Date of publication SEP-30, 2023, date of current version JUL-10, 2023
www.computingonline.net/ computing@computingonline.net

Print ISSN 1727-6209
Online ISSN 2312-5381
DOI 10.47839/ijc.22.3.3225

Learnable Extended Activation Function for Deep Neural Networks

YEVGENIY BODYANSKIY[1], SERHII KOSTIUK[2]

[1]Control Systems Research Laboratory, National University of Radio Electronics, Nauky av. 14, Kharkiv, 61166, Kharkiv, Ukraine, (e-mail: yevgeniy.bodyanskiy@nure.ua)

[2]Dept. of Artificial Intelligence, National University of Radio Electronics, Nauky av. 14, Kharkiv, 61166, Kharkiv, Ukraine, (e-mail: serhii.kostiuk@nure.ua)

Corresponding author: Serhii Kostiuk (e-mail: serhii.kostiuk@nure.ua).

ABSTRACT This paper introduces Learnable Extended Activation Function (LEAF) - an adaptive activation function that combines the properties of squashing functions and rectifier units. Depending on the target architecture and data processing task, LEAF adapts its form during training to achieve lower loss values and improve the training results. While not suffering from the "vanishing gradient" effect, LEAF can directly replace SiLU, ReLU, Sigmoid, Tanh, Swish, and AHAF in feed-forward, recurrent, and many other neural network architectures.

The training process for LEAF features a two-stage approach when the activation function parameters update before the synaptic weights. The experimental evaluation in the image classification task shows the superior performance of LEAF compared to the non-adaptive alternatives. Particularly, LEAF-as-Tanh provides 7% better classification accuracy than hyperbolic tangents on the CIFAR-10 dataset. As empirically examined, LEAF-as-SiLU and LEAF-as-Sigmoid in convolutional networks tend to "evolve" into SiLU-like forms. The proposed activation function and the corresponding training algorithm are relatively simple from the computational standpoint and easily apply to existing deep neural networks.

KEYWORDS Adaptive Hybrid Activation Function, Trainable Activation Function Form, Double-Stage Parameter Turning Process, Squashing Functions, Linear Units, Deep Neural Networks.

I. INTRODUCTION

DEEP Neural Networks (DNNs) have become a widely used tool for various data processing tasks [1]. DNNs found their application in image processing – recognition and classification, time series processing – extrapolation and forecasting, natural language processing (NLP) and video stream analysis, controlling complex systems and processes, monitoring and fault diagnosis, and many other areas.

One prominent feature of artificial neural networks (ANNs), deep and shallow, is their universal approximation capabilities [2] and their ability to "learn" from the training data. The definition of this "learning" process is a parameter optimization task. During training, the network adjusts its synaptic weights in each neuron and the overall architecture to optimize the selected learning criteria – the goal function.

Elementary perceptrons of F. Rosenblatt traditionally serve as the base building blocks of feed-forward artificial neural networks. The elementary perceptron uses squashing functions as its non-linearity elements, primarily – sigmoidal (σ-function), hyperbolic tangent (tanh), arctangent (arctan), Softsign, Satlin, and other functions. One common property of such squashing functions is their bounded change interval, with the output values heading to the corresponding bounds asymptotically.

Compared to feed-forward ANNs, recurrent networks like LSTM and GRU use some form of gated units [3], [4] – ANN blocks that simultaneously employ the sigmoid and hyperbolic tangent functions to control the flow of information in the network. Transformer networks with gating layers [5] and GLU activation functions [6] implement a similar approach to improve the training results.

While sigmoidal activation functions provide the network with universal approximation capabilities, according to [2], the squashing nature of such functions creates difficulties for the gradient-based optimization procedures. As the output value approaches the asymptotes, the corresponding deriva-

tive approaches zero, blocking backward propagation and stopping the training process. In other words, such networks are prone to the "vanishing gradient" effect. The severity of this undesired effect increases with the number of layers in the network.

In order to avoid the "vanishing gradient" effect, DNNs often employ activation functions from the rectifier units family, namely piece-wise activation functions such as ReLU, PReLU, LReLU, NReLU, ELU, and others [7].

Like with the trapezoidal rule in integration, the accuracy of piece-wise approximation depends on the number of segments, leading to a significant increase in the network's synaptic weights and layers compared to the alternatives with universal approximators. At the same time, an increase in the number of layers and parameters makes the network slower to train, requires additional computational resources, and, what is especially important in practical applications, requires a vast amount of task-specific training data, which is hard to collect.

Therefore, researchers develop new hybrid activation functions that combine the strengths of sigmoidal and piece-wise approximations, such as SiLU [8], Swish [9], S-shaped [10], WiG [11], AHAF [12], and others. Combining the strengths allows training using smaller data sets and reducing the total training time. Meanwhile, applying such hybrid functions requires careful selection and tuning of their free parameters, a non-formalized task typically based on empirical rules and a specific user's experience.

One possible idea for selecting free parameters in hybrid activation functions includes tuning the parameters during training [13]–[18]. The previously presented approach includes a two-stage process that tunes the synaptic weights first and the activation function parameters second. This approach improves the approximation capabilities of the network but increases the total training time.

Hence, [12], [19] introduce adaptive activation functions and the corresponding training algorithms that tune the activation function parameters together with the synaptic weights of individual neurons.

Meanwhile, the mentioned activation functions do not coincide with the classic sigmoidal functions and hyperbolic tangents, substantially limiting their approximation capabilities. Based on that, one reasonable idea is introducing a function and the corresponding ANN neuron that would combine the activation functions from shallow and deep neural networks.

This paper introduces Learnable Extended Activation Function (LEAF) and a method for tuning its parameters during training. LEAF combines the properties of rectifier units and squashing functions, working as a suitable replacement for ReLU, SiLU, the sigmoidal function, and hyperbolic tangent.

II. THE LEAF-BASED NEURON ARCHITECTURE

The elementary perceptron of F. Rosenblatt serves as the base ANN block and implements signal transformation of the following form:

$$\begin{aligned}\hat{y}_j(k) &= \psi_j\left(\theta_{j0} + \sum_{i=1}^{n} w_{ji}x_i(k)\right)\\ &= \psi_j\left(\sum_{i=0}^{n} w_{ji}x_i(k)\right),\end{aligned} \tag{1}$$

where j - index of the neuron, k - discrete data processing time step, $k = 1, 2, 3, ..., N, ...$, $y_j(k)$ - output signal of the j-th neuron on step k, $x_i(k)$ - i-th element of the input vector x on time step k, $i = 0, 1, 2, ..., n$, w_{ji} - synaptic weight on the i-th input of the j-th neuron, θ_j - threshold signal of the j-th neuron, $\theta j \equiv w_{j0}$, $x_0 \equiv 1$, $u_j(k)$ - the signal of internal activation in the j-th neuron, ψ_j - non-linear transformation on the output of the j-th neuron.

Or in a vector form for a specific neuron j:

$$\hat{y}_j(k) = \psi_j\left(w_j^T x(k)\right) = \psi_j\left(u_j(k)\right),$$

where j - index of the neuron, k - time step, $x(k)$ - the $(n+1)\times 1$-dimentional vector of the input signals, $x_0 \equiv 1$, w_j - the $(n+1)\times 1$-dimentional vector of weights, $u_j(k)$ - the signal of internal activation in the j-th neuron, $\psi_j(u_j(k))$ - non-linear transformation on the output of the j-th neuron.

Activation function ψ_j in the perceptron provides the non-linearity for approximation. We propose the learnable non-linear activation function of the following form:

$$\begin{aligned}\psi_j(u_j) &= (\rho_{j1}u_j + \rho_{j2})\sigma(\rho_{j3}u_j) + \rho_{j4} =\\ &= \frac{\rho_{j1}u_j + \rho_{j2}}{1 + e^{-\rho_{j3}u_j}} + \rho_{j4},\end{aligned} \tag{2}$$

where parameters ρ_{j1}, ρ_{j2}, ρ_{j3}, ρ_{j4} define the activation function form and get tuned during the neural network training.

Fig. 1 shows the architecture of a neural network neuron with LEAF, including the linear transformation and the activation function. The figure shows that the neuron with LEAF has $n+5$ trainable parameters, where n corresponds to the number of inputs, compared to $n+1$ in the classic neural network node.

Depending on the values of trainable parameters, LEAF can assume the shape and properties of other known activation functions. LEAF includes many known activation functions as its corner cases, including ReLU, SiLU, hyperbolic tangent, the sigmoidal function, and even adaptive functions, such as Swish and AHAF. Table 1 shows the LEAF parameter values that provide exact replacements or sufficiently close approximations of the previously known activation functions.

As the table illustrates, replacing non-adaptive activations with LEAF requires setting the LEAF parameters to the pre-determined constant values. Replacement of adaptive activations, such as Swish and AHAF, requires setting some LEAF parameter values to the corresponding values

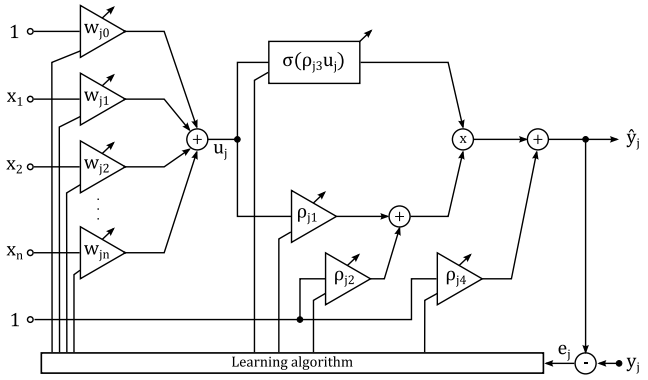

Figure 1. Neuron with Learnable Extended Activation Function (LEAF)

Table 1. Values of trainable parameters that transform LEAF to other known activation functions

Equivalent function	**LEAF parameter values**			
	ρ_1	ρ_2	ρ_3	ρ_4
ReLU	1	0	$+\infty$	0
SiLU	1	0	1	0
Tanh	0	2	2	-1
Sigmoid	0	0	1	0
Swish	1	0	ρ_3	0
AHAF	ρ_1	0	ρ_3	0

in the original activation functions. For example, AHAF parameters β and γ in [12] correspond to parameters ρ_1 and ρ_3 in LEAF.

III. THE LEAF-BASED NEURON TRAINING PROCESS

The training process depends on the loss function definition. Without losing generality, we demonstrate the training process using the traditional quadratic criterion:

$$E_j(k) = \frac{1}{2}e_j^2(k) = \frac{1}{2}(y_j(k) - \hat{y}_j(k))^2, \tag{3}$$

where $y_j(k)$ - the expected output of the j-th neuron, $\hat{y}_j(k)$ - the actual output of this neuron on time step k.

For an elementary perceptron of F. Rosenblatt, the loss defines as:

$$\begin{aligned} E_j(k) &= \frac{1}{2}(y_j(k) - \hat{y}_j(k))^2 = \\ &= \frac{1}{2}(y_j(k) - \psi_j(u_j(k)))^2 = \\ &= \frac{1}{2}\left(y_j(k) - \psi_j\left(\sum_{i=0}^{n} w_{ji}x_i(k)\right)\right)^2, \end{aligned} \tag{4}$$

so that the delta rule for synaptic weight w_{ji} on time step k equals:

$$\begin{aligned} w_{ji}(k) &= w_{ji}(k-1) - \eta_w(k)\frac{\delta E_j(k)}{\delta e_j(k)}\frac{\delta e_j(k)}{\delta w_{ji}} = \\ &= w_{ji}(k-1) + \eta_w(k)\delta_j(k)x_i(k), \end{aligned} \tag{5}$$

where $\eta_w(k)$ - the learning rate on time step k, $\delta_j(k) = e_j(k)\psi_j^{'}(u_j(k))$ - the delta-error value propagating between the layers during training in multi-layer networks. The delta-error value strongly depends on the activation function derivative, leading to the "vanishing gradient" effect and effectively stopping the backpropagation when $\psi_j^{'}(u_j(k))$ approaches zero.

The training process for the LEAF-based neuron uses a variant of the double-stage parameter tuning (DSPT) [12] procedure. The first stage includes updating the trainable activation function parameters $\rho_1, ..., \rho_4$ while the synaptic weights $w_{j0}, ..., w_{ji}, ..., w_{jn}$ remain fixed. The second stage includes updating the synaptic weights without tuning the activation function parameters but using their corrected values to calculate the loss.

The delta rule for activation function parameters $\rho_1, ..., \rho_4$ depends on the corresponding partial derivatives.

The partial derivative of the output signal $\psi_j(u_j)$ by ρ_1 on time step k is:

$$\begin{aligned} \frac{d\psi_j(u_j)}{d\rho_{j1}} &= \frac{d((\rho_{j1}u_j + \rho_{j2})\sigma(\rho_{j3}u_j) + \rho_{j4})}{d\rho_{j1}} = \\ &= \frac{d(\rho_{j1}u_j + \rho_{j2})}{d\rho_{j1}}\sigma(\rho_{j3}u_j) = \\ &= \frac{d(\rho_{j1}u_j)}{d\rho_{j1}}\sigma(\rho_{j3}u_j) = u_j\sigma(\rho_{j3}u_j), \end{aligned}$$

where j - index of the neuron, $\sigma(var) = \frac{1}{1+e^{-var}}$ - the sigmoidal function, u_j - the signal of internal activation, and time indexes k, $(k-1)$ are omitted for brevity.

Similarly, the definition of partial derivatives for ρ_2, ρ_3, ρ_4 on time step k is:

$$\begin{aligned} \frac{d\psi_j(u_j)}{d\rho_{j2}} &= \frac{d((\rho_{j1}u_j + \rho_{j2})\sigma(\rho_{j3}u_j) + \rho_{j4})}{d\rho_{j2}} = \\ &= \frac{d(\rho_{j2})}{d\rho_{j2}}\sigma(\rho_{j3}u_j) = \sigma(\rho_{j3}u_j); \end{aligned}$$

$$\begin{aligned} \frac{d\psi_j(u_j)}{d\rho_{j3}} &= \frac{d((\rho_{j1}u_j + \rho_{j2})\sigma(\rho_{j3}u_j) + \rho_{j4})}{d\rho_{j3}} = \\ &= (\rho_{j1}u_j + \rho_{j2})\sigma(\rho_{j3}u_j)\sigma(-\rho_{j3}u_j)u_j; \end{aligned}$$

$$\frac{d\psi_j(u_j)}{d\rho_{j4}} = \frac{d((\rho_{j1}u_j + \rho_{j2})\sigma(\rho_{j3}u_j) + \rho_{j4})}{d\rho_{j4}} = \frac{d\rho_{j4}}{d\rho_{j4}} = 1.$$

With the partial derivatives defined, the delta rule for parameters $\rho_1, \rho_2, \rho_3, \rho_4$ on time step k on the first stage of the DSPT procedure gets the following form:

$$\begin{aligned}\rho_{j1}(k) =&\rho_{j1}(k-1) - \eta_{\rho_1}(k)\frac{\delta E_j(k)}{\delta \rho_{j1}} = \\ =&\rho_{j1}(k-1) - \eta_{\rho_1}(k)e_j(k)\frac{\delta \psi_j(k)}{\delta \rho_{j1}} = \\ =&\rho_{j1}(k-1) + \eta_{\rho_1}(k)\cdot \\ &\cdot (y_j(k) - \psi_j(u_j(k)))\cdot \\ &\cdot u_j(k)\sigma(\rho_{j3}(k-1)u_j(k));\end{aligned} \tag{6}$$

$$\begin{aligned}\rho_{j2}(k) =&\rho_{j2}(k-1) + \eta_{\rho_2}(k)\cdot \\ &\cdot (y_j(k) - \psi_j(u_j(k)))\cdot \\ &\cdot (\rho_{j1}(k-1)u_j(k) + \rho_{j2}(k-1))\cdot \\ &\cdot \sigma(\rho_{j3}(k-1)u_j(k))\cdot \\ &\cdot \sigma(-\rho_{j3}(k-1)u_j(k))\cdot u_j(k)v\end{aligned} \tag{7}$$

$$\begin{aligned}\rho_{j3}(k) =&\rho_{j3}(k-1) + \eta_{\rho_3}(k)\cdot (y_j(k) - \psi_j(u_j(k)))\cdot \\ &\cdot \sigma(\rho_{j3}(k-1)u_j(k));\end{aligned} \tag{8}$$

$$\rho_{j4}(k) =\rho_{j4}(k-1) + \eta_{\rho_4}(k)\cdot (y_j(k) - \psi_j(u_j(k))), \tag{9}$$

where $\psi_j(u_j(k)) = LEAF_j(u_j(k), \rho_{j1}(k-1), \rho_{j2}(k-1), \rho_{j3}(k-1), \rho_{j4}(k-1))$, and η_{ρ_1} η_{ρ_2}, η_{ρ_3}, η_{ρ_4} - learning rates for ρ_1, ρ_2, ρ_3, ρ_4 correspondingly.

The second stage of DSPT uses the corrected loss value:

$$\begin{aligned}\tilde{E}_j(k) =&y_j(k) - \psi_j(u_j(k), \rho_{j1}(k), \rho_{j2}(k), \rho_{j3}(k), \rho_{j4}(k)) \\ =&y_j(k) - ((\rho_{j1}(k)u_j(k) + \rho_{j2}(k))\cdot \\ &\cdot \sigma(\rho_{j3}(k)u_j(k)) + \rho_{j4}(k)) \\ =&y_j(k) - (\rho_{j1}(k)w_j^T(k-1)x(k) + \rho_{j2}(k))\cdot \\ &\cdot \sigma(\rho_{j3}(k)w_j^T(k-1)x(k)) - \rho_{j4}(k).\end{aligned} \tag{10}$$

The delta rule for synaptic weights depends on the partial derivative by $\tilde{u}_j(k)$:

$$\begin{aligned}\frac{d\psi_j(\tilde{u}_j)}{d\tilde{u}_j} =&\frac{d((\rho_{j1}\tilde{u}_j + \rho_{j2})\sigma(\rho_{j3}\tilde{u}_j) + \rho_{j4})}{d\tilde{u}_j} = \\ =&\rho_{j1}\sigma(\rho_{j3}\tilde{u}_j) + (\rho_{j1}\tilde{u}_j + \rho_{j2})\cdot \\ &\cdot \sigma(\rho_{j3}\tilde{u}_j)\sigma(-\rho_{j3}\tilde{u}_j)\rho_{j3},\end{aligned} \tag{11}$$

where j - index of the neuron, $\tilde{u}_j$ - the corrected signal of internal activation after updating the LEAF parameters, and time indexes k, $(k-1)$ are omitted for brevity.

The second stage of DSPT updates the synaptic weights based on the corrected loss value and the updated LEAF parameters:

$$\begin{aligned}w_{ji}(k) = w_{ji}(k-1) + \eta_w(k)\tilde{e}_j(k)\frac{d\psi_j(\tilde{u}_j(k)}{d\tilde{u}_j(k))}x_i(k) = \\ =w_{ji}(k-1) + \eta_w(k)\tilde{e}_j(k)\cdot \\ \cdot \rho_{j1}(k)\sigma(\rho_{j3}(k)\tilde{u}_j(k)) + (\rho_{j1}(k)\tilde{u}_j(k) + \rho_{j2}(k))\cdot \\ \cdot \sigma(\rho_{j3}(k)\tilde{u}_j(k))\sigma(-\rho_{j3}(k)\tilde{u}_j(k))\rho_{j3}(k)\cdot x_i(k),\end{aligned} \tag{12}$$

where j - index of the neuron, $\tilde{u}_j$ - the corrected signal of internal activation after updating the LEAF parameters, $w_{ji}(k-1)$ - synaptic weight for the i-th input on the previous time step, $\rho_{j1}(k), ..., \rho_{j4}(k)$ - the updated LEAF parameter values after the first stage of DSPT, $\eta_w(k)$ - learning rate for synaptic weights w on time step k.

In summary, the neural network training procedure for the LEAF-based networks is close to the standard gradient procedure with backward error propagation. The only difference is that the procedure for LEAF computes the errors twice: once before clarifying the LEAF parameters and one more time after the clarification.

In a multi-layer network, it is sensible to update the LEAF parameters across all layers first, update the non-activation parameters across all layers second and repeat this procedure for each new mini-batch in the training set [20].

IV. EXPERIMENTAL EVALUATION

We evaluate the performance of LEAF on the image classification task using two data sets: Fashion-MNIST [21] and CIFAR-10 [22]. We use two neural network architectures for evaluation: LeNet-5 [23] and KerasNet [24]. In order to keep the results reproducible and comparable, we use 42 as the fixed seed value and the same starting weights for all network variants. During training, we record the average training set loss across mini-batches and the resulting test set error at the end of each epoch. The implementation is available on GitHub: s-kostyuk/leaf-aaf. The following subsections provide details on the experiment process.

A. AUGMENTATION AND BATCHING

We employ data augmentation for both the Fashion-MNIST and CIFAR-10 datasets to extend the dataset and improve the robustness of the trained models. The experiment implementation executes a random horizontal flip with a probability of 50% and a random two-dimensional shift by at most 0.1 times the image size. The implementation applies augmentation on a mini-batch basis, so the same image gets different random transformations on different epochs.

We use the 5:1 split between the training and the test sets, the standard split for Fashion-MNIST and CIFAR-10. The mini-batch size is 128 images per mini-batch for optimal GPU usage, reducing the data transfer time between the system RAM and the GPU RAM (VRAM), hence shortening the epoch duration.

B. NEURAL NETWORK ARCHITECTURE

LeNet-5 and KerasNet are multi-layer convolutional neural networks (CNNs) with different architectures. LeNet-5 is the simpler of the two, containing two convolutional layers with max pooling, one fully connected layer with an activation function, and one fully connected layer on the output with Softmax. Fig. 2 illustrates the architecture of LeNet-5.

KerasNet has a VGG-like architecture with two convolutional layers in series. KerasNet has, in total, four convolutional layers, two max-pooling layers, two two-dimensional

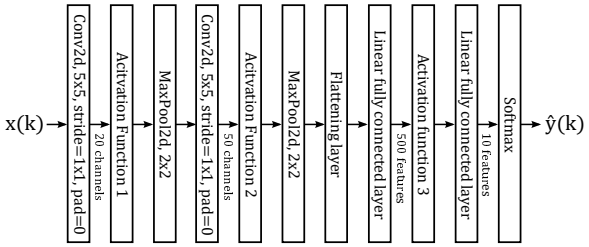

Figure 2. The architecture of LeNet-5

stochastic dropout [25] layers, two fully-connected linear layers, and one stochastic dropout layer between the fully-connected layers. While activation functions in the internal layers differ between the experiments, the output activation function for KerasNet is always Softmax.

Fig. 3 illustrates the architecture of KerasNet. In this illustration, we combined the activation function blocks with the corresponding convolutional and fully-connected blocks to reduce the figure size.

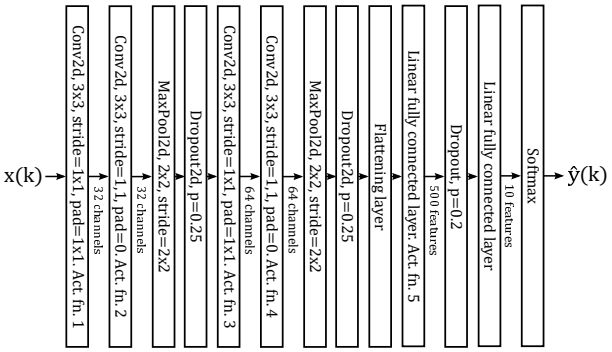

Figure 3. The architecture of KerasNet

C. NETWORK TRAINING PROCEDURE

We train all network variants for 100 epochs from scratch, minimizing the cross-entropy loss on the training data set.

Unless explicitly stated, all experiments use the ADAM optimizer with the base learning rate of 1×10^{-3}. For networks with trainable activation functions (AHAF and LEAF), we evaluate both the classic and the double-stage parameter tuning (DSPT) processes. The DSPT implementation employs separate optimizer instances for the activation function parameters and the non-activation synaptic weights.

For networks with LEAF, we reduce the learning rate for parameters ρ_2 and ρ_4 to 1×10^{-6} to stabilize the training process with ADAM. Here and below, we denote this modified procedure with reduced learning rates for ρ_2 and ρ_4 as "P24SI".

In order to validate the training process for LEAF-as-ReLU, we run a separate set of experiments using the RMSprop optimizer instead of ADAM. In this set of experiments, we set the learning rate to 1×10^{-4} for all network parameters, including LEAF parameters ρ_2 and ρ_4.

D. ACTIVATION FUNCTION VARIANTS

For all networks and data sets, we evaluate four different sets of activation functions:

- ReLU-like activations;
- SiLU-like activations;
- Sigmoid-like activations;
- Tanh-like activations.

For ReLU-like and SiLU-like activations, we compare the performance between the base network with non-adaptive activations, the AHAF-based network, and the LEAF-based network. Following Table 1, we set the initial parameters of AHAF and LEAF to closely follow the base functions, ReLU or SiLU, depending on the comparison set. As sufficiently close approximations of ReLU, we use AHAF with $\gamma = 2^{16}$ and LEAF with $\rho_3 = 2^{16}$ instead of setting the corresponding parameters to $+\infty$.

We use a similar approach to compare the networks with Sigmoid-like and Tanh-like activations. As AHAF does not replace Sigmoid and Tanh, we only compare the base and the LEAF-based network variants.

The networks with adaptive activations (AHAF and LEAF) use adaptive activations across all hidden layers, both the convolutional and the fully connected. The only exception from the rule is the output network layer that always employs the non-adaptive Softmax activation.

During the evaluation, we discovered differences in the output values between the built-in SiLU implementation in PyTorch and the same function implemented from scratch. The delta squared error between two implementations reaches about 9×10^{-13} for an NVIDIA GPU with float32. Fig. 4 illustrates the error.

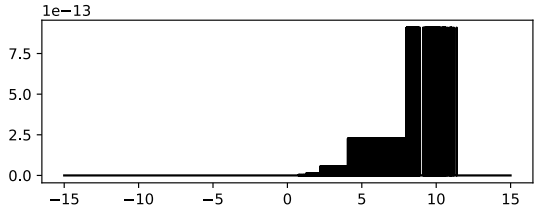

Figure 4. The delta squared error between two SiLU implementations

In order to keep the performance of all SiLU-like functions on the same level, the experiment uses a custom implementation of the following form as the base function:

```
import torch
def silu_manual(x):
    return x * torch.sigmoid(x)
```

E. EXECUTION ENVIRONMENT

The experiment implementation uses PyTorch 2.0 [26] as the deep learning framework and Python 3.8 as the programming language. We execute all experiments on a desktop computer with the NVIDIA RTX A4000 GPU. The floating

point precision is 32 bits (float32) for all experiments, the default floating point precision value for GPU in PyTorch.

V. RESULTS AND DISCUSSION

A. NETWORKS WITH LINEAR UNITS - TRAINING RESULTS

Networks with linear units (ReLU-like and SiLU-like activations) show high performance on the image classification task. As expected, the additional complexity of the KerasNet model provides a measurable advantage over LeNet-5 on the Fashion-MNIST dataset and up to 7% better classification accuracy on CIFAR-10. Tables 2 and 3 illustrate the best test set accuracy across all training epochs.

Table 2. Training results for LeNet-5, up to 100 epochs

Activ.	Init.	Procedure	F-MNIST		CIFAR-10	
			Acc.,%	Ep.	Acc.,%	Ep.
ReLU	N/A	Classic	92.39%	89	79.09%	93
AHAF	ReLU	Classic	92.56%	57	78.90%	90
AHAF	ReLU	DSPT	92.65%	59	78.56%	91
LEAF	ReLU	P24SI	92.75%	89	79.35%	75
LEAF	ReLU	DSPT, P24SI	92.56%	100	79.38%	86
SiLU	N/A	Classic	92.29%	42	78.08%	84
AHAF	SiLU	Classic	92.61%	37	78.53%	92
AHAF	SiLU	DSPT	92.56%	76	78.55%	87
LEAF	SiLU	P24SI	92.57%	91	78.41%	97
LEAF	SiLU	DSPT, P24SI	92.56%	98	78.89%	87

Table 3. Training results for KerasNet, up to 100 epochs

Activ.	Init.	Procedure	F-MNIST		CIFAR-10	
			Acc.,%	Ep.	Acc.,%	Ep.
ReLU	N/A	Classic	93.97%	93	83.74%	91
AHAF	ReLU	Classic	93.85%	73	84.30%	94
AHAF	ReLU	DSPT	94.00%	96	84.32%	99
LEAF	ReLU	P24SI	94.02%	100	84.21%	75
LEAF	ReLU	DSPT, P24SI	94.20%	96	84.26%	98
SiLU	N/A	Classic	93.75%	89	85.50%	97
AHAF	SiLU	Classic	94.25%	89	86.23%	96
AHAF	SiLU	DSPT	94.24%	85	86.55%	99
LEAF	SiLU	P24SI	94.20%	91	86.41%	93
LEAF	SiLU	DSPT, P24SI	94.27%	93	86.52%	99

While better than the base model, the performance of AHAF-based and LEAF-based networks remains on the same level. The corresponding LEAF and AHAF variants remain close across all epochs.

Like AHAF, LEAF keeps its general form while mostly manipulating the angle of its linear portion (the ρ_1 parameter). The ρ_3 parameter in LEAF-as-ReLU remains mostly unchanged due to its gradient properties with high initial values. Fig. 5 displays the LEAF-as-SiLU function form after 100 training epochs with DSPT and P24SI.

Depending on the starting values of non-activation synaptic weights, either AHAF or SiLU gets an advantage in performance. We ran additional experiments on CIFAR-10 with different seed values to validate this observation. Out of five runs, AHAF-as-SiLU shows the best performance with the 42, 128, and 7823 seed values, while LEAF-as-SiLU shows the best with seeds 100 and 1999. In a LeNet-5 network, AHAF-as-SiLU and AHAF-as-ReLU show the best with seeds 1999 and 7823 correspondingly, while LEAF-as-ReLU shows the best for the rest of the seeds.

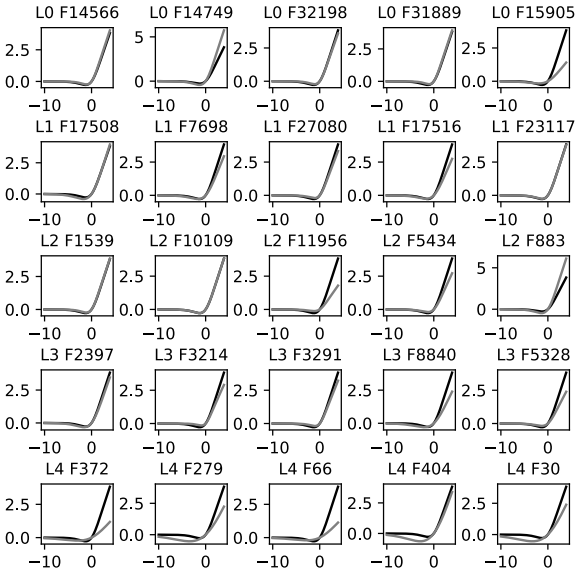

Figure 5. Activation function form of randomly sampled LEAF-as-SiLU instances in KerasNet on CIFAR-10. "L" denotes the layer, "F" denodes an instance in this layer

B. NETWORKS WITH SQUASHING UNITS - TRAINING RESULTS

One of the LEAF strengths is its ability to replace squashing functions in deep neural networks. Applying LEAF-as-Tanh and LEAF-as-Sigmoid in convolutional networks leads to some interesting results. Tables 4 and 5 show the training results for LeNet-5 and KerasNet.

Table 4. Training results for LeNet-5, up to 100 epochs

Activ.	Init.	Procedure	F-MNIST		CIFAR-10	
			Acc.,%	Ep.	Acc.,%	Ep.
Tanh	N/A	Classic	91.70%	99	76.48%	93
LEAF	Tanh	P24SI	92.45%	97	78.24%	73
LEAF	Tanh	DSPT, P24SI	92.55%	97	78.44%	75
Sigm.	N/A	Classic	91.21%	99	71.76%	97
LEAF	Sigm.	P24SI	91.83%	93	71.51%	98
LEAF	Sigm.	DSPT, P24SI	91.60%	98	71.95%	100

Table 5. Training results for KerasNet, up to 100 epochs

Activ.	Init.	Procedure	F-MNIST		CIFAR-10	
			Acc.,%	Ep.	Acc.,%	Ep.
Tanh	N/A	Classic	91.79%	96	77.77%	99
LEAF	Tanh	P24SI	93.12%	79	84.46%	99
LEAF	Tanh	DSPT, P24SI	93.35%	94	84.46%	100
Sigm.	N/A	Classic	92.05%	94	75.99%	91
LEAF	Sigm.	P24SI	92.97%	91	77.21%	99
LEAF	Sigm.	DSPT, P24SI	92.61%	100	77.61%	99

Utilizing its self-adaptation ability, LEAF-as-Tanh performs significantly better than its base counterpart. The performance of LEAF-as-Tanh approaches the performance

of LEAF-as-ReLU and LEAF-as-SiLU on the same datasets, showing the potential of adaptive activation functions even in "classic" environments dominated by linear units. The performance of LEAF-as-Sigmoid is significantly lower due to the "vanishing gradient" effect on negative inputs.

Based on the training process recordings, LEAF-as-Tanh continues updating its parameters and reaches higher performance while the base Tanh function leads the network to a performance plateau. Fig. 6 illustrates the loss and accuracy across epochs for LEAF-as-Tanh.

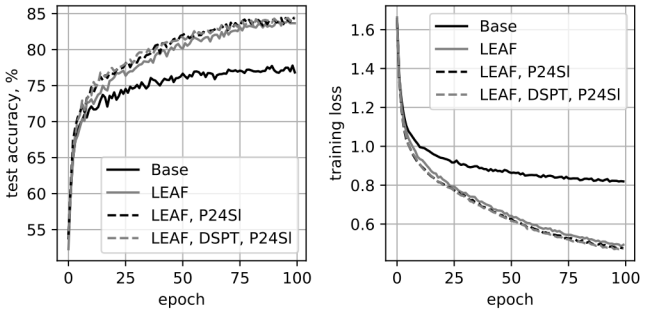

Figure 6. The training set loss and the test set accuracy across epochs for KerasNet with Tanh-like activations on CIFAR-10

Analyzing the activation function form, LEAF-as-Tanh and LEAF-as-Sigmoid attempt to "rectify" themselves in a convolutional network, hinting at the advantages of linear functions in CNNs. Fig. 7 shows the form of LEAF-as-Tanh in KerasNet after training on CIFAR-10 with DSPT and P24SI. Fig. 8 illustrates LEAF-as-Sigmoid.

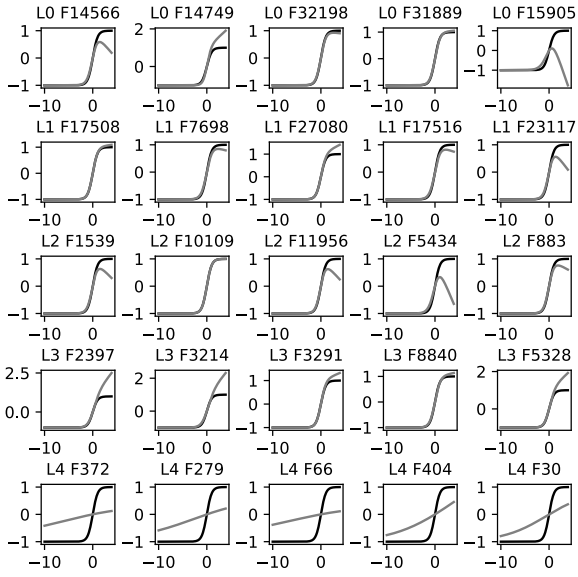

Figure 7. Randomly sampled LEAF-as-Tanh instances

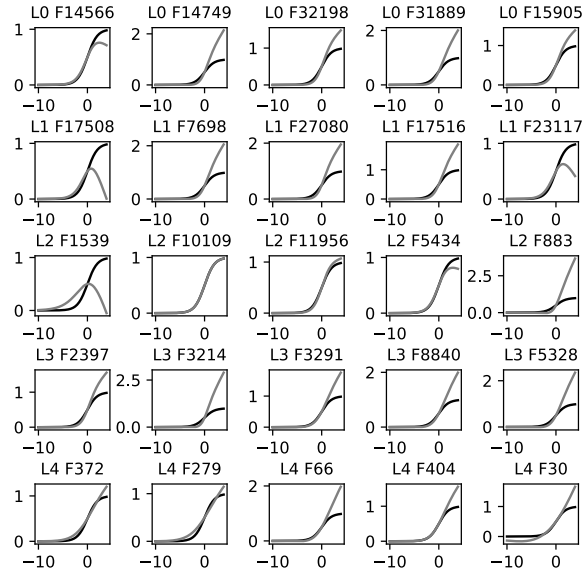

Figure 8. Randomly sampled LEAF-as-Sigmoid instances

C. LEAF-AS-RELU TRAINING STABILITY

While working on the experiment, we discovered the positive impact of the reduced learning rates on LEAF stability. With the regular learning rates, LEAF-as-ReLU shows serious convergence issues with ADAM and shows inferior performance with RMSprop. As mentioned in Section IV, we use reduced learning rates for LEAF parameters ρ_2 and ρ_4 (the "P24SI" procedure) to improve the training.

Fig. 9 illustrates the convergence for different network training procedures, including the "classic" procedure with the same learning rate for all parameters and the "P24SI" procedure with reduced learning rates for ρ_2 and ρ_4.

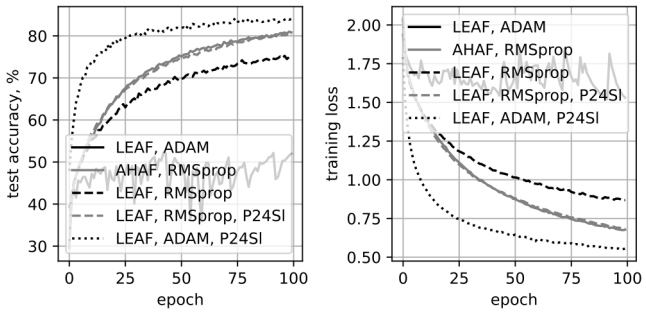

Figure 9. Comparison of training procedures for LEAF in KerasNet on CIFAR-10

VI. CONCLUSIONS

This paper introduces a learnable extended activation (LEAF) that can change its form during training and assume the shape of existing activation functions.

While combining the strengths of rectifier units and squashing functions, LEAF does not suffer from the "vanishing gradient" effect. It successfully operates in feed-forward

and recurrent networks while adapting to the current data processing task. The experiments prove the function's ability to adapt its form to reach lower loss values, particularly for squashing functions in convolutional neural networks.

The proposed activation function and the corresponding training algorithm significantly improve the data processing quality in the image classification task over the base non-adaptive implementations. The proposed solution is relatively simple from the computational standpoint and suitable for deep neural network architectures.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[2] G. Cybenko, "Approximation by superpositions of a sigmoidal function," Mathematics of Control, Signals and Systems, vol. 2, no. 4, pp. 303–314, Dec 1989. [Online]. Available: https://doi.org/10.1007/BF02551274

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[4] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Association for Computational Linguistics, 2014. [Online]. Available: https://doi.org/10.3115/v1/d14-1179

[5] E. Parisotto, H. F. Song, J. W. Rae, R. Pascanu, C. Gulcehre, S. M. Jayakumar, M. Jaderberg, R. L. Kaufman, A. Clark, S. Noury, M. M. Botvinick, N. Heess, and R. Hadsell, "Stabilizing transformers for reinforcement learning," in Proceedings of the 37th International Conference on Machine Learning, ser. ICML'20. JMLR.org, 2020. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/3524938.3525632

[6] N. Shazeer, "Glu variants improve transformer," 2020. [Online]. Available: https://arxiv.org/abs/2002.05202

[7] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," Neurocomputing, vol. 503, pp. 92–108, Sep. 2022. [Online]. Available: https://doi.org/10.1016/j.neucom.2022.06.111

[8] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," Neural Networks, vol. 107, pp. 3–11, Nov. 2018. [Online]. Available: https://doi.org/10.1016/j.neunet.2017.12.012

[9] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," in 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=Hkuq2EkPf

[10] X. Jin, C. Xu, J. Feng, Y. Wei, J. Xiong, and S. Yan, "Deep learning with s-shaped rectified linear activation units," ArXiv, vol. abs/1512.07030, 2015.

[11] M. Tanaka, "Weighted sigmoid gate unit for an activation function of deep neural network," Pattern Recognit. Lett., vol. 135, pp. 354–359, 2018.

[12] Y. Bodyanskiy and S. Kostiuk, "Adaptive hybrid activation function for deep neural networks," System research and information technologies, no. 1, pp. 87–96, Apr. 2022. [Online]. Available: https://doi.org/10.20535/srit.2308-8893.2022.1.07

[13] J. Kruschke and J. Movellan, "Benefits of gain: speeded learning and minimal hidden layers in back-propagation networks," IEEE Transactions on Systems, Man, and Cybernetics, vol. 21, no. 1, pp. 273–280, Jan 1991.

[14] Z. Hu and H. Shao, "The study of neural network adaptive control systems," Control and Decision, no. 7, pp. 361–366, 1992.

[15] C.-T. Chen and W.-D. Chang, "A feedforward neural network with function shape autotuning," Neural Netw., vol. 9, no. 4, p. 627–641, Jun. 1996. [Online]. Available: https://doi.org/10.1016/0893-6080(96)00006-8

[16] E. Trentin, "Networks with trainable amplitude of activation functions," Neural Netw., vol. 14, no. 4–5, p. 471–493, May 2001. [Online]. Available: https://doi.org/10.1016/S0893-6080(01)00028-4

[17] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, "Learning activation functions to improve deep neural networks," 2015.

[18] L. R. Sütfeld, F. Brieger, H. Finger, S. Füllhase, and G. Pipa, "Adaptive blending units: Trainable activation functions for deep neural networks," 2018.

[19] Y. V. Bodyanskiy, A. O. Deineko, I. Pliss, and V. Slepanska, "Formal neuron based on adaptive parametric rectified linear activation function and its learning," in International Workshop on Digital Content & Smart Multimedia, 2019.

[20] Y. Bodyanskiy and S. Kostiuk, "The two-step parameter tuning procedure for artificial neurons with adaptive activation functions," in Neural network technologies and their applications NNTA-2022: collection of scientific papers of the 21-st International scientific conference "Neural network technologies and their applications NNTA-2022", S. Kovalevsky, Ed. Kramatorsk: DSEA, 2022, pp. 30–36, in Ukrainian.

[21] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[22] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

[24] F. Chollet, "Train a simple deep cnn on the cifar10 small images dataset — keras 1.2.2." 2017, online. [Online]. Available: https://github.com/keras-team/keras/blob/1.2.2/examples/cifar10_cnn.py

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

YEVGENIY BODYANSKIY Professor at the Department of Artificial Intelligence, Scientific Head at the Control Systems Research Laboratory (CSRL), Kharkiv National University of Radio Electronics Member of the specialized scientific council, Member of STC, IEEE Senior Member, Doctor of Technical Sciences, Professor. Scientific interests: Hybrid systems of Computational Intelligence, Data Stream Mining, Big Data, Deep Learning, Evolving Systems.

SERHII KOSTIUK Ph.D. Student, Department of Artificial Intelligence, Kharkiv National University of Radio Electronics. Scientific interests: Arificial Neural Networks, Deep Learning, Transfer Learning, Evolving Systems, Adaptive Activation Functions.