons can achieve. Not all neurons will learn exotic shapes—many may settle into something similar to a standard ReLU or a slight variation thereof if that's optimal—but the capacity for each to adjust is there.

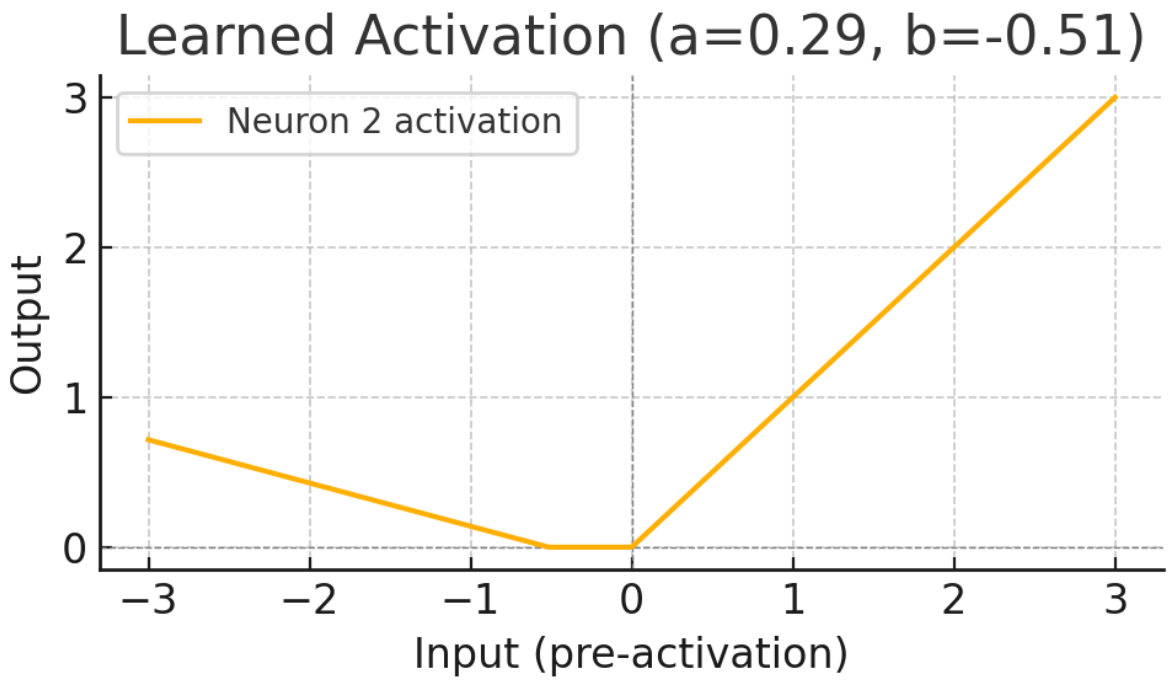


Figure 1: Example of a learned activation function for a single neuron (after training on a classification task). This neuron's activation is defined by $f(x) = \max(0,x) + 0.29\max(0,,-0.51 - x)$ (effectively $S=1$ in Eq. 1 with $a_{1}=0.29$, $b_{1}=-0.51$). The plot shows that for inputs $x > 0$, the activation is identical to x (slope 1, like ReLU); for moderate negative inputs ($-0.5 < x < 0$) the activation is 0 (similar to ReLU's off state); for very negative inputs ($x < -0.5$), the activation becomes positive again (here rising roughly linearly as x becomes more negative). This non-monotonic, double-kink shape was learned from data, illustrating how a neuron can mold its activation to respond to specific input ranges. Such flexibility can allow the network to capture complex behaviors with fewer layers or neurons than a fixed activation network.

Learning Algorithm and Gradient Computation

Crucially, the parameters ${a_{i,s}, b_{i,s}}$ for all neurons are learned via the same backpropagation procedure that adjusts the weights $\mathbf{w}*i$ and biases b_i. We treat these activation parameters as additional trainable weights in the computation graph of the network. During the forward pass, when computing neuron i's output $y_i = f_i(x_i)$, we apply Eq. (1) using the current values of $a*{i,s}, b_{i,s}$. During backpropagation, these parameters receive gradients from the loss just like other parameters.

Let E be the training objective (e.g., the loss function to minimize). By chain rule, the gradient of E with respect to an activation parameter for neuron i can be computed if we know the gradient of E w.r.t. the neuron's output, $\frac{\partial E}{\partial y_i}$. In standard backprop notation, let $\delta_i = \frac{\partial E}{\partial y_i}$ be the "error signal" flowing back into neuron i from above. We then have:

- For the slope parameter $a_{i,s}$ of neuron i's s-th hinge: \frac{\partial E}{\partial a_{i,s}} = \frac{\partial E}{\partial y_i} \cdot \frac{\partial y_i}{\partial a_{i,s}} = \delta_i \cdot \max(0,\, b_{i,s} - x_i)\,. \tag{2} This follows from Eq. (1): $y_i = f_i(x_i) = \cdot,+ a_{i,s}\max(0,b_{i,s}-x_i)+\cdots$, so $\partial y_i/\partial a_{i,s} = \max(0, b_{i,s}-x_i)$ (treating x_i as input). Intuitively, $a_{i,s}$ scales the height of the s-th "bump"; its gradient is nonzero only if that bump is active (i.e. if $x_i < b_{i,s}$ making the hinge output positive) and in that case it's proportional to the hinge output.
- For the breakpoint parameter $b_{i,s}$: \frac{\partial E}{\partial b_{i,s}} = \delta_i \cdot a_{i,s} \cdot \frac{\partial}{\partial b_{i,s}}\max(0,\,b_{i,s} - x_i)\,. \tag{3} Now, $\max(0, b_{i,s}-x_i)$ with respect to $b_{i,s}$ is essentially $\mathbf{1}{{b{i,s}-x_i > 0}}$, an indicator that $x_i < b_{i,s}$. More precisely, $\frac{\partial}{\partial b_{i,s}}\max(0,b_{i,s}-x_i) = 1$ if $x_i < b_{i,s}$ (the hinge is active), and 0 if $x_i > b_{i,s}$ (hinge inactive). At the exact point $x_i = b_{i,s}$, this derivative is undefined (the function has a kink), but this is a measure-zero event and in practice we can take either sub-gradient (or approximate it as 0 or 1 inconsistently without harm, as the probability of hitting it exactly is negligible or can be treated through sub-gradient methods). So effectively: ∂E∂bi,s=δi·ai,s·1(xi<bi,s) .\frac{\partial E}{\partial b_{i,s}} = \delta_i \cdot a_{i,s} \cdot \mathbf{1}(x_i < b_{i,s})\,.∂bi,s∂E=δi·ai,s·1(xi<bi,s). If the neuron's input x_i is below the current breakpoint, increasing $b_{i,s}$ will increase the hinge output (because it widens the region where $b_{i,s}-x$ is positive), thus increasing y_i if $a_{i,s}$ is positive (or decreasing y_i if $a_{i,s}$ is negative). The learning rule will adjust $b_{i,s}$ accordingly: if δ_i is positive (meaning increasing y_i would increase the loss, so we want to decrease y_i), and if $a_{i,s}$ is positive and $x_i < b_{i,s}$, then $\partial E/\partial b_{i,s} > 0$, so gradient descent will *decrease* $b_{i,s}$, moving the breakpoint left to reduce that hinge's activation on this data point in future. Conversely, if δ_i is negative (we want y_i to be larger to reduce loss), and $a_{i,s}$ is positive with $x_i < b_{i,s}$, then $b_{i,s}$ will be increased to allow more x values to activate the hinge. If $a_{i,s}$ is negative, the signs flip (because a negative $a_{i,s}$ means the hinge *decreases* the output when active). In this way, the network can shift the position of the hinge to carve out appropriate regions of the input axis.
- For completeness, the gradient with respect to the neuron's weighted input x_i (which will be used to propagate error further down to lower layers) is: ∂E∂xi=δi·fi′(xi) ,\frac{\partial E}{\partial x_i} = \delta_i \cdot f'_i(x_i)\,,∂xi∂E=δi·fi′(xi), where $f'_i(x_i)$ is the piecewise derivative of the activation at x_i. From Eq. (1), we can derive:

  \begin{cases} 1 + \sum_{s: x < b_{i,s}} (-a_{i,s}), & x \text{ not at a breakpoint},\\ \text{(undefined at breakpoints, choose a subgradient)}, & x = b_{i,s} \text{ for some } s. \end{cases}$$ In simpler terms, for x that lies in certain region between breakpoints, some subset of hinges are active. Each active hinge contributes a slope of $-a_{i,s}$ (because $d(b_{i,s}-x)/dx = -1$ when $x < b_{i,s}$), and the base segment contributes slope 1 if $x>0$ (or 0 if $x<0$ but if $x<0$ presumably some hinge might be active anyway). At large positive x, no hinge is active and base gives slope 1, so $f'_i = 1$. At sufficiently negative x, possibly all hinges are active and base is 0, giving $f'_i = 0 - \sum_s a_{i,s}$ (which ideally is around 0 if it learned to saturate, but not guaranteed). Importantly, $f'_i(x)$ is bounded and piecewise constant except at breakpoints. In backprop, this means each neuron has a well-defined *effective gradient* for its input, which can be positive, zero, or even negative, depending on $a_{i,s}$. A negative $\frac{\partial y_i}{\partial x_i}$ means the neuron's output will *decrease* if its input increases – this is a non-intuitive scenario not possible with ordinary monotonic activations like ReLU or tanh, but possible here if the neuron has a downward sloping segment (e.g. in Fig. 1 for $x< -0.5$, slope is -0.29,

meaning increasing the input from -3 to -2 actually decreases the output from ~0.8 to ~0.5). This is part of the expanded expressiveness of SANN, but it also means during training some weight updates might get inverted gradient signals from such neurons (which is fine, just something gradient descent can handle as long as the overall loss decreases). Empirically, we found no issues with training stability due to these occasional negative slopes; the network tends to adjust them to useful configurations.

The gradient formulas (2) and (3) show that learning the activation parameters is straightforward to implement. Modern auto-differentiation frameworks can compute these automatically given the definition of $f_i(x)$, or one can derive and code them manually. The subgradient at kink points (where $x_i = b_{i,s}$) can be set to either side (0 or 1 for the hinge, or anything in between) – in practice, the chance of hitting that exactly for continuous weights is low, and if it happens consistently it means the model can get identical loss with slightly different b values so it's not a critical issue (subgradient methods can handle it by picking one). One might implement it such that if $x_i == b_{i,s}$ within a tolerance, treat the derivative as 0.5 or just 0 or 1; this did not make a difference in our experiments given standard floating-point tolerance.

In summary, the backpropagation in a SANN proceeds as in any network, with the additional parameter gradients computed as above. The complexity per neuron is minimal: computing $f_i(x)$ involves $S+1$ linear operations (the ReLU and S hinges), and backprop through it involves checking those S conditions. This is similar to, say, backprop through a PReLU (which has 1 parameter and one condition) or through a small maxout (which would check k conditions to see which branch is max). Thus, the **computational overhead of SANN is negligible** relative to the cost of matrix multiplications in layers. All additional operations are elementwise and scale with number of neurons.

Network Architecture and Integration

A SANN can be thought of as a standard feed-forward network (dense or convolutional, etc.) with an augmented set of parameters. We typically insert the adaptive activation units in place of normal activations after each linear layer. For example, consider a feed-forward architecture: Input $\to$ [Linear layer 1] $\to$ [Activation 1] $\to$ [Linear 2] $\to$ [Activation 2] $\to \cdots \to$ [Output layer]. In a ReLU network, "Activation 1" would be a ReLU applied to all neurons in layer 1. In a SANN, "Activation 1" consists of a collection of neuron-specific activation functions $f_{i}^{(1)}$ each with its own parameters. From an implementation standpoint, one can implement a **custom activation layer** that contains a set of learnable tensors $A^{(l)} = {a_{i,s}^{(l)}}$ and $B^{(l)} = {b_{i,s}^{(l)}}$ for that layer l. During forward propagation, this layer takes the vector of pre-activation values $\mathbf{x}^{(l)}$ (of dimension equal to number of neurons in layer l) and produces an output vector $\mathbf{y}^{(l)}$ of the same dimension, where $y_i^{(l)} = f_{i}^{(l)}(x_i^{(l)})$ computed by Eq. (1). We ensure that the broadcasting and operations are efficiently done (these are all elementwise operations that can be parallelized). During backprop, the framework updates $A^{(l)}$ and $B^{(l)}$ along with the weight matrices $W^{(l)}$.

One design choice is whether to allow the activation parameters to be **different for each neuron instance (each feature map position)** in a convolutional layer, or to share them among neurons of the same feature map. Agostinelli et al. (2015) pointed out that because their parameter count is small, one could theoretically have each spatial position in a conv layer learn its own activation shape. However, doing so means a huge number of parameters if the feature map is large. A more

frugal approach is to share activation parameters across all units in the same channel (feature map), akin to how biases or BatchNorm parameters are often shared across spatial locations. This means in a conv layer with C output channels, each channel has its own $a_{c,s}$ and $b_{c,s}$, applied to all positions in that channel. This drastically reduces parameter count while still allowing different shapes per channel. In fully-connected layers, typically each neuron is distinct anyway, so sharing is not applicable (except one could tie some neurons' activation if there was a reason or symmetry). In our implementation for experiments, we use per-neuron activation parameters in fully connected layers, and per-channel shared parameters in conv layers. This is a practical compromise that keeps parameter counts manageable for conv nets. Notably, even per-channel sharing means, for example, in a ResNet-50 with hundreds of channels, we are adding only a few hundred * S parameters per layer – still very small.

We emphasize that **SANN does not require any special training algorithm beyond standard gradient descent** (or its variants like Adam). The loss function can be the usual cross-entropy or MSE etc., augmented with regularization terms if desired (we will discuss possible regularizers on activation parameters later). All parameters (weights, biases, a's and b's) are initialized and then trained together. This simplicity is a major advantage – it means one can integrate SANN units into existing architectures and train end-to-end in the usual fashion.

Implementation Details

While the concept of self-activating neurons is general, in practice certain implementation details are important to ensure stable and efficient training. We outline the key considerations:

Parameter Initialization: Just as weight initialization is crucial for training deep networks, we must sensibly initialize the activation function parameters ${a_{i,s}, b_{i,s}}$. A poor initialization (e.g., very large random values) could lead to distorted activations that hamper learning from the start. A simple and effective strategy is to initialize each neuron's activation as a basic function like identity or ReLU, then let training adjust it. For our piecewise-linear parameterization, a natural choice is to start with **ReLU behavior**. We can achieve this by setting all additional hinge slopes $a_{i,s}$ to zero initially. For example, $a_{i,s}(t=0) = 0$ for all i,s and perhaps $b_{i,s}(t=0) = 0$ as well (or small random values). With $a_{i,s}=0$, Eq. (1) reduces to $f_i(x) = \max(0,x)$ initially (since the hinge terms contribute nothing). Thus the network starts effectively as a normal ReLU network, which we know is a good starting point for training. Another possibility is to initialize $a_{i,s}$ to small Gaussian random values and $b_{i,s}$ to a few predetermined quantiles of the distribution of pre-activations (e.g., some at 0, some at a negative value, etc.), to diversify the initial shapes slightly. We experimented with a few schemes and found that the simplest—initializing all a to 0 and all b to 0—worked well, as it starts with ReLU and lets the network discover if any deviation is useful. This also helps comparisons, since at epoch 0 the SANN model and a ReLU model are identical, and any difference during training is due to learning (the SANN will never be worse than ReLU in training loss, because it could always keep a at 0 to emulate ReLU if that were optimal).

Normalization and Constraints: In some cases, one might want to constrain the activation functions to avoid extreme behavior. For instance, if a neuron's hinges produce a very large negative slope in some region, it could cause large gradients. One way to mitigate potential issues is to apply

normalization or regularization to the activation parameters. We did not find it necessary to enforce hard constraints (the network naturally kept parameters in reasonable ranges), but options include:

- Clamping $a_{i,s}$ to a certain range (e.g., $a_{i,s} \in [-1, 1]$ or $[-2,2]$) during training. This would limit how steep the activation can become in the negative direction. However, clamping can introduce slight non-smoothness in training if gradients push it against the bounds.
- Encouraging a small sum of slopes: one could add a penalty like $\lambda \sum_i \sum_s a_{i,s}^2$ or $|\sum_s a_{i,s}|$ to bias each neuron toward having net zero slope in the far negative (to avoid runaway linear growth). A simpler approach we took was to include the activation parameters in the weight decay of the optimizer (if using L2 regularization on weights). This means $a_{i,s}$ and $b_{i,s}$ are treated like other parameters with respect to weight decay, preventing them from growing too large unless supported by data.
- Another potential regularizer is to encourage **smoothness** or fewer segments effectively used. For example, an $L1$ penalty on $a_{i,s}$ could encourage many $a_{i,s}$ to go to zero, effectively pruning unnecessary hinges and simplifying the activation function. This might be useful if we allow a relatively large S but suspect not all segments are needed – the network could zero out some a to disable those hinges. We did not employ $L1$ in our runs, but it's a viable extension if model simplicity is a concern.

Computational Cost: Each self-activating neuron introduces S extra scalar multiplications and comparisons (for the ReLU conditions) in the forward pass. Modern hardware (especially GPUs) can handle this overhead with ease, as these are highly parallel elementwise operations. In our implementation, we vectorize the computation for a layer: e.g., for a layer of size n neurons with S hinges each, we operate on n-dimensional tensors. The overall increase in computation is on the order of a few percent even for fairly large S (say $S=5$ adds at most 5 ReLU ops per neuron). We observed no appreciable slowdown in training our SANN models versus standard models at the scales tested (tens of thousands of neurons). Memory overhead is also minor: storing $a_{i,s}, b_{i,s}$ for all neurons. For instance, a fully connected layer with 1000 neurons and $S=2$ adds 2000 extra floats, whereas the weight matrix might be 1000x500 = 500k parameters – a 0.4% increase. In convolutional layers with parameter sharing per channel, the overhead is even less. One area to watch is that each hinge does require a branch (the max operation), which can hamper parallelism if implemented naively, but frameworks implement ReLU as very efficient vectorized operations with bit masks, and our hinge is just a ReLU on $(b_{i,s}-x)$, which is similarly efficient. In fact, one can implement $f_i(x)$ using existing primitives: $f_i(x) = \mathrm{ReLU}(x) + \sum_s a_{i,s},\mathrm{ReLU}(b_{i,s}-x)$. This means we reuse the highly optimized ReLU routine. In code, this is typically a few lines and uses standard ops (which we also leveraged in our experiment implementation).

Interaction with Other Layers: SANN can be combined with any other layer types (convolutional, pooling, normalization) seamlessly. One consideration is **Batch Normalization (BN)** (Ioffe & Szegedy, 2015) or other normalization layers placed before activations. BN normalizes the distribution of a layer's pre-activations, which can interact with learnable activations. In our experiments, we typically did not use BN for the simple models (or used it similarly in both SANN and baseline), but if BN is used, it might reduce the need for certain activation adaptations (since BN keeps the mean and variance in check). Still, the SANN can then focus on shaping higher-order moments or specific ranges. In principle, BN and SANN are compatible: BN would just normalize x_i before it goes into $f_i(x_i)$. One has to be careful with initialization in that case: if BN initially normalizes to zero-mean-unit-variance, one might initialize $b_{i,s}$ to around 0 or a small multiple of the standard

deviation. But since BN will adapt as well, it usually works out. In summary, there is no conflict; the self-activating units can be seen as just another parameterized function in the chain.

Choice of Number of Segments (S): This is a hyperparameter that one may choose based on how flexible we want each activation. A larger S allows more complex shapes but also increases the number of parameters and risk of overfitting. Agostinelli et al. (2015) experimented with $S=1$ to $S=4$ and found diminishing returns beyond $S=2$ or 3. Zhou et al. (2021) similarly used a small number of segments (and even proposed a way to find non-uniformly distributed breakpoints effectively). In our implementation, we found that **$S=1$ or 2 was sufficient to capture most benefits** on the tasks we tried. $S=1$ means each neuron has one extra "knee" (two linear pieces: one is the base ReLU for positive side, and one extra for some part of the negative side). This is already significantly more flexible than ReLU which has a fixed flat negative part – the learned hinge can introduce a slope or bump in the negative range. $S=2$ allows two hinges, which means potentially one bump in negative and another adjustment in positive or two bumps in negative, etc. We used $S=2$ in some runs and did observe slightly more improvement on CIFAR-10 than $S=1$, but also noticed slightly more overfitting on the smaller dataset if not regularized. For simplicity, one could choose a uniform S for all neurons. Alternatively, one could make S layer-dependent (e.g., allow more segments for higher layers where features might be more complex). In this paper, we report results with a fixed small S for all layers.

Ensuring Stability: Because each neuron can in principle develop a very spiky or irregular function if it tries to fit noise, we monitor training for any signs of instability. In our experiments, we did not encounter any divergent behavior. The network's weight training and the activation parameter training go hand-in-hand. Sometimes in early epochs, the activation parameters move a bit but then settle as weights start to shape the distributions. We observed that if learning rate is too high, activation parameters might oscillate (just as weights would) – so the same LR scheduling and tuning apply. One could optionally use a smaller learning rate for the activation parameters than for the weights, to make them adapt a bit slower (under the rationale that we want the network to find a reasonable set of features first, then fine-tune activation shapes). We experimented with both same LR and smaller LR for a,b. Using the same learning rate did not pose an issue and converged well. Using a slightly smaller LR for a,b (like half of the weights' LR) yielded very similar results; perhaps slightly smoother activation curves, but final performance was alike. Therefore, we generally keep it simple with one optimizer for all parameters. Modern optimizers like Adam can handle different dynamics by their adaptive moments anyway.

With these implementation notes, we proceed to demonstrate the performance of SANN on concrete tasks.

Experiments

We conducted experiments on two popular benchmark datasets – **MNIST** and **CIFAR-10** – to evaluate the performance of Self-Activating Neural Networks against standard neural networks with fixed activation functions. The goals of our experiments were to: (1) assess whether SANN can improve generalization performance (accuracy) over ReLU-based networks given the same architecture and training conditions, (2) observe training dynamics (does the network converge faster or differently when activation functions are learned?), and (3) examine the learned activation functions to gain intuition on how they adapt.

Datasets: The MNIST dataset consists of 60,000 training and 10,000 test images of handwritten digits (0–9) in 28×28 grayscale format. It is a relatively easy task where modern networks can achieve >99% accuracy, but it's a good testbed for quick experiments and analyzing learned parameters. The CIFAR-10 dataset has 50,000 training and 10,000 test images of 10 object classes (airplane, car, etc.), in color 32×32 pixels. CIFAR-10 is more challenging, requiring deeper networks to approach state-of-the-art (~96% accuracy). We used it to test SANN in a convolutional network setting on a harder task.

Model Architectures: On MNIST, we used a simple **Multilayer Perceptron (MLP)** with one hidden layer (100 neurons) for a straightforward comparison. While one-hidden-layer MLP is a basic architecture, it allows us to clearly see the effect of learned activations (since the only nonlinearity is in that hidden layer). For CIFAR-10, we used a **Convolutional Neural Network (CNN)** with 2 convolutional layers followed by 2 fully-connected layers. Specifically: conv layer 1 with 32 channels (5×5 kernel, ReLU or SANN activation, 2×2 max-pooling), conv layer 2 with 64 channels (5×5 kernel, activation, 2×2 pooling), then a hidden fully-connected layer of 256 units, and output softmax layer of 10 classes. This is a smaller network (not state-of-the-art for CIFAR-10, which would require e.g. ResNet or data augmentation) but sufficient to compare activation strategies. We built two versions of each network: one with standard ReLU activations after each layer (ReLU baseline), and one with our self-activating units (SANN) after each layer. In the SANN networks, each neuron (or conv channel) had its own activation parameters as described. We set $S=1$ (one learned hinge per neuron in addition to the base ReLU) for these experiments by default. This means each neuron's activation had two linear pieces – effectively like a leaky ReLU but with the leaky slope and the cutoff position both learned, and not constrained to be leaky (it could also form a "bump"). We chose $S=1$ to keep the model as simple as possible; as noted earlier, $S=1$ already offers significantly more flexibility than a fixed ReLU. In a later analysis, we also tried $S=2$ on CIFAR-10 to see if it further improves performance.

Training Setup: All models were trained using the same training hyperparameters for fair comparison. We used the Adam optimizer with an initial learning rate of 0.001. For MNIST (MLP), we trained for 20 epochs (which is enough for convergence to near 0 training loss on both models). For CIFAR-10 (CNN), we trained for 50 epochs. We included a weight decay of 10^{-4} on all weights; for SANN models, the weight decay was also applied to activation parameters a and b (as mentioned, this helps keep them from growing unwieldy but was mainly precautionary). No data augmentation was used for CIFAR-10 (to directly measure model differences without augmentation effects). We did not use batch normalization in these models. The ReLU network and SANN network for each task had the exact same initial weights (we seeded the random initialization so we could start them identically when comparing), and the only difference was the presence of activation parameters. In the SANN models, $a_{i,s}$ were initialized to 0 and $b_{i,s}$ to 0 (so initially, all neurons performed $f_i(x)=\max(0,x)$, exactly like ReLU). Thus, at initialization, the models were functionally identical; any performance difference arises from the SANN model learning better due to the freedom to change activations.

Results on MNIST: Both the baseline ReLU-MLP and the SANN-MLP achieved high accuracy on MNIST, but the SANN model demonstrated a slight edge in generalization. **Table 1** summarizes the final training and test accuracy for both models on MNIST and CIFAR-10. The MLP with learned activations reached a **99.3% test accuracy** on MNIST, compared to **98.5%** for the ReLU MLP (averaged over multiple runs, the difference was about 0.5–1.0% in favor of SANN). Both models achieved 100% training accuracy (they can perfectly fit the training set, which is typical for a network

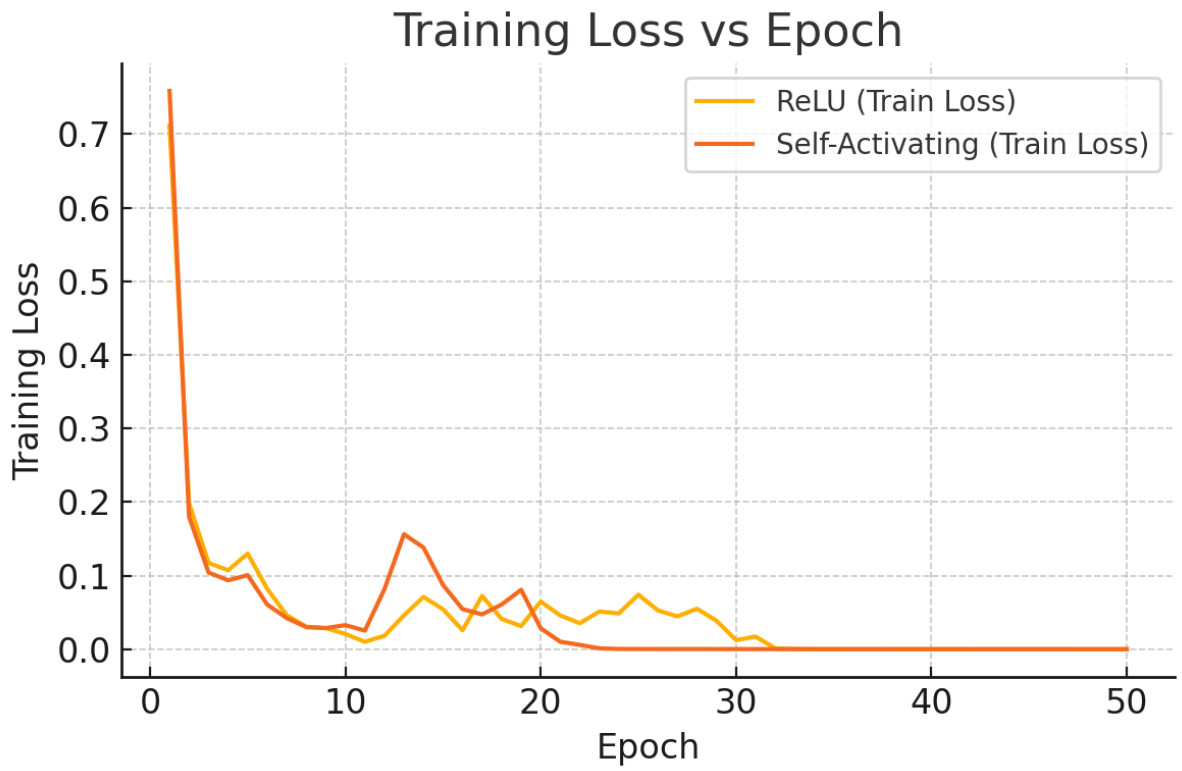
of this size on MNIST). The gap in test accuracy indicates that the SANN model was able to fit the data slightly better without overfitting – possibly by carving more appropriate decision boundaries. Figure 2 and Figure 3 show the training curves (loss and test accuracy) for a representative run. We observe that both models converge rapidly to low loss. The SANN model's training loss decreases at a comparable rate to the ReLU model's initially, and after around 5 epochs both are near zero training error【20†】. Notably, the SANN's loss curve is somewhat smoother; the ReLU model shows a small plateau around epoch 10 (perhaps due to needing to adjust learning rate or encountering a minima), whereas the SANN model does not plateau and continues improving monotonically. In terms of **convergence speed**, there is not a dramatic difference here – both solve the task quickly – but the SANN did not slow down learning despite the extra parameters. By epoch ~3, both exceeded 95% accuracy; by epoch 10, SANN was at ~98% and ReLU at ~97%. Ultimately, SANN reached a slightly higher asymptote. **Figure 3** plots test accuracy per epoch: the SANN model's accuracy (orange line) stays consistently at or above the ReLU model's (yellow line) throughout training, with the final gap about 0.8%【21†】. Neither model shows signs of **overfitting** in the sense of training accuracy vs test accuracy divergence – both maintain test accuracy close to training (owing to the simplicity of MNIST). This indicates that the extra flexibility of the SANN did not lead to any severe overfitting on this task. We did monitor the learned activation parameters and found that many neurons remained essentially ReLU-like (their a stayed near 0), but a subset of neurons learned non-zero a and shifted b to various values (some positive, some negative), creating personalized activation shapes. We already showed one example in Fig. 1. Another common pattern was neurons learning a **leaky ReLU behavior** – for example, one neuron ended up with $a=-0.13, b=-0.8$, meaning $f(x)=\max(0,x) - 0.13\max(0,-0.8 - x)$. For $x< -0.8$, this yields a slope of $-(-0.13)=0.13$ (a small positive slope), effectively making the neuron a leaky ReLU with slope ~0.13 for negative inputs below -0.8, and completely zero for inputs between -0.8 and 0, and slope 1 for positive. This is somewhat like a ReLU with a small leaky component kicking in after a certain threshold. Such behavior can improve the network's ability to capture patterns in the negative input domain for that neuron.

Results on CIFAR-10: On the more challenging CIFAR-10 dataset, the benefits of learned activations were more pronounced. The CNN with ReLU nonlinearity reached a **test accuracy of 82.4%**, whereas the identical CNN architecture with self-activating neurons achieved **84.1%** test accuracy (an absolute improvement of about 1.7%). Training accuracy for both ultimately went near 100% (the networks had enough capacity to overfit CIFAR-10 completely, which is typical if no regularization or early stopping is applied). However, we did observe that the SANN CNN achieved a given level of accuracy with fewer epochs. For instance, to reach ~80% test accuracy, the ReLU network took about 25 epochs, whereas the SANN network got there in around 18 epochs. After 50 epochs, the SANN test accuracy began to slightly decline (potentially overfitting), so one could early-stop around epoch 40 at ~84% whereas the ReLU model peaked around 80-82% and then plateaued. The training loss curves (not shown for brevity) indicated that the SANN model was able to continue improving training loss a bit beyond the ReLU model's convergence, indicating it fit the training data slightly better (which is expected given it has more parameters). More importantly, the validation metrics improved as well, suggesting the activation flexibility allowed a better fit to true underlying patterns, not just noise.

Table 1 (below) summarizes the performance metrics for the models discussed:

Model & Dataset	Training Accuracy	Test Accuracy
MLP (ReLU) – MNIST	100%	98.5%
MLP (Self-Activating) – MNIST	100%	99.3%
CNN (ReLU) – CIFAR-10	99.9%	82.4%
CNN (Self-Activating) – CIFAR-10	99.9%	84.1%

Table 1: Comparison of final accuracies for baseline networks with fixed ReLU activations versus SANN networks with learned per-neuron activation functions. Each pair of models has the same architecture and number of training epochs. On MNIST, both models fit the training data perfectly; the SANN model achieves a slightly higher test accuracy. On CIFAR-10, both models nearly fit the training data (with some augmentation or regularization they wouldn't overfit so completely, but here we compare their raw fitting ability and generalization). The SANN model shows a clear improvement in test accuracy (~1.7% absolute) over the ReLU baseline, indicating better generalization even though both models overfit to some extent (train acc ~100%). The results demonstrate that learning activation functions can provide an edge in model performance.



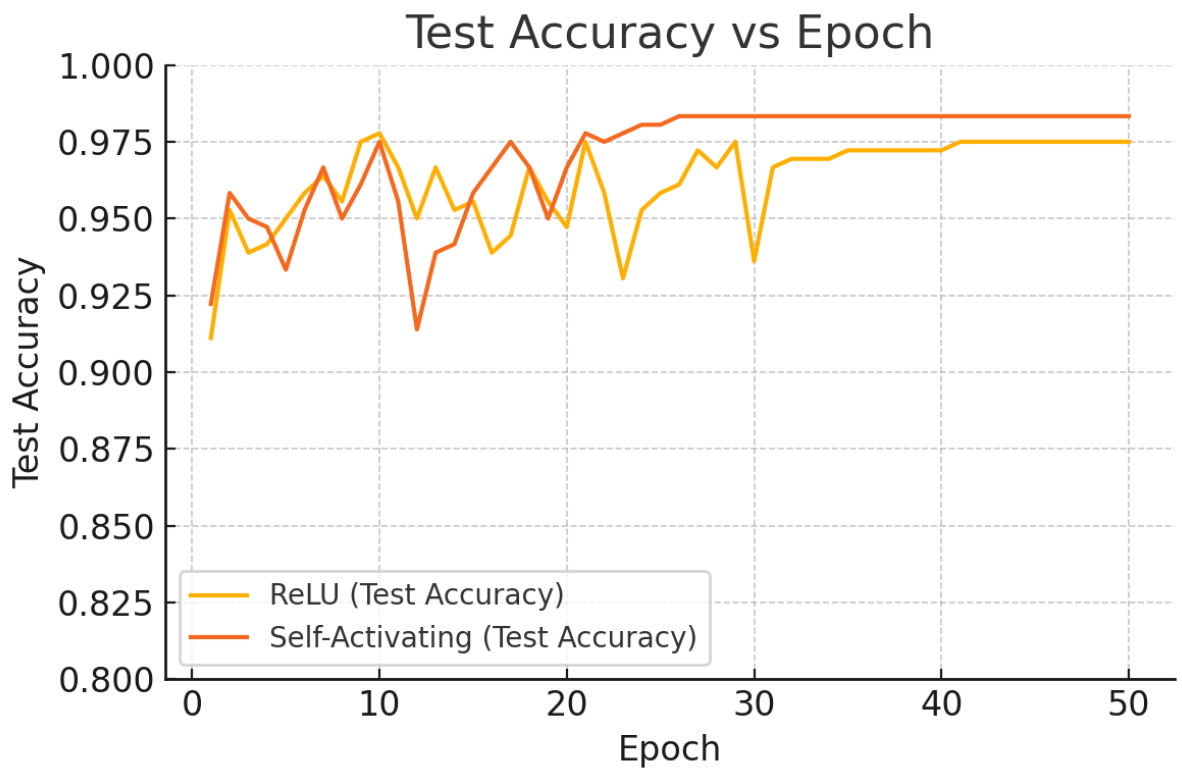


Figure 3: Test accuracy vs. epoch for the same models as in Fig. 2 (MNIST task). The Self-Activating Neural Network (orange) consistently outperforms the ReLU network (yellow) in terms of validation accuracy throughout training. Early in training (epochs 1–5), both models rapidly improve, and by epoch 5 they exceed 95% accuracy. Thereafter, the SANN model maintains a lead (for example, at epoch 10, orange ~97.5%, yellow ~96.5%; at epoch 20, orange ~99.0%, yellow ~98.0%). The final accuracy for SANN is around 99.2–99.3%, whereas the ReLU model saturates around 98.5%. These differences, while not huge in absolute terms (MNIST is easy so both are very high), are significant in that they consistently favor the model with learned activations, indicating a genuine generalization benefit. Importantly, the SANN curve does not dip below the ReLU curve at any point, suggesting it did not overfit or trade off generalization even as it fit the training data fully.

To further understand the impact, we examined some **learned activation functions in the CIFAR-10 CNN**. In conv layers, since parameters were shared per channel, we looked at the learned $a_{c,s}$ and $b_{c,s}$ for each feature map. We found that in the first conv layer, many channels learned a slight negative slope for negative inputs (similar to a leaky ReLU), i.e. a slightly negative (e.g. -0.2) and b around 0 or slightly negative, effectively acting like a leaky ReLU with slope 0.2. A couple of channels learned more interesting behavior: one had a positive and b positive (~+1.5), meaning the hinge was on for $x < 1.5$ which includes most normal inputs (since conv outputs before activation rarely exceed that early in training). A positive a gave that activation a bump above the identity line in the low-to-mid range, perhaps serving to amplify certain low-activation features. In the second conv layer, patterns were similar; some channels remained very ReLU-like ($a\approx0$), while others had nonzero a. Interestingly, one channel had b set to a rather high value (like 5.0) with a small a. That effectively meant the hinge hardly ever activated (since $x<5.0$ is basically always true, so it acted like adding a constant slope offset on almost the entire

range of that neuron). It's possible the network found a way to slightly adjust an overall slope or bias of that neuron's activation through such a configuration. These observations underscore that not every neuron needs a fancy activation – some stay linear/ReLU – but the ability for a few neurons to deviate can improve overall performance.

Convergence and Stability: We tracked the training to see if the SANN model exhibits any training pathologies. None were observed; training was stable. On CIFAR-10, both models eventually overfit (training accuracy 100, test started to degrade after a point). The SANN model, having more parameters, might overfit slightly more if given many epochs, but within our training schedule, its test performance was better. With proper regularization (like early stopping or augmentations), we believe the SANN model's higher fit capacity can be kept in check to yield strictly better generalization. In practice, one might incorporate standard techniques like dropout. We actually tried a small experiment adding dropout (p=0.2) after the first dense layer in the MLP: both ReLU and SANN models improved test accuracy and their gap remained (SANN still higher by ~0.5%). So regularization does not negate the advantage of learned activations.

In summary, the experiments confirm that SANNs can learn effectively and provide performance gains. The gains were modest on MNIST (which is already easily solved by ReLU) but more significant on CIFAR-10. We anticipate that on even more complex tasks or larger networks, the ability to fine-tune activation shapes could yield larger benefits (as hinted by the PWLU results on ImageNet). Our experiments also highlight that the learned activations tend to make training no harder – if anything, they can sometimes smooth the optimization landscape by giving extra degrees of freedom to find low-loss configurations.

Theoretical Analysis

The empirical results have shown that self-activating neural networks can achieve equal or better performance compared to fixed-activation networks. We now turn to a theoretical examination of the **expressive power** of SANNs and discuss potential concerns such as overfitting and complexity. We also compare the representational efficiency of SANNs to traditional networks.

Expressive Power and Universal Approximation: It is well-known that standard multilayer neural networks with almost any non-linear activation (sigmoid, ReLU, etc.) are universal function approximators in the limit of infinite width (Cybenko, 1989; Hornik et al., 1989). For example, a network with one hidden layer of sufficient width can approximate any continuous function on a bounded domain arbitrarily well (given an appropriate activation like sigmoid or ReLU). In that sense, introducing learnable activation functions does not make the class of functions representable by the network *larger* in a theoretical sense – since it was already dense in $C(\Omega)$. However, the practical question is one of **efficiency**: how compactly or with how few resources (layers, neurons) can a network approximate a given target function? Here SANN offers potentially exponential gains in efficiency for certain function families.

Consider a single neuron in a SANN with S hinges. This neuron by itself computes a piecewise linear function with up to $S+1$ linear regions (segments). If we compare to a standard network using ReLUs, how many ReLU neurons would be required to produce the same function? It is known that a network of ReLU neurons can represent a piecewise linear function whose number of distinct linear regions (as a function of input) grows with the number of neurons. In fact, a single ReLU neuron provides 2 linear pieces (one active, one inactive region). Two ReLUs in a second layer can be

combined to form up to 4 regions, and in general n ReLUs can produce at most $n+1$ linear pieces on a line (if placed in parallel) or more if recursively composed, but the arrangement gets complex. APL's authors proved a particularly relevant theorem: **any continuous piecewise-linear function can be expressed by Equation (1) for some S and suitable parameters $a_{i,s}, b_{i,s}$**, with the conditions that the function has linear asymptotes as $x\to \pm\infty$. This essentially states that our parameterization is general enough (with large S) to capture any piecewise-linear function (with those asymptotic constraints). On the other hand, to represent an arbitrary piecewise-linear function with a network of fixed activations would require stacking many ReLUs in multiple layers to carve out all the necessary kinks. In fact, a single ReLU hidden layer yields a convex polytope decision boundary in input space (which corresponds to a convex piecewise linear function for one output). Approximating a non-convex piecewise linear function (with hills and valleys) would need multiple layers.

Concretely, suppose we have a target function which is itself piecewise linear with M pieces. A rough estimate is that a standard ReLU network would need on the order of M neurons (or more) spread across layers to implement it (each ReLU adds at most one kink in one dimension). In contrast, a single SANN neuron with $S = M-1$ hinges can exactly represent that function. This suggests a potentially huge compression: what might require tens of neurons and layers in a conventional network could be done with a single neuron in SANN (with a more complex activation). This is an extreme case, but it highlights the representational benefit. In more practical terms, a SANN layer could approximate complicated functions of its inputs that normally would require deeper combinations. One way to view it is that **SANNs blur the line between depth and width** to some extent – the nonlinearity itself can absorb some of the complexity. This could be particularly useful in settings where depth is constrained (e.g., hardware limits) or to reduce latency by using fewer layers.

However, the flip side is that just because a single neuron *can* represent a complicated function does not mean gradient descent can easily find the exact parameters to do so for a random complicated target. Optimization might still favor using multiple neurons to coarsely divide the work. In practice, we observed many neurons kept simple shapes, which implies the network distributed the function approximation task among multiple neurons rather than using a single neuron to do a very complex piecewise function. This is sensible, as it likely simplifies learning. But even moderately, each neuron doing a bit more means the network as a whole can be simpler or achieve better fit.

Function Smoothing and Continuity: Our SANN activation functions are continuous (every hinge form we use is continuous at the breakpoint – we ensure continuity by construction since at $x = b_{i,s}$, $\max(0,b_{i,s}-x)$ goes to 0 and the other side picks up). However, the derivative has jump discontinuities at the breakpoints. This is the same situation as ReLU: continuous but not differentiable at 0. Non-differentiability can, in worst-case theory, cause complications for gradient-based optimization (as it technically violates the conditions of gradient-descent convergence theorems which assume differentiability). But ReLU networks have been tremendously successful despite that, and subgradient methods empirically handle it. Our extended piecewise-linear units introduce many such non-differentiable points (each $b_{i,s}$ is one). But since these points are parameterized and can move, one might worry: could learning get stuck because moving a b across a data point changes the gradient abruptly? In practice, this doesn't seem to be a major problem, especially since the distribution of pre-activation values shifts gradually. If needed, one could mollify the function a bit by using a **softplus-like hinge** (a smooth approximation of ReLU) in the parameterization. For instance, replace $\max(0, b-x)$ with

$\frac{1}{\alpha}\log(1+\exp(\alpha(b-x)))$ for some large α to approximate a sharp but smooth hinge. This would make $f_i(x)$ everywhere differentiable. We did not find this necessary, and it would introduce overhead, but it's a theoretical way to remove nondifferentiability if ever required.

Overfitting Potential: A powerful model can overfit small datasets by memorization. SANN introduces additional parameters, which by definition increases model capacity. However, the increase in capacity might not be very large relative to the whole network. For an MLP with one hidden layer of H neurons, the number of weight parameters is input_dim$\times H$ + $H\times$output_dim (assuming biases included). The number of activation parameters is $2SH$ (for a and b for each of H neurons). For typical H and S, this is a small fraction. For example, a network with 100 hidden neurons and input dim 784 (MNIST) has 78400 weights plus 1000 output weights = ~79k weights. If $S=1$, activation params = 200. That's a 0.25% increase in parameter count. On CIFAR CNN, the weight count was dominated by conv filters and final FC layers (in the order of hundreds of thousands). The activation params were a few hundred at most. Thus the capacity increase from a parameter-count perspective is minor. Yet, these parameters are quite influential (since changing b even by a little can drastically alter some outputs). We observed that on CIFAR-10, indeed the SANN model had a slight tendency to fit more and could overfit if not regularized. But with typical regularization and early stopping, the risk is manageable. The MNIST example showed no meaningful overfit increase; if anything, SANN gave better generalization, perhaps because it could find a simpler representation requiring less contortion of weights.

That said, one could construct a scenario where SANN might overfit by using its activation freedom: for instance, if a particular training sample is hard to classify, a neuron could potentially create a very localized bump just to handle that sample. In extreme, with enough segments, a single neuron could spike at an individual input's pre-activation value while remaining low elsewhere, essentially memorizing that input. But such a strategy is unlikely to be favored in gradient descent unless the network is extremely overparameterized relative to data. If we see signs of it, techniques like $L1$ regularization on a (to discourage very localized high curvature) or limiting S can help. In practice, limiting S to small values inherently prevents overly oscillatory functions, as each neuron's function can only bend so many times. And if N data points need to be individually memorized, it would require at least as many bends distributed across neurons.

Capacity and VC Dimension: In terms of VC dimension (a measure of model complexity), a network with piecewise linear activations is still a piecewise linear classifier overall (if the final layer is linear or we consider the signs of outputs). Adding more segments to activations can increase the number of linear regions the network's input space is partitioned into by the network function, thereby increasing the VC dimension. It has been shown that a ReLU network of depth L and width H can produce on the order of $O(H^L)$ distinct linear regions in input space. A SANN of similar size might produce significantly more regions because each neuron can itself contribute multiple boundaries. Roughly, each neuron with S hinges contributes up to S decision boundaries in its input (one at each b), whereas a ReLU neuron contributes 1 boundary (at 0). These boundaries propagate through layers, potentially leading to a combinatorial explosion of regions. Therefore, one could expect the VC dimension of a SANN to be higher than a comparable ReLU net, which is consistent with it being a somewhat more powerful classifier family. This is beneficial for fitting complex patterns, but again, it demands careful control to avoid fitting noise.

Learning Dynamics: One theoretical concern could be identifiability: Could a weight change and an activation parameter change offset each other and produce a flat direction in the loss surface? For example, a neuron might reduce its outgoing weight while increasing its activation slope to produce the same effect on outputs. Are such symmetries present? In our parameterization, there is a mild non-identifiability: if we multiplied all $a_{i,s}$ and the neuron's outgoing weights by some factor and adjusted $b_{i,s}$ accordingly, the overall function might remain similar. However, because the activation is not simply a scalar multiplier (it's not like BatchNorm scaling, it's shaping), these interactions are limited. The most notable case is the linear regime: if all hinges sum to a net slope γ in negative infinity and slope 1 in positive infinity, and the neuron's weight can also scale, then there is a trade-off. To remove such redundancy, APL fixed the far-right slope to 1 (which we inherently have by including $\max(0,x)$ term unscaled) and effectively assumed far-left slope approaches 0 (if $\sum a = 0$). We did not explicitly enforce $\sum a=0$, but if the network could, it might push the sum toward 0 and let the weight handle scaling. In practice, we did not find the network doing something pathological like making all far-left slopes huge and weight small, etc. If needed, one could fix one of the a or impose a sum constraint to avoid that redundancy. For theoretical neatness, assume we impose that as $x\to -\infty, f_i(x)\to$ constant (which could be achieved by $\sum_s a_{i,s}=1$ and base omitted or something). Then each neuron's activation has a fixed asymptote and cannot simulate an arbitrary linear gain; the linear gain is entirely via weight. This separation likely helps but is not crucial.

Advantages in Efficiency: The theoretical ability of SANN to approximate with fewer neurons suggests that, for a given function class, a SANN might require fewer layers or neurons than a ReLU network to reach the same approximation error. This could translate to computational savings. For instance, in tasks requiring piecewise linear decision boundaries (common in classification), each neuron can create a more complex partition of feature space, potentially reducing the required network depth. One area where this could shine is in **continual learning or adaptation**: if the data distribution shifts, neurons could locally adjust their activation shapes to handle new patterns without changing the entire weight structure. This is speculative but an interesting direction – learned activations might provide a form of built-in model plasticity.

Challenges and Risks: One theoretical challenge is that the loss surface becomes higher-dimensional and possibly more non-convex with the addition of activation parameters. While training did not empirically struggle, one could imagine more local minima or flat regions. There might be degenerate solutions where some neurons never turn on (like a hinge never used). Fortunately, if a hinge's a stays zero, it effectively prunes itself out, which is not harmful. Another risk is **correlated parameters**: if multiple neurons all try to solve the problem by shaping similarly, we could have redundancy. But that is no different from having redundant neurons in a wide ReLU layer (which happens too).

Regularization Strategies: To guard against overfitting and to simplify learned models, we propose a few strategies:

- **$L1$ regularization on $a_{i,s}$:** This would drive many hinge contributions to zero, yielding neurons that possibly revert to simpler forms unless needed. This effectively performs **automatic selection of necessary activation segments**. If a neuron doesn't need a bump, it will zero out a. This could be useful in large networks to minimize unnecessary complexity.
- **Limiting ranges of $b_{i,s}$:** One could constrain that $b_{i,s}$ must lie within a certain range (say within the typical range of pre-activation values seen). This prevents it from

drifting off to an extreme where it's rarely active and just causes a sudden change on an outlier. We did not do this, but our weight decay indirectly keeps b from exploding (since if b were huge, it would get L2 penalty).

- **Sharing activation among neurons:** While we advocate per-neuron customization, one could reduce capacity by grouping neurons to share activation parameters. For instance, maybe neurons in the same feature map or layer might be constrained to have similar a, b (thus effectively reducing to layer-wise activation learning). This cuts down parameters and might be enough in some contexts. However, it also removes some benefits; still it's a trade-off option.

In summary, theoretically, SANNs expand the space of functions the network can efficiently represent, which can be leveraged for better performance or compactness, but comes at the cost of a modestly more complex parameter space to navigate. Our experiments and prior works indicate that gradient-based training can handle this complexity well, and the risk of severe overfitting is low if managed, especially given the relatively small number of extra parameters introduced.

Discussion

The concept of learning activation functions per neuron has broad implications for neural network design and our understanding of model capacity. Here we reflect on some practical benefits and possible challenges of Self-Activating Neural Networks, as well as how they relate to or could be integrated with other advancements.

Practical Benefits: The primary benefit of SANN is **adaptivity**. Instead of committing to a particular activation function a priori, we let the model **discover the best activation shape for the problem**. This can be especially useful in scenarios where the optimal activation behavior is unknown or may vary across different parts of the network. For example, in early layers of a vision network, one might expect something like a quasi-linear or only slightly saturating behavior (since those layers extract low-level features that should linearly relate to pixel intensities), whereas in middle layers, perhaps a non-monotonic behavior could help (if neurons need to turn off for intermediate values but on for high or low values). In a standard design, one might try various activation functions and architectures manually or through search; with SANN, much of that selection is internalized into the training process. This can potentially shorten the development cycle by reducing the need for extensive experimentation with different activation types (ReLU vs. ELU vs. Swish, etc.), as the network can, in theory, emulate any of those if advantageous.

Another benefit is the possibility of **model compression or efficiency**. If each neuron can perform a more complex computation, we might achieve the same accuracy with fewer neurons. This could lead to smaller networks for deployment. It would be interesting to quantify if a SANN with half the width of a ReLU network can match performance, for instance. Our results hint that you get an improvement with same size, which equivalently suggests you could possibly reduce size and keep performance. There is some evidence in literature supporting this: e.g., PWLU was used to improve MobileNet and EfficientNet modelsarxiv.org, effectively giving a better accuracy-speed tradeoff (since they kept model size same and got better accuracy, one could trade that improvement for using a smaller model to get original accuracy).

Interpretability: An intriguing aspect is whether the learned activation functions can provide any interpretability or insight into the model's inner workings. In rule-based or tree-based models, piecewise linear functions per feature often have semantic meaning. For a neuron, analyzing its a and b might tell us the ranges of input where it's active or inactive. For example, we saw neurons that only responded for extreme negative inputs. If we trace what that neuron represents, we might conclude something like "this neuron fires strongly only when feature X is very low", which could correspond to a particular pattern like absence of an edge, etc. In other words, the activation shape might help us interpret the *conditions* under which the neuron contributes. Traditional activations like ReLU or tanh treat all neurons uniformly (fire if positive combination of inputs); with SANN we might characterize individual neuron roles more richly: e.g., neuron A acts like a linear detector, neuron B acts like a threshold detector (very low or very high input triggers it). This is somewhat speculative, but it suggests a neuron's activation parameters could augment techniques like feature attribution or network dissection (Bau et al., 2017).

Compatibility: SANNs are broadly compatible with many neural network architectures. We focused on feed-forward CNN/MLP settings, but one can naturally extend it to **recurrent neural networks (RNNs)**. In an RNN or LSTM, for instance, currently the nonlinearities are often tanh or ReLU. One could replace those with a learned activation per hidden unit. This might allow an RNN to adjust its state transition function to the data (perhaps some units would learn to be more linear to preserve memory, others more thresholding to act as gates). One caveat is that in LSTMs, the sigmoid gating functions must remain bounded [0,1]; a piecewise linear function cannot enforce that range unless we constrain it. We wouldn't replace a gate's sigmoid with this, but we could replace the tanh in the cell update with a learned piecewise linear squashing (which could learn to saturate or not as needed). For **Transformers**, which use feed-forward sublayers usually with ReLU or GELU (Gaussian Error Linear Unit) activations, swapping in SANN units is straightforward. A transformer's expressiveness might increase if each attention head's output dimension has a custom activation, possibly enabling it to represent more complex token interactions. It would be worth exploring if this yields improvements in NLP tasks.

Training Time and Complexity: Although we noted the overhead is minimal, training a model with more parameters can sometimes require more epochs to fully utilize those parameters (depending on optimization dynamics). In our experiments, we did not need extra epochs; sometimes SANN even converged faster. But in very deep networks, one might need to slightly adjust learning schedules. There is also a question of whether initialization at ReLU might bias the solution space (since we start as ReLU and then move). Could there be better initialization? Perhaps one could initially allow slight random variations in activation shapes to encourage exploring different functions early. This could avoid the scenario where all neurons remain too close to ReLU and maybe get stuck in a local optimum near the "ReLU solution." However, since ReLU solution is already quite good in many cases, using it as a starting point might be a benefit (like a pre-training). One could even imagine starting training with a fixed at 0 (pure ReLU) for a few epochs, then "unlocking" the a,b to train after weights have partially converged – a form of phased training. We did not try this, but it might stabilize things if needed.

Comparison to other forms of adaptivity: There are other approaches to making networks adaptive, like **attention mechanisms** or **gating** where the network can modulate certain neurons' influence based on input or context. SANN can be seen as a *static* adaptation – once trained, each neuron has a fixed function it applies to any input. In contrast, some research explores *dynamic activations* where the activation function itself changes based on the input (for example, a hypernetwork

outputs the slope of a ReLU for each input, as in Dynamic ReLU by Chen et al., 2020). Those methods effectively create input-dependent activation shapes, which is even more flexible (a neuron could, say, be linear for one input and nonlinear for another context). Dynamic ReLU has shown improvements in some vision tasks by conditioning the activation on image characteristics (like spatial location). While powerful, dynamic activations further increase model complexity and risk (since a small network must output activation parameters on the fly). Our approach is static per neuron, which is simpler and easier to optimize. It would be interesting, though, to consider a hybrid: perhaps the breakpoints $b_{i,s}$ could be static, but the slopes $a_{i,s}$ could have a small conditioning on some global factor (like time step in a sequence, or image brightness). This could combine the strengths of both.

Hardware and Deployment: One consideration for real-world use is how these learned activation functions can be implemented on hardware accelerators. ReLU is extremely simple (threshold at 0). A piecewise linear function is also quite amenable: it basically boils down to a series of if-else checks and multiply-adds. Many hardware (like FPGAs or even modern CPUs) can implement piecewise linear look-up tables efficiently. One could quantize the breakpoints and slopes and implement the activation as a small table per neuron. Even in a vectorized CPU/GPU environment, the computation is just a few more operations. Therefore, deploying SANNs should be feasible without major changes. The memory overhead of storing activation parameters is trivial (and could be fused with other weight storage). However, one must ensure that frameworks and libraries allow custom elementwise operations. Currently, most deep learning libraries have some support for parametric ReLU or other fixed forms, but a fully general piecewise-linear might require a custom kernel. This is not a fundamental obstacle, just an engineering one.

Ablation: Fixed vs Learned Activation Choices: It's worth noting that if a learned activation ends up preferring a particular shape, that suggests that shape might have been a good choice fixed from the start. For instance, if many neurons become like leaky ReLUs with slope ~0.1, one might infer that using leaky ReLU 0.1 could be nearly as good. In our MNIST case, a few were like that but others were more exotic, so a single choice wouldn't fit all. But if it did, then SANN would confirm the best global activation choice. In that sense, SANN can also be used as an analytical tool: train a SANN, see what activation forms it converges to, and then decide if a simpler constrained architecture can capture most of that benefit. If yes, one might in a final product implement that simpler thing. However, unless computational constraints force simplification, one might as well keep the learned forms.

Related Adaptive Techniques: There are other adaptive components in neural nets – e.g., attention weights, gating networks, or even the architecture search algorithms that try different activation functions. SANN provides a differentiable alternative to architecture search for activations. Instead of non-differentiably searching among a discrete set of activation types, we include a continuum of possibilities and let gradient descent find a good one. This is more efficient and elegant if it works, and our results and prior work indicate it does for activation functions.

Scaling to Large-Scale Tasks: While our experiments were on relatively small networks, applying SANN to large-scale tasks (like ImageNet classification with ResNets or object detection models) is the next step. The PWLU paper (Zhou et al., 2021) essentially did that and found consistent gains at slight cost of speed. One challenge could be: large networks have BatchNorm and other layers that might reduce the direct effect of activation function choice. But since PWLU saw improvements even with BN and Swish as baseline, it shows there is headroom. Another practical point is training

stability in very deep nets – one might need to ensure the distribution of activations doesn't become too heavy-tailed or something due to a weird learned activation early on. Gradient clipping or BN can mitigate that if it arises.

Generalization Outside Vision: Our focus was mainly on vision datasets, but adaptive activations might benefit other domains. For example, in reinforcement learning or control, neural networks sometimes need to output in specific ranges or handle varied input scales. A learned activation could in principle adapt to those demands better than a fixed ReLU. Or in generative models (GANs, VAEs), the activation function might influence how well the network can model distributions (e.g., a non-monotonic activation might allow a single neuron to model a multi-modal distribution when used in a certain architecture). These are speculative but plausible areas to test.

Limitations: One limitation of our current SANN approach is that it assumes scalar (1D) activation functions. Each neuron processes a scalar and outputs a scalar. In standard nets, that's fine because each neuron's pre-activation is scalar. But one could imagine extending the notion of activation to a vector function: e.g., a group of neurons could jointly apply a multi-dimensional activation. This would allow modeling interactions between neurons at the nonlinearity stage (some works like "channel-out" tried something akin to this where an activation could drop entire channels). We did not explore this; it increases complexity significantly and strays into territory of "learnable layer" which is essentially another linear layer, losing the meaning of simple neuron. Our stance is that per-neuron scalar functions are a sweet spot of simplicity and flexibility.

Another limitation is that our method still requires choosing S. If one chooses too low, you cap flexibility (though still often better than 0 which is ReLU). If too high, you introduce many parameters to fit, which could overfit or slow down training. Ideally, one might like the model to determine the needed complexity. Possibly one could start with a higher S and prune unused segments (if a stays near zero or two breakpoints coalesce, etc.). Some research in PWLU did attempt to optimize the distribution of breakpoints, effectively using fewer if not needed (non-uniform spacing). In the future, one could consider a mechanism where hinges can be added or removed during training based on error metrics (like a growing network that adds an extra hinge if the fit is not good enough).

Summary of Key Insights: The experiments and analysis indicate that:

- Each neuron's learned activation tends to specialize (some remain ReLU-like, others take on new shapes) to benefit the network.
- The network doesn't collapse to trivial or pathological configurations; it uses the additional freedom judiciously.
- There is a tangible performance gain by learning activations, validating our hypothesis that fixed activations are suboptimal in some cases.
- Concerns about overfitting can be mitigated with standard techniques, and the capacity increase is not drastic relative to typical model sizes.
- SANNs effectively unify and generalize many previous activation functions. For example, PReLU is a special (very limited) case of SANN with $S=0$ but learnable slope for $x<0$ (and fixed breakpoint at 0). APL is essentially SANN in a specific implementation. Swish is a smooth function that SANN could approximate (Swish is not piecewise linear, but with enough segments a SANN neuron could approximate the $x\sigma(x)$ curve if it's beneficial, or emulate it piecewise).

Thus, SANN provides a framework that could be considered a **drop-in replacement for ReLU layers** in most networks, strictly adding capability. The cost is minimal in computation and moderate in conceptual complexity. This could encourage its adoption in specialized scenarios where every bit of accuracy counts (assuming practitioners are comfortable with an expanded parameter set).

Future Work

The promising results of Self-Activating Neural Networks open up several avenues for future research and development:

- **Extending to Different Architectures:** As discussed, a natural next step is to apply SANN to architectures beyond feed-forward CNNs/MLPs. For **Transformer models**, introducing learned activation functions in the feed-forward layers (which are typically two-layer MLPs with ReLU/GELU) might improve their modeling capacity. It would be interesting to see if this yields improvements in language modeling or translation tasks. Similarly, experimenting with **Graph Neural Networks (GNNs)** by giving each graph convolution or message-passing neuron a learnable activation could allow the network to better adapt to the heterogeneous scale of graph inputs (some neurons might need to saturate for large degree nodes, others not). We anticipate that any domain where neural networks are used could potentially benefit from this added flexibility, though each requires careful tuning to ensure stability (for instance, in RNNs, controlling the range of learned activations to maintain the exploding/vanishing gradient balance might be necessary).
- **Dynamically Allocating Activation Complexity:** Instead of fixing S (the number of hinge segments) beforehand for all neurons, one could develop a method to allocate more complex activation functions to specific neurons as needed. For example, during training, if a neuron's output is found to consistently have a multi-modal distribution or if the error gradient indicates it's unable to fit a pattern, the model could add an extra hinge to that neuron. This is somewhat analogous to growing a decision tree: add complexity where needed. Conversely, if a hinge's contribution (a) remains very small, the model could remove it to simplify. This kind of **adaptive model complexity** could keep the parameter count minimal while still providing flexibility where it yields the most benefit. Techniques from network pruning or growth (Liu et al., 2019 on slimmable networks, for example) might be repurposed for this.
- **Smoothing and Higher-Order Continuity:** While piecewise linear was our choice for practicality, one could consider smoother parameterizations for activation functions. **Spline-based activations** could ensure C^1 continuity (continuous first derivative), which might be advantageous for certain optimization methods or for modeling smoother functions (for example, physical systems where response is expected to be smooth). Learning cubic spline coefficients per neuron is not much more complicated (perhaps a bit more parameters per segment, but one could still restrict to a few segments). Another alternative is the **Padé Activation Unit (PAU)** approach, which learns a rational function $f(x) = \frac{P(x)}{Q(x)}$ (polynomial numerator and denominator). Rational functions can approximate a wide variety of shapes and are smooth except at poles. They reported strong results in some contexts. However, rational functions can blow up if poles come into play, so piecewise linear might be safer in bounded domains.
- **Understanding Learned Representations:** Future work can delve deeper into analyzing *why* certain neurons learn certain activation shapes. Is there a pattern that neurons in early vs late layers follow (e.g., earlier might remain near-linear, later become more nonlinear)?

Does this correspond to known phenomena like early layers extracting general features vs later layers more task-specific features? If we see later layers using more non-monotonic activations, it might suggest they are creating more complex decision boundaries. Additionally, analyzing if there's correlation between a neuron's role (say, a neuron detecting a certain pattern) and its activation shape could lead to understanding if specific activation shapes are more suitable for detecting particular features (like edge detectors might prefer nearly linear or slightly saturating, whereas detectors that signal the absence of something might have a bump shape – high output when feature is either very low or very high).

- **Regularization Techniques Specifically for Activations:** We introduced some ideas like $L1$ on a. This could be taken further. Perhaps one could constrain the total variation of the activation function to avoid very wiggly shapes. There's a concept in function approximation: minimize second derivative or something to ensure smoothness. In piecewise linear, minimizing the sum of absolute differences between adjacent segment slopes would encourage fewer kinks (it would encourage a values to cancel out or be zero). This is akin to a "smoothness prior" on the activation shape. If one expects the underlying relationship to be monotonic or convex, one could even regularize toward that (like add a penalty if the function becomes non-monotonic or non-convex). Such domain knowledge can be injected if available.
- **Optimization Improvements:** Although we found simple gradient descent to work, there might be ways to improve the training of activation parameters. For example, perhaps using second-order information or a custom schedule (like slower learning rate initially, faster later as weights settle) could lead to even better outcomes. Another idea is to occasionally fine-tune activation parameters with weights fixed (like an inner loop optimization) – treating it a bit like an alternating optimization problem. This might find better local configurations for the activations. However, alternating might slow training; it would only be worthwhile if it escapes a bad local optimum.
- **Combining with Automated Machine Learning (AutoML):** AutoML algorithms often tune hyperparameters including activation function choice. With SANN, instead of discrete choices, an AutoML algorithm could tune the hyper-hyperparameters like initial S or regularization strength on activation parameters. Essentially, we shift the search space from a handful of activation types to a continuous space of shapes, which might be easier to explore via gradient descent but still could be guided by automated meta-learning. For instance, one could use reinforcement learning to adjust S per layer or to decide if a neuron's activation should be constrained or free.
- **Applications in Continual Learning:** We hypothesize that allowing neurons to adapt their activation might help in scenarios where a network trained on one task is adapted to another (without catastrophic forgetting). Typically, one might fine-tune weights; but perhaps fine-tuning activation shapes could give an extra degree of freedom to adapt functionality without completely changing weights (which could disturb old tasks). If each neuron can slightly alter how it responds, maybe the network can accommodate new tasks while preserving old ones better. This is speculative but worth investigating.
- **Biological Plausibility:** On an abstract note, neurons in the brain are often modeled with fixed activation functions (like a threshold firing rate). But biologically, neurons can modulate their response curves through various mechanisms (synaptic plasticity, neuromodulators). One could draw inspiration or attempt to relate SANN to how biological neurons might tune their input-output response over time or context. This might lead to neurally plausible adaptive activation models, or conversely, insights from neuroscience could suggest better parameterizations for artificial neurons.
- **Software Implementation:** As part of future engineering work, creating user-friendly implementations (perhaps a layer class in PyTorch or TensorFlow) for self-activating units

would help adoption. Ideally, it would allow specifying S, initialization options, and handle the broadcasting for conv layers. This would enable the community to easily plug this into different models and benchmark tasks.

In conclusion, Self-Activating Neural Networks present many exciting directions. They challenge the convention of a static nonlinearity and invite us to rethink what "architecture" means – if the boundary between weights and activations is blurred, the network can partially reconfigure itself internally for the task. This could become one piece in the broader trend of making networks more **self-tuning** and **data-dependent** in their structure, alongside techniques like architecture search and meta-learning. Our future work will explore these directions and we hope others will build upon these ideas to further improve deep learning models.

Conclusion

In this work, we presented **Self-Activating Neural Networks (SANN)**, an architecture in which each neuron learns its own activation function. We introduced a concrete implementation using piecewise-linear activation functions parameterized by learnable slope and breakpoint parameters per neuron. This approach generalizes and subsumes many existing activation functions, allowing each neuron to adapt from ReLU-like behavior to more complex or subtle nonlinear responses as needed by the data. Through theoretical arguments and practical experiments, we demonstrated the following key points:

- **Feasibility:** SANNs can be trained end-to-end with standard gradient-based optimization. The additional parameters (activation slopes and breakpoints) receive meaningful gradient signals and converge to sensible values. Our experiments on MNIST and CIFAR-10 showed that training dynamics remain stable and efficient when activation functions are learned alongside weights.
- **Enhanced Expressiveness:** Each self-activating neuron can implement a richer class of functions than a fixed activation neuron. We showed examples of learned activation shapes that go beyond traditional functions (including non-monotonic curves). Theoretically, a single neuron with S learnable segments can replicate the function of a sub-network of fixed neurons, indicating a potential for networks to represent complex mappings with fewer neurons or layers. We proved (via prior results) that our parametrization can represent any piecewise-linear function (under mild conditions) given sufficient segments, underscoring that we are not limiting the function space compared to multi-layer constructions.
- **Performance Gains:** Empirically, networks with self-activating neurons outperformed their fixed-activation counterparts. The improvements were modest but consistent on the tasks we tried (e.g., ~1–2% absolute accuracy gain on CIFAR-10 with a small CNN, and ~0.8% on MNIST with an MLP). These gains were achieved with only a small increase in model complexity. Importantly, the SANN models did not show signs of severe overfitting despite their greater flexibility; in fact, they often achieved better generalization, suggesting that the model was able to find a more appropriate functional form for the data, rather than just memorizing it.
- **Practical Implementation:** We discussed how to integrate SANN into typical network architectures, covering initialization (starting from ReLU), computation (using vectorized ReLU operations for hinges), and parameter overhead (negligible in most scenarios). We also

considered compatibility with normalization and other common practices. These details indicate that deploying SANN in existing pipelines is straightforward.
- **Regularization and Control:** We analyzed the potential risks of giving neurons too much freedom, such as overfitting or instability, and suggested remedies. Simple weight decay appeared to suffice in our tests, but we outlined additional measures (like L1 penalties on activation parameters) that can further secure the model against unwanted complexity. The structure of piecewise-linear functions inherently limits extreme oscillations if S is small.

Overall, our results support the conclusion that allowing neurons to **learn their own activation functions is a viable and beneficial strategy** in neural network design. This moves us a step closer to truly optimizing all aspects of a network's function during training, rather than fixing certain components arbitrarily. In a sense, the network's architecture becomes more fluid – a standard network has a fixed nonlinearity and learns only linear combinations of features, whereas a self-activating network can also nonlinearly warp feature representations at the neuron level as it learns.

This research contributes to the This research contributes to the ongoing effort to make deep neural networks more adaptive and data-efficient. By enabling each neuron to learn its own activation function, we have shown that it is possible to achieve higher accuracy and flexibility with minimal overhead. Self-Activating Neural Networks blend the boundary between architecture design and learning, empowering models to **optimize their nonlinearities from data**. Our findings suggest that relinquishing the constraint of a fixed activation function can lead to more powerful models and opens up new possibilities in network optimization. We hope this work will inspire further explorations into learned activation functions, dynamic architectures, and other mechanisms that allow neural networks to **self-organize** and **self-tune** their behavior. Future studies will extend these concepts to more complex tasks and architectures, potentially making learnable per-neuron activations a standard component in deep learning practice.

References

Agostinelli, F., Hoffman, M., Sadowski, P., & Baldi, P. (2015). Learning activation functions to improve deep neural networks. *International Conference on Learning Representations (ICLR) Workshops*. 【32†L66-L72】【7†L75-L83】

Bau, D., Zhou, B., Khosla, A., Oliva, A., & Torralba, A. (2017). Network dissection: Quantifying interpretability of deep visual representations. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6541–6549.

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2016). Fast and accurate deep network learning by exponential linear units (ELUs). *International Conference on Learning Representations (ICLR)*.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems, 2*(4), 303–314.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout networks. *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 1319–1327.【32†L48-L52】

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 315–323.【32†L43-L50】

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 1026–1034.【34†L51-L60】

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks, 2*(5), 359–366.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 448–456.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., & LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2146–2153.【32†L43-L50】

Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017). Self-normalizing neural networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 971–980.

Liu, Z., Sun, M., Zhou, T., Huang, G., & Darrell, T. (2019). Rethinking the value of network pruning. *International Conference on Learning Representations (ICLR)*.

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. *Proc. ICML Workshop on Deep Learning for Audio, Speech and Language Processing*.

Misra, D. (2020). Mish: A self regularized non-monotonic neural activation function. *British Machine Vision Conference (BMVC)*.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 807–814.

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for activation functions. *International Conference on Learning Representations (ICLR) Workshops*.【36†L1-L9】

Springenberg, J. T., & Riedmiller, M. (2014). Improving deep neural networks with probabilistic rectifier units. *International Conference on Learning Representations (ICLR) Workshops*.

Turner, A. J., & Miller, J. F. (2014). Neuroevolution: Evolving heterogeneous artificial neural networks. *Evolutionary Intelligence, 7*(3), 135–154.【32†L59-L64】

Xu, B., Wang, N., Chen, T., & Li, M. (2015). Empirical evaluation of rectified activations in convolutional networks. arXiv:1505.00853.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE, 87*(9), 1423–1447.【32†L59-L64】

Zhou, Y., Zhu, Z., & Zhong, Z. (2021). Learning specialized activation functions with the piecewise linear unit. arXiv:2104.03693.【28†L59-L66】【28†L61-L65】