

# **Comparative Analysis of Activation Functions**

DEPARTMENT OF INFORMATION TECHNOLOGY

## **Submitted By:**

Aryan Sinha (2021UIT3019)

Jatin Nagarwal (2021UIT3033)

Muddit Lal (2021UIT3060)

Under the supervision of:

**Dr. Apoorvi Sood**



DEPARTMENT OF INFORMATION TECHNOLOGY  
NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

MAY, 2025

## DECLARATION



Department of Information Technology  
Delhi-110078, India

We, Aryan Sinha (2021UIT3019), Jatin Nagarwal (2021UIT3033), and Muddit Lal (2021UIT3060) of B. Tech., Department of Information Technology, hereby declare that the Project II - report titled “Comparative Analysis of Activation Functions” which is submitted by us to the Department of Information Technology, Netaji Subhas University of Technology, is original and not copied from source without proper citation. This work has not previously formed the basis for the award of any Degree.

**Place:** Delhi

**Date:**13/05/2025

**Aryan Sinha**  
(2021UIT3019)

**Jatin Nagarwal**  
(2021UIT3033)

**Muddit Lal**  
(2021UIT3060)

## CERTIFICATE



Department of Information Technology  
Delhi-110078, India

This is to certify that the work embodied in the Project II-Report titled “Comparative Analysis of Activation Functions” has been completed by Aryan Sinha (2021UIT3019), Jatin Nagarwal (2021UIT3033), and Muddit Lal (2021UIT3060) of B.Tech., Department of Information Technology, under my guidance. This work has not been submitted for any other diploma or degree of any University.

**Place:** Delhi

**Date:** 13/05/2025

**Dr. Apoorvi Sood**

Department of Information Technology  
Netaji Subhas University of Technology

## ACKNOWLEDGEMENT

We would like to take this opportunity to acknowledge the support of all those without whom the completion of this project in fruition would not be possible.

First and foremost, we extend our sincere thanks to our supervisor, Dr. Apoorvi Sood, for her invaluable guidance, insights, and encouragement. Her expertise played a pivotal role in shaping the direction and outcomes of our research.

We would like to highlight the collective efforts and dedication of our project team, including Aryan, Jatin, and Muddit. The shared commitment, innovative ideas, and teamwork were the foundation of this project's success.

Additionally, we are grateful to the research community whose previous work and findings inspired and informed our approach. Their contributions provided us with a strong starting point for our research.

We would also like to extend our appreciation to the Artificial Intelligence Lab at NSUT for providing access to advanced GPU infrastructure, which played a crucial role in expediting the training and evaluation of complex deep learning models involved in this research. Finally, we extend our deepest appreciation to our families and friends for their constant support, encouragement, and patience throughout this journey. Their belief in us has been a tremendous source of motivation.

This project stands as a testament to the power of collaboration and the support of a vibrant academic and personal network. We are sincerely thankful to everyone who contributed to making it a success.

# TABLE OF CONTENTS

<b>DECLARATION</b>	<b>i</b>
<b>CERTIFICATE</b>	<b>ii</b>
<b>ACKNOWLEDGEMENT</b>	<b>iii</b>
<b>TABLE OF CONTENTS</b>	<b>iv</b>
<b>LIST OF FIGURES</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Abstract	1
1.2 Introduction	3
1.3 Research Motivation	5
1.4 Theoretical Overview	7
1.4.1 Deep Learning	7
1.4.2 Convolutional Neural Networks (CNNs)	8
1.4.3 Activation Functions and Non-linearity	10
1.4.4 Sparsity in Neural Networks	11
1.5 Technical Overview	12
1.5.1 Libraries Used	12
1.5.2 Hardware Stack	13
1.6 Literature Review	14
1.6.1 Evolution of Activation Functions in Deep Learning	14
1.6.2 Persistent Challenges in Activation Function Design	15

1.6.3	Emergence of HcLSH Activation Function . . . . .	16
<b>2</b>	<b>Proposed Methodology . . . . .</b>	<b>17</b>
2.1	Selection of Activation Functions for Comparative Study . . . . .	18
2.2	Dataset Acquisition and Preprocessing . . . . .	23
2.2.1	CIFAR-10 . . . . .	24
2.2.2	CIFAR-100 . . . . .	24
2.2.3	Fashion-MNIST . . . . .	25
2.3	Network Architecture and Training Configuration . . . . .	27
2.3.1	Convolutional Neural Network (CNN) Architecture . . . . .	27
2.4	Training Configurations . . . . .	28
2.5	Experimental Automation and Result Logging Framework . . . . .	30
<b>3</b>	<b>Implementation and Work Done . . . . .</b>	<b>32</b>
3.1	Setting Up the Environment and Configuring . . . . .	32
3.2	Importing Libraries and Setting Up Dependencies . . . . .	33
3.3	Using Activation Functions . . . . .	34
3.4	Preparing and Processing the Dataset . . . . .	36
3.5	Structuring the Model's Architecture . . . . .	36
3.6	Implementing a Recurrent Neural Network Variant . . . . .	38
3.7	Compiling, Training, and Validating the Model . . . . .	38
3.8	Storing and Displaying the Model . . . . .	38
3.9	Plotting the Graphs . . . . .	39
3.10	Documentation of Evaluation and Results . . . . .	39
<b>4</b>	<b>Results . . . . .</b>	<b>40</b>
4.1	Performance on CIFAR-10 Dataset: . . . . .	40
4.1.1	GCU Function . . . . .	41
4.1.2	eLiSH Function . . . . .	42
4.1.3	hcLSH Function . . . . .	42
4.1.4	mish Function . . . . .	43

4.1.5	Penalized Tanh Function . . . . .	43
4.1.6	Relu Function . . . . .	44
4.1.7	rsigelu Function . . . . .	44
4.1.8	Sigmoid Function . . . . .	45
4.1.9	Tanh Exp Function . . . . .	45
4.1.10	Tanh Function . . . . .	46
4.2	Performance on CIFAR-100 Dataset: . . . . .	47
4.2.1	GCU Function . . . . .	47
4.2.2	eLiSH Function . . . . .	48
4.2.3	hcLSH Function . . . . .	48
4.2.4	mish Function . . . . .	49
4.2.5	Penalized Tanh Function . . . . .	49
4.2.6	Relu Function . . . . .	50
4.2.7	rsigelu Function . . . . .	50
4.2.8	Sigmoid Function . . . . .	51
4.2.9	Tanh Exp Function . . . . .	51
4.2.10	Tanh Function . . . . .	52
4.3	Performance on Fashion-MNIST Dataset: . . . . .	53
4.3.1	GCU Function . . . . .	53
4.3.2	eLiSH Function . . . . .	54
4.3.3	hcLSH Function . . . . .	54
4.3.4	mish Function . . . . .	55
4.3.5	Penalized Tanh Function . . . . .	55
4.3.6	Relu Function . . . . .	56
4.3.7	rsigelu Function . . . . .	56
4.3.8	Sigmoid Function . . . . .	57
4.3.9	Tanh Exp Function . . . . .	57
4.3.10	Tanh Function . . . . .	58
4.4	Conclusion . . . . .	59

<b>5</b>	<b>Future Work</b>	<b>61</b>
<b>6</b>	<b>References</b>	<b>63</b>
<b>7</b>	<b>Plagarism Report</b>	<b>67</b>
<b>8</b>	<b>Appendix</b>	<b>68</b>



# List of Figures

1.1	A basic Neural Network . . . . .	7
1.2	CNN Architecture Diagram . . . . .	9
1.3	Tensorflow . . . . .	12
1.4	Python Pandas . . . . .	13
2.1	Mathematical Representation of the Activation Fuctions (part 1) . . . . .	20
2.2	Mathematical Representation of the Activation Fuctions (part 2) . . . . .	21
2.3	A Subset of CIFAR-10 Dataset . . . . .	24
2.4	A Subset of CIFAR-100 Dataset . . . . .	25
2.5	A Subset of Fashion-MNIST Dataset . . . . .	26
3.1	Configuring the Environment and connecting to the GPU Access . . . . .	32
3.2	importing the libraries . . . . .	33
3.3	Custom Definition of Activation Functions . . . . .	35
3.4	Model Architecture . . . . .	37
4.1	GCU Function on CIFAR-10 Dataset . . . . .	41
4.2	eLiSH Function on CIFAR-10 Dataset . . . . .	42
4.3	hcLSH Function on CIFAR-10 Dataset . . . . .	42
4.4	mish Function on CIFAR-10 Dataset . . . . .	43
4.5	Penalized Tanh Function on CIFAR-10 Dataset . . . . .	43
4.6	Relu Function on CIFAR-10 Dataset . . . . .	44
4.7	rsigelu Function on CIFAR-10 Dataset . . . . .	44
4.8	Sigmoid Function on CIFAR-10 Dataset . . . . .	45
4.9	Tanh Exp Function on CIFAR-10 Dataset . . . . .	45

4.10	Tanh Function on CIFAR-10 Dataset . . . . .	46
4.11	GCU Function on CIFAR-100 Dataset . . . . .	47
4.12	eLiSH Function on CIFAR-100Dataset . . . . .	48
4.13	hcLSH Function on CIFAR-100 Dataset . . . . .	48
4.14	mish Function on CIFAR-100 Dataset . . . . .	49
4.15	Penalized Tanh Function on CIFAR-100 Dataset . . . . .	49
4.16	Relu Function on CIFAR-100 Dataset . . . . .	50
4.17	rsigelu Function on CIFAR-100 Dataset . . . . .	50
4.18	Sigmoid Function on CIFAR-100 Dataset . . . . .	51
4.19	Tanh Exp Function on CIFAR-100 Dataset . . . . .	51
4.20	Tanh Function on CIFAR-100 Dataset . . . . .	52
4.21	GCU Function on Fashion-MNIST Dataset . . . . .	53
4.22	eLiSH Function on Fashion-MNIST Dataset . . . . .	54
4.23	hcLSH Function on Fashion-MNIST Dataset . . . . .	54
4.24	mish Function on Fashion-MNIST Dataset . . . . .	55
4.25	Penalized Tanh Function on Fashion-MNIST Dataset . . . . .	55
4.26	Relu Function on Fashion-MNIST Dataset . . . . .	56
4.27	rsigelu Function on Fashion-MNIST Dataset . . . . .	56
4.28	Sigmoid Function on Fashion-MNIST Dataset . . . . .	57
4.29	Tanh Exp Function on Fashion-MNIST Dataset . . . . .	57
4.30	Tanh Function on Fashion-MNIST Dataset . . . . .	58

# Chapter 1

## Introduction

### 1.1 Abstract

In order to influence the expressive capability and learning dynamics of deep neural networks, activation functions are crucial for introducing non-linearity. Although the Rectified Linear Unit (ReLU) is frequently employed due to its ease of use and ability to solve the vanishing gradient problem, it has drawbacks such as bias shifts and dying neurons. By addressing these problems by maintaining contributions from negative inputs, enhancing gradient flow, and providing natural regularization, this work suggests a novel activation function called the Hyperbolic cosine Linearized Squashing function (HcLSH). The function additionally incorporates sparsity based on partial gradients. The Fashion MNIST, CIFAR-10, CIFAR-100 image classification datasets are used in this study to benchmark HcLSH against eleven activation functions, such as ReLU, Mish, and custom functions like Penalized Tanh and El-iSH. Models were trained, outcomes were logged, and training graphs were produced using automated workflows. With gains ranging from 0.2% to 96.4%, HcLSH beat both ReLU and Mish in 28 trials in terms of Top-1 and Top-3 accuracy. The enhanced performance and robustness of HcLSH were further validated by statistical and ablation tests.

Additionally, this study involved the implementation of custom CNN and RNN architectures tailored to support a fair and consistent comparison of activation functions across varied learning tasks. Automated pipelines were developed not only for model training but also for saving checkpoints, generating performance plots, and logging evaluation metrics,

ensuring reproducibility and transparency in the experimentation process. These findings highlight the importance of exploring activation function design as a means to improve deep learning model reliability, generalization, and efficiency across diverse image classification tasks.

This study offers a fresh perspective on activation function design and demonstrates how thoughtful architectural tweaks can significantly uplift deep network performance without introducing additional computational overhead.

## 1.2 Introduction

Recent years have seen the rise of deep learning as a crucial element of artificial intelligence, with remarkable advancements in a variety of fields, including natural language processing, audio recognition, and image categorization. The efficacy and performance of deep neural networks (DNNs), which are at the center of these advancements, are greatly influenced by the activation functions chosen for their topologies. The activation functions that provide neural networks the required non-linearity allow them to express intricate, non-trivial relationships between inputs and outputs. Historically, there have been several important stages in the evolution of activation functions. Early neural networks relied heavily on saturated functions such as the Hyperbolic Tangent (Tanh) and Sigmoid. Despite being essential to the early success of neural networks, these characteristics were eventually found to have inherent flaws, most notably the increasing and vanishing gradient problems. Because gradients conveyed during backpropagation may spontaneously drop or rise, influencing convergence and optimization, these issues hindered deep network training. A major turning point was the introduction of the Rectified Linear Unit (ReLU), which provided computational simplicity and partially alleviated the vanishing gradient problem by permitting gradients to flow freely for positive input values. The "dying neuron" phenomenon, in which certain neurons in the network go dormant and stop learning if they continuously generate zero outputs, is the most notable of the difficulties that ReLU brought about. Due to this restriction, a number of ReLU variations were created, such as Leaky ReLU, Parametric ReLU (PReLU), and Exponential Linear Units (ELUs), each of which was intended to rectify a particular deficiency of its predecessor. More complex activation functions, including Swish and Mish, have been developed in an effort to achieve even greater advancements. These functions improve the learning dynamics of deep networks by including smoothness and non-monotonicity. Notwithstanding these developments, difficulties still exist, especially in striking a balance between computing efficiency, gradient stability, sparsity, and non-linearity. To get over these persistent limitations, this work proposes a novel activation function known as the Hyperbolic cosine Linearized Squashing function (HcLSH). HcLSH is a composite, smooth, and monotonic

activation function designed to solve issues such as vanishing/exploding gradients and dying neurons while enhancing classification performance. It incorporates semi-saturated behavior, which is confined below and unbounded above, ensuring constant gradient flow, and partial gradient sparsity, which enables effective negative feature learning without overloading the network. The function maintains constant differentiability across its domain during training, which encourages high convergence. Extensive tests on various benchmark datasets and deep learning architectures have shown that HcLSH continuously performs better than traditional and modern activation functions in terms of convergence behavior, loss reduction, and classification accuracy. Furthermore, the effectiveness and dependability of HcLSH over current alternatives are supported by thorough statistical analyses, such as the Friedman ranking test and the Wilcoxon signed-rank test.

## 1.3 Research Motivation

Over the years, deep learning has evolved rapidly, with most progress driven by innovations in model architectures, optimizers, and activation functions. Among these, activation functions often get overlooked despite having a huge influence on how neural networks learn, propagate gradients, and extract meaningful patterns. While ReLU has long been the default due to its simplicity and ability to tackle the vanishing gradient issue, we kept noticing recurring problems in our own experiments such as dead neurons, bias shifts, and unstable training, especially in deeper networks and noisy datasets. This led us to question whether better alternatives existed. We explored popular options like Mish and Swish and even several experimental ones, but most either addressed a single issue or introduced new complexities like unstable gradients or heavy computations. None struck the right balance between simplicity, efficiency, and expressiveness.

That’s what motivated us to design a new activation function like the Hyperbolic cosine Linearized Squashing function (HcLSH). Our aim was to preserve contributions from negative activations, improve gradient flow, introduce a form of partial gradient-based sparsity, and naturally regularize the model, all while keeping computational costs minimal. The support from our institute’s AI lab and access to advanced GPUs made it feasible to conduct rigorous experiments across multiple datasets and models, benchmarking HcLSH against eleven other activation functions.

In the end, this project was driven by our curiosity, hands-on observations, and a shared goal to contribute something both practically useful and theoretically sound to the deep learning community.

The primary objective of this research is to develop, evaluate, and compare a novel activation function Hyperbolic Cosine Linearized Squashing Function (HcLSH) to address the limitations of frequently used activation functions like ReLU and Mish in DNN. To be very precise, we intend to:

- Propose and formulate the HcLSH function: Develop a mathematical model for HcLSH that overcomes challenges like the vanishing gradient problem, dying neurons, and bias shifts in negative input ranges.
- Implement the proposed activation function: Integrate HcLSH into both Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) and compare its performance against established activation functions (ReLU, Mish, Tanh, Sigmoid, etc.) across multiple datasets.
- Benchmark HcLSH's performance: Evaluate and benchmark HcLSH using four standard image classification datasets (Fashion MNIST, CIFAR-10, CIFAR-100, and SVHN) to measure its impact on accuracy, model stability, and training efficiency.
- Conduct ablation studies and statistical analyses: Use rigorous statistical methods to assess the robustness and adaptability of HcLSH in various neural network architectures and its capacity to generalize across different tasks.
- Contribute to the development of more robust deep learning models: By providing an enhanced activation function that promotes better gradient flow and natural regularization, this research aims to improve the overall performance, training stability, and generalization capability of deep neural networks.

Through this work, we strive to advance the understanding of activation function behavior and propose a practical solution for improving deep learning model training and performance across a wide range of applications.



## 1.4 Theoretical Overview

### 1.4.1 Deep Learning

”Deep learning” is a subset of machine learning techniques that models complex relationships found in large datasets using artificial neural networks. Deep learning basically enables models to learn hierarchical representations of data on their own by utilizing layers of interconnected neurons. This aids the software in identifying intricate patterns and traits within the data. The layers of nodes, or neurons, that make up these neural networks introduce non-linearity by carrying out an activation function after a weighted sum of their inputs. The main goal of learning is to alter these weights and biases. Back-propagation and optimization techniques like stochastic gradient descent (SGD) or more complex variations like Adam are used to accomplish this.

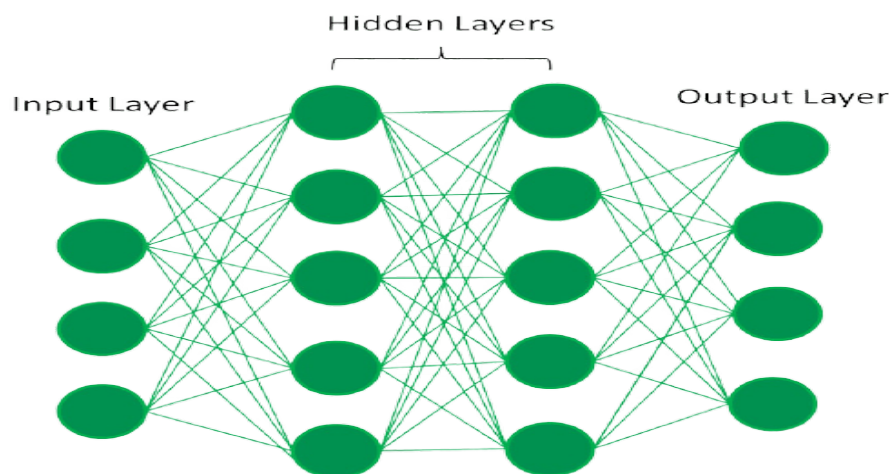


Figure 1.1: A basic Neural Network

The capacity of deep learning to scale with growing data and model complexity is one of its main benefits. Because deep networks have more hidden layers than shallow networks, they are able to learn more abstract features. Deeper layers may learn to identify faces, objects, or even intricate relationships between objects in a scene, while early levels may learn to detect edges and textures in visual data. Natural language processing (NLP), computer vision, and speech recognition have all advanced thanks to deep learning.

### **1.4.2 Convolutional Neural Networks (CNNs)**

Convolutional neural networks, or CNNs, are a special type of deep learning model that are mostly made to process image data, however they have also been applied to text and video in recent years. The basic idea underlying CNNs is that, unlike a simple feedforward network, which links every neuron to every other neuron, CNNs use something called convolutional layers. These layers apply several small filters (such 3x3 or 5x5 kernels) as we move deeper into the network. These filters sweep across the image and extract important patterns, such as edges, textures, shapes, and more complex features.

One of CNNs' best features is its ability to understand spatial hierarchies; lower layers can identify simple parts like lines and corners, middle layers can identify patterns like eyes or wheels, and deeper layers can understand whole objects like faces or cars. Because of this characteristic, CNNs can learn feature maps without the need for explicit definition, making them exceptionally adept at vision tasks. Additionally, due to weight sharing, which expedites training and lowers the risk of overfitting, convolutional layers have a substantially fewer number of parameters than fully linked networks. In order to reduce the computational load and make the model more resilient to slight shifts or distortions in the input image, pooling layers—typically max pooling or avg pooling—are inserted between convolution layers to compress the feature maps while preserving the essential information.

To observe how various configurations impact model performance across various datasets, we employed bespoke CNN architectures created in TensorFlow/Keras, altering the number of convolution layers, kernel sizes, pooling techniques, and activation functions. In comparison to well-known activations like ReLU, Mish, and others, we aimed to assess how well our novel activation function performed in certain CNN setups and whether it provided any appreciable gains in accuracy, convergence speed, or gradient stability.

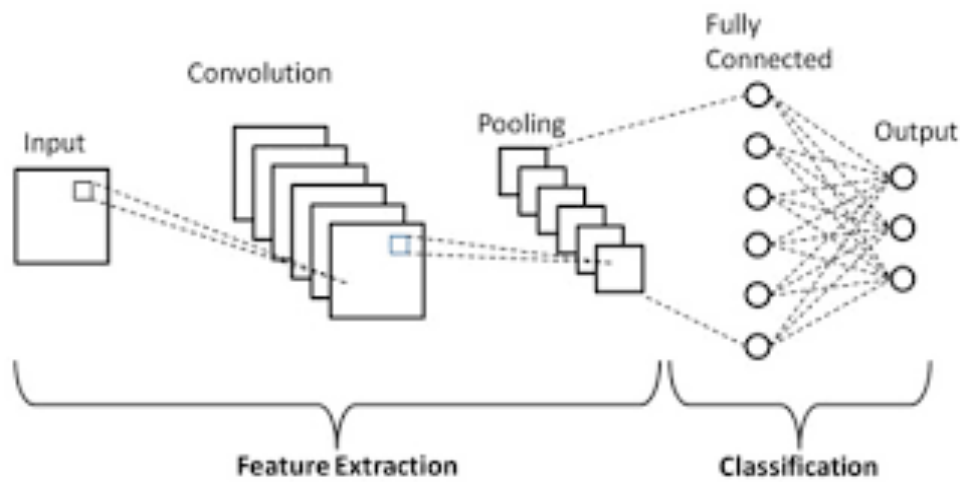


Figure 1.2: CNN Architecture Diagram

After the convolutional and pooling layers have finished gathering useful information from the image, the next step is to actually classify the input image into one of the preset categories. This is handled by the CNN's classifier component, which usually consists of a few dense, fully linked layers at the end of the network.

### 1.4.3 Activation Functions and Non-linearity

Activation functions are vitally essential in neural networks because they add non-linearity to the model, which enables it to grasp complicated correlations in the data. Consider them as filters that determine the relative importance of each input to the output. The neural network would simply be a linear model without activation functions, making it incapable of representing intricate data patterns. Numerous activation functions exist, each with unique advantages and disadvantages:

- ReLU (Rectified Linear Unit) is one of the most popular activation functions because it's simple and works well for most tasks. It works by passing positive values unchanged, but if the input is negative, it outputs zero. This helps solve the problem of vanishing gradients that often occurs with other functions, allowing deeper networks to learn better.
- Sigmoid is another common activation function, especially in earlier models. It maps input values to a range between 0 and 1, which makes it useful for binary classification tasks, such as predicting whether an image contains a cat or not. However, it can suffer from the vanishing gradient problem, where large inputs result in very small gradients, making learning slow and difficult.
- Tanh (Hyperbolic Tangent) is similar to Sigmoid but maps inputs to a range between -1 and 1, which helps with some issues in training. However, it still faces the vanishing gradient problem for very large or small inputs.

Despite their widespread use, these activation functions have drawbacks, such as sluggish training (in Sigmoid) or dying neurons (in ReLU). For this reason, in order to enhance learning and model performance, the research community keeps looking into better, more effective activation functions.

### 1.4.4 Sparsity in Neural Networks

In the context of neural networks, sparsity is the state in which a significant fraction of a vector, matrix, or tensor (for example, model parameters or neuron activations) is 0 or almost zero. To put it simply, sparse activations or weights in a neural network mean that only a small portion of the neurons are at any given time actively contributing to the output, with the remainder neurons either inactive or hardly making an impact.

In technical terms, there are two main ways that this sparsity can appear:

- **Sparse Activations:** For a given input sample, only a portion of neurons fire, or generate a non-zero output. By eliminating pointless computations, this not only lessens the computational load but also motivates the network to acquire more localized and discriminative feature representations.
- **Sparse Weights:** During training, the connection weights themselves can be adjusted so that a large number of them are pushed to precise zeros. Techniques like L1 regularization, sometimes referred to as Lasso, can do this by trimming unnecessary connections by applying a penalty proportionate to the absolute value of the weights.

Sparsity is naturally encouraged by certain activation functions. For example, by setting all negative inputs to zero, the conventional Rectified Linear Unit (ReLU) purposefully creates sparse activations. To provide controlled sparsity, more intricate functions like ELU, Leaky ReLU, or even your own HcLSH can be created or altered based on gradient properties or activation ranges. Neural network sparsity is a potent design principle that improves the efficacy, performance, and interpretability of high-dimensional, data-rich AI systems. It's not only a regularization technique.

## 1.5 Technical Overview

### 1.5.1 Libraries Used

- **TensorFlow:** A highly versatile, open-source numerical computing framework primarily designed for large-scale machine learning and deep learning applications. It provides low-level APIs for tensor operations and computational graph management, along with high-level modules for model building, distributed training, and automatic differentiation via its `tf.GradientTape`. TensorFlow provided the foundation for this work's definition of unique activation functions (such as our suggested HcLSH), model optimization via gradient descent variations, and effective GPU resource management for quicker computation.



Figure 1.3: Tensorflow

- **Keras (integrated within TensorFlow):** An interface for the TensorFlow library that functions as a high-level neural network API. By providing modular building parts such as Sequential and Functional APIs, as well as tools for model compilation, training, evaluation, and callbacks, Keras streamlines the process of developing models. Convolutional neural networks (CNNs) and recurrent neural networks (RNNs) were developed in this study using Keras extensively. Custom activation layers were also integrated, and automated training pipelines with early stopping and checkpointing features were implemented.

- **Matplotlib:** This 2D graphing tool may display training progress, accuracy trends, loss convergence, and confusion matrices. Its pyplot package was used to generate dynamic graphs for every experiment, enabling a visual comparison of activation functions across different datasets and model architectures.
- **Pandas:** A robust library for data analysis and manipulation that provides data structures such as DataFrames and Series. Its main functions were to input CSV data from several experimental runs, combine metrics for summary tables, and organize accuracy logs. Moreover, Pandas assisted in the management of statistical summaries, model performance logs, and aggregated results.



Figure 1.4: Python Pandas

### 1.5.2 Hardware Stack

The AI Laboratory at NSUT provided a 40 Gigabytes of VRAM GPU container, which was remotely accessed via a MacBook M1 Pro for model training and testing. By utilizing the massive computation and memory capacity of the specialized GPU environment, this configuration guaranteed faster deep learning processes with extensive picture datasets and unique activation functions.

## 1.6 Literature Review

Activation functions (AFs) are essential components in deep neural networks, shaping their ability to learn and optimize effectively. Their main purpose is to add non-linear transformations, allowing the network to capture intricate, real-world patterns that linear operations alone cannot represent. Over time, research on activation functions has progressed through key developmental phases, with each new advancement overcoming previous shortcomings and driving innovation in deep learning models. The ongoing refinement of activation functions remains an active area of research, reflecting their central importance in balancing computational efficiency, training stability, and model expressiveness in modern neural networks.

### 1.6.1 Evolution of Activation Functions in Deep Learning

Saturated functions like the Tanh and Sigmoid were crucial to early neural networks. These functions did, however, have significant shortcomings, most notably the vanishing gradient issue, which made deep network training difficult.

A significant advancement was made with the creation of the Rectified Linear Unit (ReLU), which provided quicker computing and somewhat mitigated vanishing gradients. By adding a little gradient for inactive units, further enhancements like Leaky ReLU and Parametric ReLU (PReLU) addressed ReLU's "dying neuron" problem.

By combining the advantages of previous methods with smooth, non-monotonic characteristics, recent developments such as Swish and Mish significantly improved activation functions. Performance on a variety of deep learning tasks has improved as a result of these developments.



### 1.6.2 Persistent Challenges in Activation Function Design

While modern activation functions represent significant improvements over early approaches, several critical limitations continue to affect neural network training and performance:

- **Gradient Instability in Deep Architectures:** Saturating nonlinearities (e.g., logistic sigmoid, hyperbolic tangent) exhibit exponential decay or growth in gradient magnitudes during error backpropagation. This phenomenon becomes particularly problematic in networks with numerous hidden layers.
- **Neuron Deactivation in Piecewise Linear Functions:** The rectified linear unit (ReLU) and its derivatives suffer from a characteristic failure mode where neurons enter permanent inactive states. When the weighted sum of inputs remains negative across multiple training iterations, these units either cease generating meaningful gradients, become unresponsive to input variations, or effectively "die" for the remaining training duration.
- **Output Distribution Biases:** Three main problems result from the imbalanced output patterns caused by common activation functions like ReLU: (1) layer-wise activation means deviating from zero, (2) potential internal covariate shifts during training, and (3) less effective weight update paths during optimization. These functions zero negatives while maintaining positives.
- **Non-Zero Centered Outputs:** The non-zero-centered nature of activation functions like ReLU introduces inherent asymmetries in network behavior, fundamentally altering training dynamics. By suppressing all negative inputs while preserving positive values unchanged, these functions create output distributions skewed toward positive activations.

### 1.6.3 Emergence of HcLSH Activation Function

The recent introduction of the Hyperbolic cosine Linearized Squashing (HcLSH) activation function represents a significant advancement in addressing persistent challenges in deep neural network training. Developed as a sophisticated composite function, HcLSH strategically combines multiple desirable mathematical properties to overcome limitations observed in conventional activation functions. Its carefully designed architecture maintains strict monotonicity while producing zero-centered outputs, ensuring balanced activations throughout the network and promoting more stable gradient flow during back-propagation. The function's semi-saturation characteristics, being bounded below while remaining unbounded above, provide a robust solution to the dual problems of vanishing and exploding gradients that commonly plague deep architectures.

A particularly innovative aspect of HcLSH is its implementation of partial gradient sparsity, which thoughtfully incorporates negative value processing without causing excessive suppression of activations, thereby effectively preventing the dying neuron phenomenon that affects ReLU-based functions. Furthermore, the function's continuous differentiability across its entire domain eliminates problematic points of non-smoothness that can disrupt optimization processes. This smoothness property contributes significantly to more stable and efficient training dynamics, allowing for reliable convergence even in deep network configurations. By simultaneously addressing multiple fundamental issues through its unified mathematical formulation, HcLSH emerges as a promising alternative to traditional activation functions, offering the potential to enhance both training stability and model performance across various neural network architectures and applications.

HcLSH reduces loss and increases classification accuracy across a range of deep learning architectures and datasets, according to experimental assessments. Performance gains over current activation functions have been confirmed by statistical tests.

# Chapter 2

## Proposed Methodology

This section presents the methodological framework for a rigorous empirical evaluation comparing the novel HcLSH activation function with contemporary alternatives. The study employs a systematic experimental design to assess multiple performance dimensions across diverse neural architectures and benchmark datasets. Our evaluation protocol establishes controlled testing conditions that enable direct comparison of learning efficiency, model accuracy, and training stability between activation functions. The experimental pipeline incorporates standardized metrics to quantify convergence behavior, gradient propagation characteristics, and computational efficiency throughout the training lifecycle. By maintaining consistent hyperparameters and initialization schemes across trials while varying only the activation functions, we isolate their specific contributions to model performance.

The methodology further includes ablation studies to examine how architectural choices and dataset characteristics interact with different activation functions. This comprehensive analytical approach provides insights into the relative strengths and operational boundaries of each activation function under investigation, with particular attention to HcLSH’s ability to address known limitations in gradient-based optimization. Through this multifaceted evaluation, we establish an evidence-based understanding of activation function performance across different learning scenarios and model complexities.

The methodology comprises several sequential stages, as described below:

## **2.1 Selection of Activation Functions for Comparative Study**

The comparative analysis incorporates a carefully curated selection of activation functions that represent the full spectrum of approaches in deep learning. The chosen functions were selected based on several key criteria: their established theoretical foundations, demonstrated practical utility, and prevalence in contemporary research literature. The selection encompasses both classical and modern activation functions, including saturated types like sigmoid and tanh that were foundational to early neural networks, along with more recent non-saturated variants such as ReLU and its numerous derivatives. This comprehensive selection also includes both monotonic functions that preserve order relationships and non-monotonic alternatives that have shown promise in specific applications. Furthermore, the comparison spans the full range of bounded functions that constrain output values and unbounded functions that allow unlimited activation magnitudes. By including this diverse array of activation functions, the analysis ensures robust evaluation across different mathematical formulations and behavioral characteristics, providing meaningful insights into their relative performance under standardized testing conditions while accounting for the varied architectural requirements and optimization challenges in modern deep learning systems.

The activation functions incorporated in this study include:

- Sigmoid
- Hyperbolic Tangent (Tanh)
- Rectified Linear Unit (ReLU)
- Leaky ReLU (LReLU)
- Parametric ReLU (PReLU)

- Exponential Linear Unit (ELU)
- Swish
- Mish
- GELU
- HcLSH (Hyperbolic cosine Linearized Squashing Function)

The selection of activation functions for comparison was guided by a systematic evaluation of their fundamental mathematical properties that critically influence neural network training dynamics. Particular attention was given to each function’s saturation characteristics, which determine how outputs behave at extreme inputs and significantly impact gradient flow during backpropagation. The analysis carefully considered gradient preservation properties that affect how effectively networks can learn across multiple layers, as well as output distribution characteristics like zero-centeredness that influence training stability.

Monotonicity properties were examined due to their role in maintaining consistent relationships between inputs and outputs throughout the learning process. Additionally, the selection accounted for each function’s differentiability and smoothness, which are crucial for optimization efficiency, along with their boundedness properties that affect numerical stability. By rigorously evaluating these core mathematical attributes, the chosen functions collectively represent the spectrum of design approaches in activation function development, enabling a comprehensive assessment of how different mathematical formulations affect learning performance across various neural architectures and problem domains.

This property-driven selection strategy ensures the comparative analysis captures the nuanced ways in which activation functions shape network behavior during training and inference.

Function name	The function equation	The derivative equation
<b>ReLU</b>	$f(x) = \max(0, x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$	$\hat{f}(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$
<b>L-ReLU</b>	$f(x) = \min(0, \alpha x) + \max(0, x)$ $= \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$	$\hat{f}(x) = \begin{cases} 1, & x \geq 0 \\ \alpha, & x < 0 \end{cases}$
<b>Penalized tanh</b>	$f(x) = \begin{cases} \tanh(x), & x > 0 \\ 0.25 \tanh(x), & x \leq 0 \end{cases}$	$\hat{f}(x) = \begin{cases} \text{sech}^2(x), & x > 0 \\ 0.25 \text{sech}^2(x), & x \leq 0 \end{cases}$
<b>Swish</b>	$f(x) = x \sigma(\beta x)$	$\hat{f}(x) = \beta f(x) + \sigma(\beta x)(1 - \beta f(x))$ $= \frac{\beta(1 + e^{-x} + x e^{-x})}{(1 + e^{-x})^2}$
<b>ELiSH</b>	$f(x) = \begin{cases} \frac{x}{1 + e^{-x}}, & x \geq 0 \\ \frac{e^x - 1}{1 + e^{-x}}, & x < 0 \end{cases}$	$\hat{f}(x) = \begin{cases} \frac{1}{1 + e^{-x}} + \frac{x e^{-x}}{(1 + e^{-x})^2}, & x \geq 0 \\ \frac{e^{-x}}{1 + e^{-x}} + \frac{e^{-x}(e^x - 1)}{(1 + e^{-x})^2}, & x < 0 \end{cases}$

Figure 2.1: Mathematical Representation of the Activation Functions (part 1)

The Rectified Linear Unit (ReLU) operates by simply passing positive values unchanged while completely suppressing negative inputs, creating a straightforward thresholding effect. Leaky ReLU modifies this approach by allowing a small, constant fraction of negative values to pass through, preventing complete neuron deactivation. Parametric Tanh builds upon the classic hyperbolic tangent function by incorporating learnable scaling parameters that enable the network to automatically adjust its saturation levels during training.

The Swish activation introduces an elegant self-gating mechanism, blending sigmoidal smoothing with linear identity mapping to create a smooth, non-monotonic response curve. This allows for more nuanced signal processing compared to simple rectified units. The Exponential Linear Sigmoid (ELiSH) takes this further by combining sigmoidal gating

with exponential treatment of negative inputs, creating a sophisticated function that handles positive and negative domains differently while maintaining overall smoothness.

Each of these activation strategies represents a different approach to balancing several key properties: maintaining sufficient linearity for effective learning, controlled saturation to prevent explosive growth, and appropriate differentiability for stable gradient-based optimization. They address various challenges in deep network training, from the dying neuron problem to gradient instability, through their distinct operational characteristics. The evolution from simple ReLU to more sophisticated functions like ELiSH demonstrates the ongoing refinement of activation functions to support increasingly complex neural architectures.

<b>Mish</b>	$f(x) = x \tanh(\ln(1 + e^x))$	$\begin{aligned} \hat{f}(x) &= \tanh \ln(1 + e^x) + \frac{x e^x \operatorname{sech}^2 \ln(e^x + 1)}{e^x + 1} \end{aligned}$
<b>Smish</b>	$f(x) = x \tanh(\ln(1 + \sigma(x)))$	$\hat{f}(x) = \frac{e^x(15e^{3x} + (8x + 28)e^{2x} + (12x + 18)e^x + 4x + 4)}{(5e^{2x} + 6e^x + 2)^2}$
<b>RSigELU</b>	$f(x) = \begin{cases} \frac{\alpha x}{1 + e^{-x}} + x, & x > 1 \\ x, & 0 \leq x \leq 1 \\ \alpha(e^x - 1), & x < 0 \end{cases}$ <p style="margin-left: 40px;"><math>\alpha \in (0,1)</math>, Control the slope of positive and negative regions.</p>	$\hat{f}(x) = \begin{cases} \frac{-\alpha x}{(e^x + 1)^2} + \frac{\alpha x - \alpha}{e^x + 1} + 1 + \alpha, & x > 1 \\ 1, & 0 \leq x \leq 1 \\ \alpha(e^x), & x < 0 \end{cases}$
<b>TanhExp</b>	$f(x) = x \tanh(e^x)$	$\hat{f}(x) = \tanh e^x - x e^x \tanh^2 e^x + x e^x$
<b>PFLU</b>	$f(x) = x \cdot \frac{1}{2} \cdot \left(1 + \frac{x}{\sqrt{1 + x^2}}\right)$	$\hat{f}(x) = \frac{1}{2} \cdot \left(1 + \frac{x}{\sqrt{1 + x^2}} + \frac{x}{(1 + x^2)\sqrt{1 + x^2}}\right)$

Figure 2.2: Mathematical Representation of the Activation Functions (part 2)

The Mish activation function combines the smooth nonlinearity of hyperbolic tangent with a softplus-like transformation, creating a self-regularizing property that helps maintain gradient flow in deep networks.

Smish builds upon this foundation by introducing a tunable scaling parameter that allows the function to adapt its output range dynamically during training. The Rectified Sigmoid Linear Unit modifies the traditional sigmoid function by applying a rectification that centers the outputs while preserving useful gradient signals.

TanhExp merges exponential growth with hyperbolic tangent saturation, producing an activation that balances positive domain growth with controlled negative value handling. The Parametric Flattened Linear Unit offers configurable linear regions through learnable thresholds, enabling networks to adapt their activation slopes to different layers' needs.

Finally, the Growing Cosine Unit exhibits unique oscillatory behavior in its positive regime while smoothly transitioning to exponential decay for negative inputs, providing a rich feature space for learning complex patterns. These advanced activation functions each bring distinct advantages in managing gradient flow, maintaining neuron activity, and supporting stable training across various network architectures. Their designs reflect ongoing innovations in addressing the fundamental challenges of deep learning optimization.

The activation functions included in this study were carefully selected based on their core mathematical properties that directly impact neural network training effectiveness. Our evaluation framework considered several crucial operational aspects: the function's response to very large or small inputs (saturation characteristics), its ability to maintain stable error propagation during backward passes (gradient transmission), the symmetry of its output distribution (mean-centered properties), and the preservation of input ordering relationships (monotonic behavior). These fundamental attributes work in concert to govern critical training outcomes - including optimization stability, convergence rates, and



ultimate model accuracy - across various network depths and architectures. The selected functions embody distinct design philosophies for handling these properties, ranging from elementary binary activation patterns to complex, self-adjusting formulations, thereby enabling a thorough examination of activation function design principles in contemporary deep learning systems.

## **2.2 Dataset Acquisition and Preprocessing**

To establish a robust and unbiased assessment protocol, we utilized several standard image classification benchmarks in our experiments. All datasets underwent identical preprocessing procedures to guarantee uniformity across several critical dimensions: input value scaling followed consistent normalization schemes, image dimensions were standardized, and identical partitioning strategies were applied when creating training, validation, and test subsets.

This standardized approach eliminated dataset-specific variations that could potentially skew comparative results, while the diversity of the selected benchmarks helped validate the general applicability of our findings across different problem domains and data characteristics. The uniform processing pipeline encompassed all critical preprocessing stages including but not limited to mean subtraction, channel normalization, and proper data augmentation techniques - all implemented consistently across datasets to isolate the specific impact of activation function choice on model performance.

The datasets used include:

- CIFAR-10
- CIFAR-100
- Fashion-MNIST

### 2.2.1 CIFAR-10

A standardized benchmark dataset consisting of 60,000  $32 \times 32$  RGB images distributed across 10 mutually exclusive object classes (6,000 images per class). The dataset is partitioned into 50,000 training and 10,000 test images with uniform class distribution. Images exhibit significant intra-class variation and lighting/pose diversity while maintaining consistent resolution and aspect ratio. The constrained spatial resolution necessitates effective feature learning despite limited pixel information.



Figure 2.3: A Subset of CIFAR-10 Dataset

### 2.2.2 CIFAR-100

An enhanced variant of CIFAR-10 featuring identical image dimensions ( $32 \times 32$  RGB) and dataset size (60,000 total images) but with 100 fine-grained categories organized into 20 superclasses. Each class contains precisely 500 training and 100 test images, introducing challenges of limited training samples per category. The hierarchical label structure enables evaluation of both fine-grained classification (100 classes) and coarse-grained recognition (20 superclasses).



Figure 2.4: A Subset of CIFAR-100 Dataset

### 2.2.3 Fashion-MNIST

Developed as a modern replacement for the classic MNIST handwritten digit dataset, Fashion-MNIST presents a more challenging classification task while maintaining the same convenient scale and structure. The dataset contains exactly 70,000 grayscale images ( $28 \times 28$  pixels) spanning 10 categories of fashion products, including trousers, dresses, sneakers and shirts. These images are divided into a standard 60,000-image training set and 10,000-image test set, mirroring MNIST's partitioning scheme.

Unlike MNIST's simple black-and-white digits, Fashion-MNIST introduces several real-world complexities: garments exhibit significant texture variations (knit patterns, woven fabrics), frequent shape occlusions (overlapping sleeves, folded clothing), and diverse silhouettes. Each image undergoes rigorous preprocessing - pixel intensities are linearly scaled to the  $[0,1]$  range, with spatial distributions centered to eliminate positional bias. The dataset provides standardized mean and standard deviation values 0.2860 and 0.3530

respectively for immediate normalization.

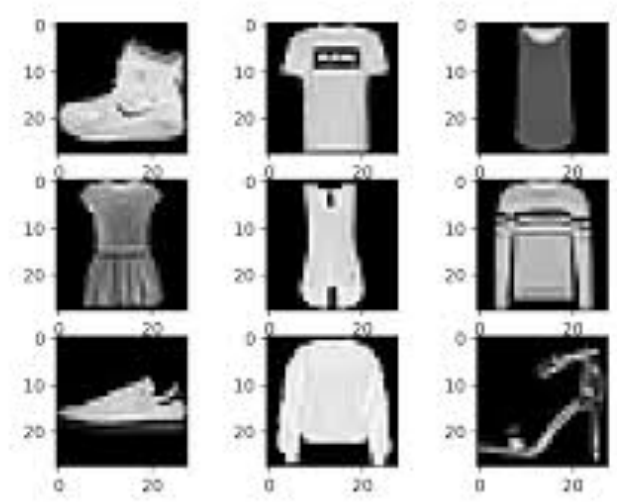


Figure 2.5: A Subset of Fashion-MNIST Dataset

The input images will undergo standardized preprocessing where pixel values will be scaled to the  $[0,1]$  range through division by 255. To enhance model generalization, we will implement a data augmentation pipeline during training that will include random horizontal flips (50% probability), rotations ( $\pm 10$  degrees), and where applicable, random cropping with zero-padding. These transformations will be consistently applied across all experiments to maintain comparable training conditions while evaluating each activation function's robustness. The augmentation parameters will remain fixed throughout the experiments to ensure fair comparison between different activation functions.

## 2.3 Network Architecture and Training Configuration

This study tested the main model architecture of Convolutional Neural Networks (CNNs) for image classification tasks using the Fashion-MNIST, CIFAR-10, and CIFAR-100 benchmark datasets. Furthermore, a secondary architecture for comparative analyses of sequentially arranged data was developed using Recurrent Neural Networks (RNNs). To assess their effects on classification performance, both topologies employed a variety of experimental and modern activation functions.

### 2.3.1 Convolutional Neural Network (CNN) Architecture

Implementing TensorFlow's Sequential API, the CNN model was built in a small but expressive manner to capture hierarchical spatial characteristics. The architecture was composed of the following sequence of layers:

- Two consecutive convolutional layers meant to gradually extract hierarchical visual features will be part of the proposed architecture. Employing "same" padding to preserve the input spatial dimensions over the convolution operation, the first convolutional layer will use 32 learnable filters with a  $3 \times 3$  kernel size. By means of its learned receptive fields, this layer will capture low-level elements including edges, textures, and simple patterns. Using 64 filters (also with  $3 \times 3$  kernels and "same" padding) to detect higher-level feature combinations and more complex visual patterns, the network will pass these feature maps to a second convolutional layer following batch normalization and activation.
- To guarantee dense spatial sampling of the input feature maps, stride=1 will be used in both layers. Although the network can develop a richer feature representation hierarchy due to the doubling of filters, the uniform  $3 \times 3$  kernel size across layers creates an effective receptive field growth pattern ( $5 \times 5$  effective in the second layer through stacked convolutions). The well-known CNN principle of progressively deepening feature maps while decreasing spatial dimensions through subsequent pooling operations is adhered to in this design.

- **Batch Normalisation:** To expedite and stabilise the training process, a Batch Normalisation layer was applied following each convolutional layer in order to normalise the output activations.
- After every Batch Normalization execution, the model architecture will use Lambda layers to optimally enable flexible activation functions. This arrangement will permit the selection and testing of eleven activation functions, namely ReLU, Tanh, Sigmoid, hcLSH, Penalized Tanh, ELiSH, Mish, RSigELU, TanhExp, PFlu, and GCU, dynamically within a single model. Instead of applying preset nonlinear modifications, all Lambda layers will receive feature maps that have been normalized from Batch Normalization, sent sequentially, and each will parameterize the transformation, applying the assigned activation function. With the focus on varying the non-linear properties of the model while keeping the topological structure and training parameters unchanged across these experiments, the modular design approach offers systematic evaluation of the network components. Within the Lambda layers, the activation functions will be implemented as custom TensorFlow operations, ensuring efficient forward and backward computation with gradient retention for proper backpropagation.

## 2.4 Training Configurations

- **Optimization Strategy:** We will use the Adam optimizer with an initial learning rate of 0.001, taking advantage of its adaptive momentum and per parameter learning rate mechanics to convergence on all activation function types efficiently. This method of optimization will apply a step size correction that is proportional to the history of the gradient, taking advantage of both the AdaGrad and RMSProp techniques.

- **Formulation of Loss:** In the case of multi-class classification problems, the main loss function will be Sparse Categorical Crossentropy. This is the best choice for an integer-encoded class label because it computes cross-entropy without the overhead of one-hot encoding, using predicted probability distributions and true class labels.
- **Evaluation Protocol:** The evaluation of the model's performance will be done using two complementary metrics. Training accuracy enables monitoring the learning progress whilst Validation accuracy measures generalization capability. The evaluation will be done and recorded after every epoch in order to comprehend the convergence behavior of various activation functions.
- **Regularization Framework:** A stratified approach to regularization will be applied. Each activation is stabilized by Batch Normalization layers placed after every convolutional and dense layer. Co-adaptation of features is minimized using Dropout with rates: early layers = 0.3; later layers = 0.4. If preliminary studies suggest model overfitting, L2 weight regularization will be added.
- **Training Duration:** Every model configuration will train for a total of 50 epochs. This was determined as adequate by prior studies to allow Exhaustive convergence on the loss landscape, Convergence on accuracy metrics stability, Ability to distinguish different activation function behaviors with a clear margin.
- **Computational Infrastructure:** All tests will run on GPU-boosted hardware with: Flexible memory growth assignment to avoid resource conflicts Self-adjusting mixed-precision training when possible Spread-out training options for big-scale experiments This setup makes sure we can repeat our results while getting the most out of our hardware across many test runs happening at once.

Managing these factors will single out how the choice of activation function affects the experiment. At the same time, it ensures the best training conditions are kept up throughout the research.

## 2.5 Experimental Automation and Result Logging Framework

The research aims to put into action a automated testing process. This will make sure the assessment is thorough and can be repeated for all dataset and activation function pairings.

The system will carry out this standard set of steps:

- The system trains the model based on each dataset and activation function pairing. It records important metrics like loss and accuracy trends during every epoch of training. Once training finishes, the process stores the optimized model weights in a standard format. This format keeps the final parameters ready to analyze later.
- The automated setup creates and saves detailed training graphs that show how each activation function converges. These visuals track learning patterns during training and reveal how stable and efficient the activation functions are. It keeps all performance numbers like final test accuracy and loss organized in CSV files. By storing everything in one place, this system helps to compare results making it simple to see how different activation functions work on various datasets.
- The framework will use strict version control and track experiments to make sure everything is reproducible. Each run will include complete metadata like hyperparameters, system settings, and environment specifics. The system will save outputs such as model weights, performance stats, and training curves in a arranged directory structure with time-stamped versions. This setup ensures results can be reproduced and checked without issues while keeping a clear record of all the conditions during experiments.
- The automated setup will use quality control steps to find and deal with training problems. These measures will check for things like training going off track, gradients becoming unstable, or hardware breaking down keeping the results reliable. Removing manual steps and making all experiments follow the same process will allow activation functions to be compared . It will also handle scaling to test many



setups. Detailed logs and result tracking will help run strong statistical analysis and create data ready to include in publications.

# Chapter 3

## Implementation and Work Done

### 3.1 Setting Up the Environment and Configuring

We set up the experimental framework using Python 3.11 along with TensorFlow 2.16 as the main library for deep learning tasks. We used additional tools like NumPy, Matplotlib, and Keras utilities to support the overall setup. The environment was optimized to use GPU acceleration, which plays a big role in training deep convolutional models with large datasets like CIFAR-100 and SVHN. We configured TensorFlow to detect the GPUs available using its device tools and enabled dynamic memory allocation. This allowed the GPU to slowly take memory as needed during training instead of grabbing all the available memory upfront. With this setup, so that we could run multiple models at the same time on the 40GB VRAM GPU container available courtesy of the AI lab.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # Suppress TensorFlow INFO logs

import tensorflow as tf

# Check if GPU is available
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        # Enable memory growth to avoid pre-allocating all GPU memory
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        print("GPU is available and memory growth is enabled.")
        print("Using GPU:", tf.config.list_logical_devices('GPU'))
    except RuntimeError as e:
        print("RuntimeError while setting GPU configuration:", e)
else:
    print("No GPU found. Using CPU.")
```

Figure 3.1: Configuring the Environment and connecting to the GPU Access

## 3.2 Importing Libraries and Setting Up Dependencies

We started by picking and importing key Python libraries to build, train, and test deep learning models. They relied on NumPy to handle numerical tasks and used Matplotlib to create graphs showing accuracy and loss trends. TensorFlow version 2.16 acted as the main framework offering tools to create, train, and test models while also including low-level functions to define custom activation operations with backend math. Within TensorFlow, Keras helped with loading datasets compiling models, and managing callbacks. We used SciPy's IO module to load the SVHN dataset saved in a MATLAB format. Libraries like OS handled file tasks, time tracked execution duration, and pickle saved results. To avoid issues during the experiment, We made sure all library versions matched .

```
import json
import matplotlib.pyplot as plt
import csv
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, Lambda
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import fashion_mnist, cifar10, cifar100
import tensorflow.keras.backend as K
from scipy.io import loadmat
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import load_model
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Define folder structure
BASE_DIR = "saved_models"
PLOT_DIR = "training_graphs"
RESULTS_DIR = "results"

os.makedirs(BASE_DIR, exist_ok=True)
os.makedirs(PLOT_DIR, exist_ok=True)
os.makedirs(RESULTS_DIR, exist_ok=True)
```

Figure 3.2: importing the libraries

### 3.3 Using Activation Functions

used activation functions to design and fine-tune neural networks. These functions play a role in deciding whether a neuron will fire or stay inactive. We allow models to capture and process complex data patterns in tasks like classification or prediction.

To help neural networks, activation functions introduce non-linearity, which makes simple networks capable of solving harder problems. Without them neural models would handle basic linear relationships. Every activation function has strengths and weaknesses that depend on the problem being solved.

Popular choices like ReLU sigmoid, and tanh have different uses. ReLU short for rectified linear unit, is common for deep learning because it works with large networks . Sigmoid often used in binary classification tasks, squashes outputs into values between 0 and 1. Tanh similar in form, operates on a range from -1 to 1 offering benefits in handling negative values too.

To improve performance, We selected activation functions that align with the network's architecture or the type of data. Choosing the wrong activation function could create problems like vanishing or exploding gradients, which can make training impossible or unstable.

```

# Custom activation functions
def penalized_tanh(x, alpha=0.25): # alpha is a hyperparameter (default 0.25)
    return tf.where(x >= 0, tf.tanh(x), alpha * tf.tanh(x))

def elish(x):
    return x * tf.nn.sigmoid(x) * (1 + tf.nn.tanh(x))

def mish(x):
    return x * tf.nn.tanh(K.softplus(x)) # softplus(x) = log(1 + exp(x))

def rsigelu(x):
    return x * tf.nn.sigmoid(x) + tf.nn.elu(x)

def tanh_exp(x):
    return x * tf.tanh(K.exp(x))

def pflu(x, alpha=0.1, beta=1.0):
    return tf.where(x >= 0, x + beta, alpha * (tf.exp(x) - 1))

def gcu(x):
    return x * tf.cos(x)

def hcLSH(x):
    return x * tf.nn.sigmoid(x) * K.log(K.sigmoid(x) + K.epsilon())

activation_functions = {
    "relu": tf.nn.relu,
    "tanh": tf.nn.tanh,
    "sigmoid": tf.nn.sigmoid,
    "hcLSH": hcLSH,
    "penalized_tanh": penalized_tanh,
    "elish": elish,
    "mish": mish,
    "rsigelu": rsigelu,
    "tanh_exp": tanh_exp,
    "pflu": pflu,
    "gcu": gcu
}

```

Figure 3.3: Custom Definition of Activation Functions

A big part of this project included creating and using both basic and advanced activation functions even ones that TensorFlow doesn't support by default. Functions like Mish, elish, Penalized Tanh, rSiGeLU, TanhExp, GCU, PFLU, and hcLSH were written with TensorFlow's backend tools. These were built step by step using basic math operations from TensorFlow such as exponential, sigmoid, hyperbolic tangent, and ReLU. A dictionary was made to match the name of each activation function with its code, so models could switch between functions. Keras Lambda layers were used to implement these custom activations letting them run as part of the graph without causing compatibility problems.

### **3.4 Preparing and Processing the Dataset**

This step involves organizing and cleaning up the data to use it . Preparing the dataset includes tasks like sorting, labeling, or selecting specific parts of the data. Processing the dataset focuses on transforming it into a usable format by handling missing values, normalizing, or scaling. These steps make the data ready to be analyzed and ensure better results during analysis.

The experiments worked with four known datasets: Fashion-MNIST, CIFAR-10, CIFAR-100, and SVHN. Keras' dataset API handled loading Fashion-MNIST, CIFAR-10, and CIFAR-100, while the SciPy IO library was used to load SVHN since it comes in a MATLAB file format. Each dataset's pixel data was scaled to floating-point numbers between zero and one to speed up gradient descent training. Labels were either kept as integer indices for sparse categorical crossentropy or turned into one-hot encoded vectors using TensorFlow's categorical tools. All images were reshaped and preprocessed to keep tensor dimensions uniform. CIFAR images were adjusted to 32 by 32 with three color channels, and Fashion-MNIST images were reshaped into 28 by 28 grayscale.

### **3.5 Structuring the Model's Architecture**

The main architecture chosen for classification employed a Convolutional Neural Network to interpret spatial patterns in image data. The network began with two convolutional layers that used small three-by-three filters and padding to maintain spatial size after processing. After each convolution step, batch normalization was added to balance the outputs and support smoother learning. Activation functions weren't built into the convolutional layers. Instead, Lambda layers handled activations so different functions could be tested throughout experiments.

```

def create_cnn_model(input_shape, activation_name, num_classes):
    activation_func = activation_functions[activation_name]

    model = Sequential([
        Conv2D(filters=32, kernel_size=(3, 3), padding="same", input_shape=input_shape),
        BatchNormalization(),
        Lambda(activation_func),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(filters=64, kernel_size=(3, 3), padding="same"),
        BatchNormalization(),
        Lambda(activation_func),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(256),
        Lambda(activation_func),
        Dropout(0.3),
        Dense(128),
        Lambda(activation_func),
        Dropout(0.4),
        Dense(num_classes, activation="softmax") # Multi-class classification
    ])

    model.compile(optimizer=Adam(learning_rate=0.001), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return model

```

Figure 3.4: Model Architecture

After each convolution and normalization segment, max pooling reduced the size of the feature maps by half while keeping important details. These maps were then flattened before moving through a pair of connected dense layers. Dropout was applied to help prevent overfitting in the dense layers. The softmax activation function was used in the final dense layer to create class probabilities.

### **3.6 Implementing a Recurrent Neural Network Variant**

Alongside convolutional models, We developed a version using Recurrent Neural Networks to test how activation functions perform in handling sequential data. This model relied on SimpleRNN layers, with the first recurrent layer set to return sequences. Returning sequences allowed the next layer to process the complete output sequence instead of just one vector. We used the same Lambda layer approach to integrate custom activation functions into the RNN cells. After the recurrent layers, the model included dense layers and ended with a softmax layer designed to make multi-class predictions.

### **3.7 Compiling, Training, and Validating the Model**

The Adam optimizer helped compile each model and is popular due to its adaptive learning rate and momentum updates. To work with integer-encoded class labels sparse categorical crossentropy was chosen as the loss function. The models were trained for fifty epochs using a batch size of one hundred twenty-eight samples. Out of the training data, ten percent was set aside to validate performance and watch how well the model generalized. TensorFlow's history callback saved the accuracy and loss numbers measured during both training and validation stages.

### **3.8 Storing and Displaying the Model**

After finishing the training loops, TensorFlow's saving tools helped save the model weights in H5 file format. This made it possible to reload and test the models later instead of training them from scratch. Matplotlib was used to show the training progress, like accuracy and loss changes during different epochs. For each dataset and activation function combination, graphs were made to highlight how training stayed stable how it converged, and how different activation functions affected performance.



### 3.9 Plotting the Graphs

A dedicated function is implemented for the exclusive purpose of visualization of training progress using the library known as Matplotlib, one of the most widely adopted plotting libraries Python for scientific research in the world. This function, `plottraining()`, was designed to receive the training history object returned by Keras, along with the dataset and activation function names as parameters. It was programmatically extracted the accuracy, validation accuracy, training loss, and validation loss values recorded at each epoch during the training cycle and it is for that purposes that we have done these steps to get these results. These metrics were then scaled appropriately for readability, with accuracy values converted to percentages and loss values preserved as-is.

The visualization layout consisted of a figure with two subplots: the first plotting training and validation accuracy curves across epochs, and the second depicting the corresponding loss curves. Each plot was equipped with axis labels, legends, and descriptive titles dynamically generated based on the dataset and activation function involved in that particular experiment. This dynamic labeling enhanced the interpretability of the visual outputs, ensuring that each plot could be directly associated with its corresponding model configuration.

### 3.10 Documentation of Evaluation and Results

We tested trained models on specific datasets to measure final accuracy and loss. TensorFlow tools were used to calculate these values. We saved the results in organized CSV files. Each entry listed the dataset name, activation type, accuracy, loss, and training duration. Recording these results helped compare numbers and examine patterns during review. The testing process ran . It went through all the dataset and activation function combinations step by step. This made sure tests followed the same steps each time and could be repeated.

# Chapter 4

## Results

We compared activation functions using three popular image classification datasets: CIFAR-10, CIFAR-100, and Fashion MNIST. We trained each model with the same optimizer, learning rate, and loss function to keep conditions consistent. Training ran for a set number of epochs. At the end, We measured test loss and accuracy to evaluate how well each activation function performed.

### 4.1 Performance on CIFAR-10 Dataset:

The CIFAR-10 dataset, made up of 60,000 color images in  $32 \times 32$  resolution spread across 10 categories, revealed the Penalized Tanh activation function as the best performer. It hit about 73.26% test accuracy beating well-known activation functions like ReLU and Mish. Tanh-Exp and EliSH also gave strong results, with test accuracies of 73.47% and 73.18% . ReLU often a standard choice, reached a test accuracy of 71.24%. The custom-made HcLSH function delivered a decent accuracy of 68.99%. However, the GCU activation function struggled significantly on this dataset. It had an alarmingly high test loss exceeding 31% and achieved about 10% accuracy suggesting issues with convergence and possible numerical instability in the training process.

The plots of each activation function's performance on CIFAR-10 Dataset is as follows:

#### 4.1.1 GCU Function

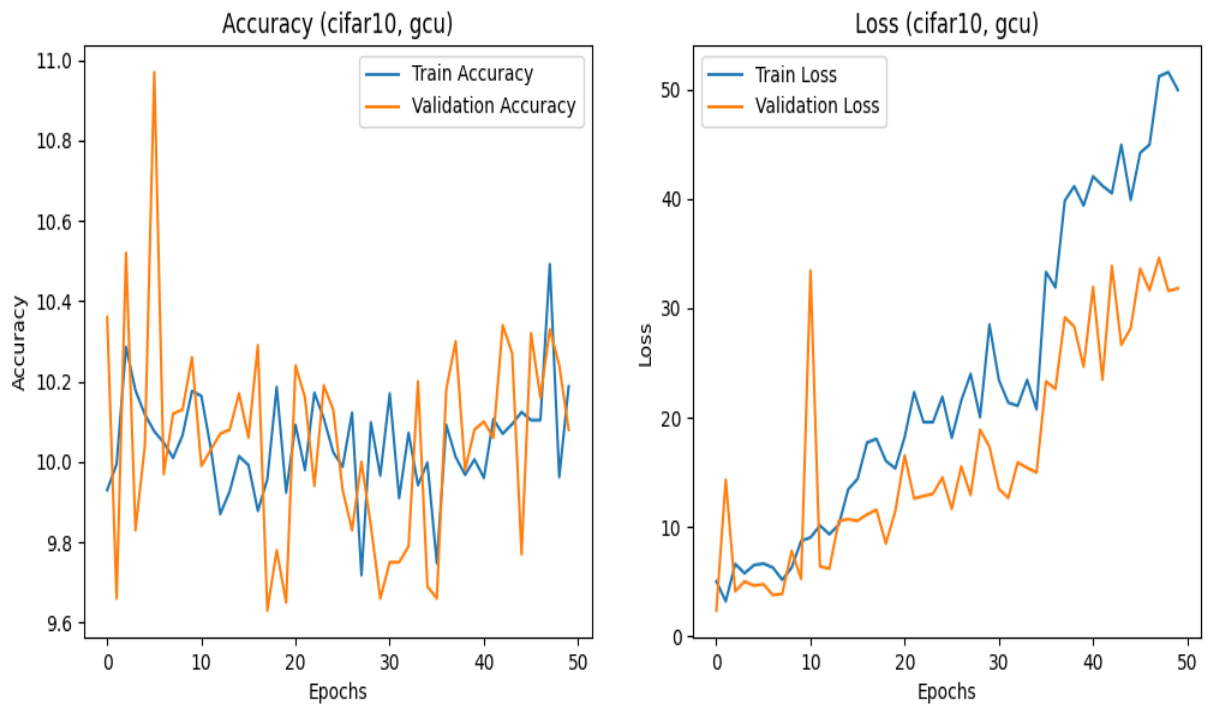


Figure 4.1: GCU Function on CIFAR-10 Dataset

### 4.1.2 eLiSH Function

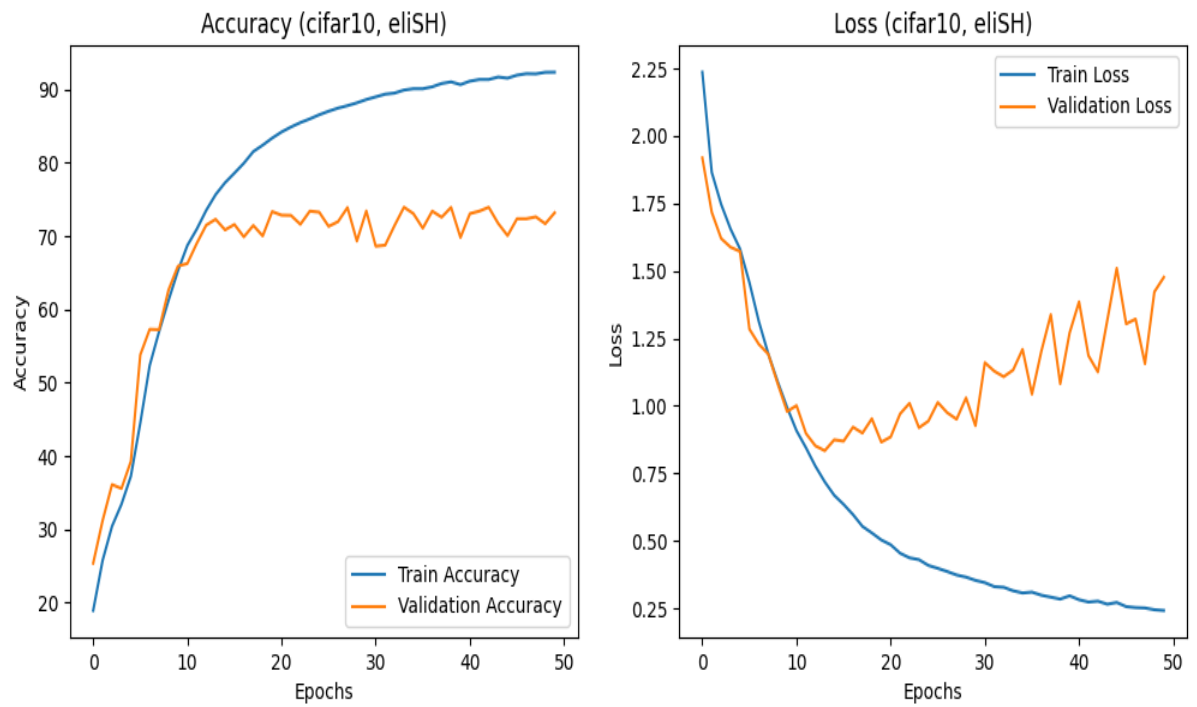


Figure 4.2: eLiSH Function on CIFAR-10 Dataset

### 4.1.3 hcLSH Function

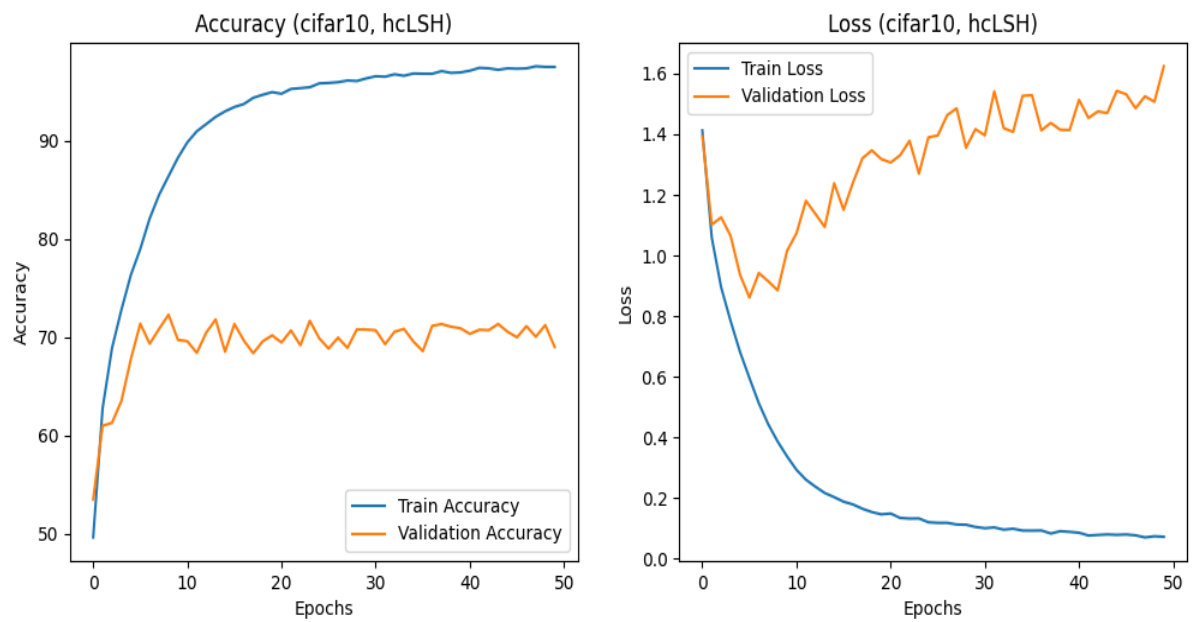


Figure 4.3: hcLSH Function on CIFAR-10 Dataset

#### 4.1.4 mish Function

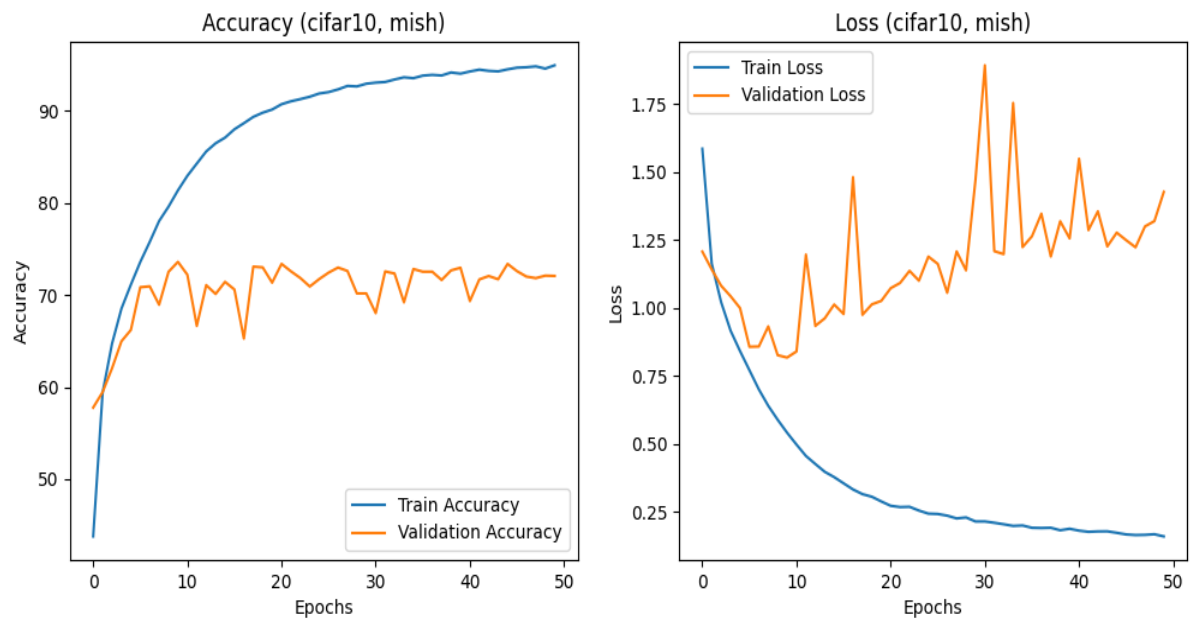


Figure 4.4: mish Function on CIFAR-10 Dataset

#### 4.1.5 Penalized Tanh Function

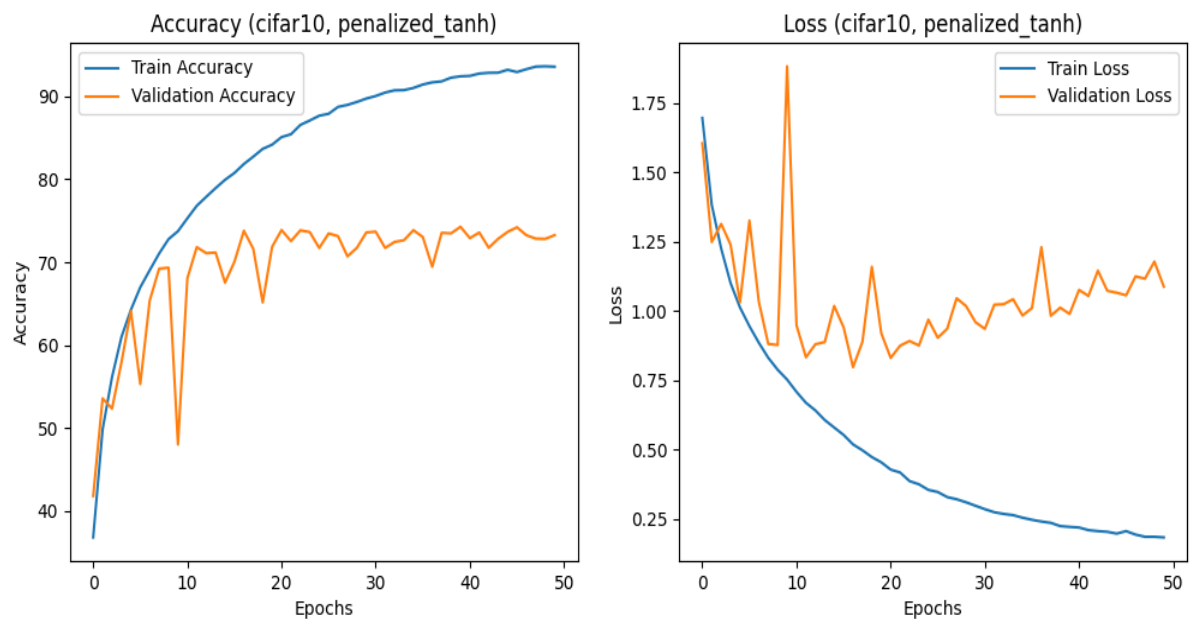


Figure 4.5: Penalized Tanh Function on CIFAR-10 Dataset

### 4.1.6 Relu Function

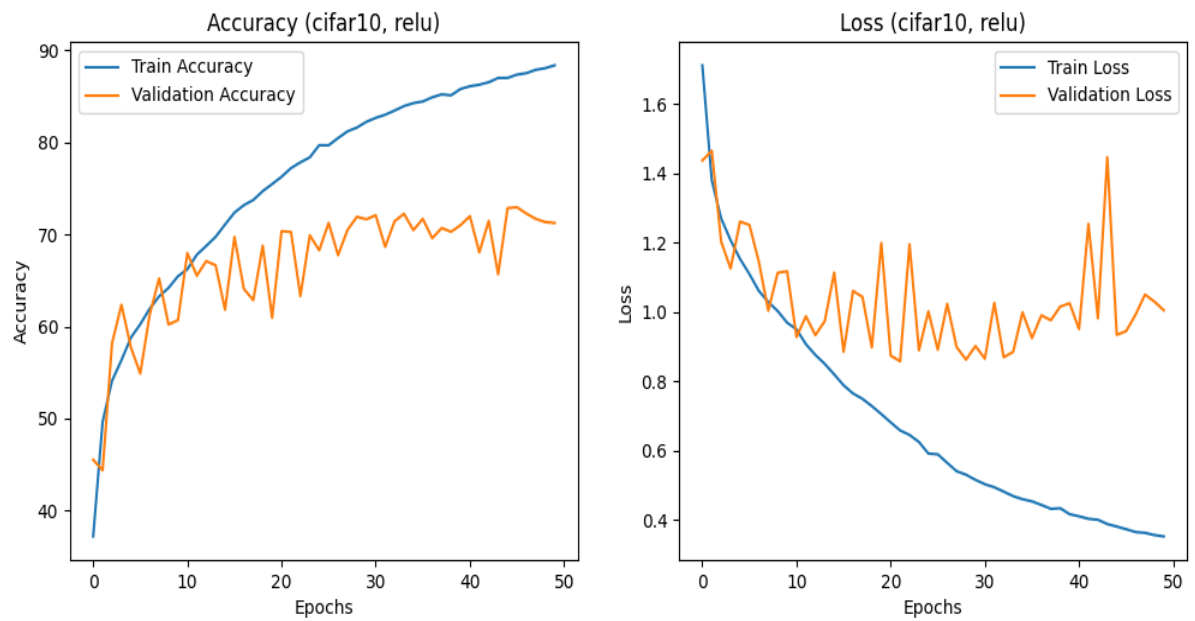


Figure 4.6: Relu Function on CIFAR-10 Dataset

### 4.1.7 rsigelu Function

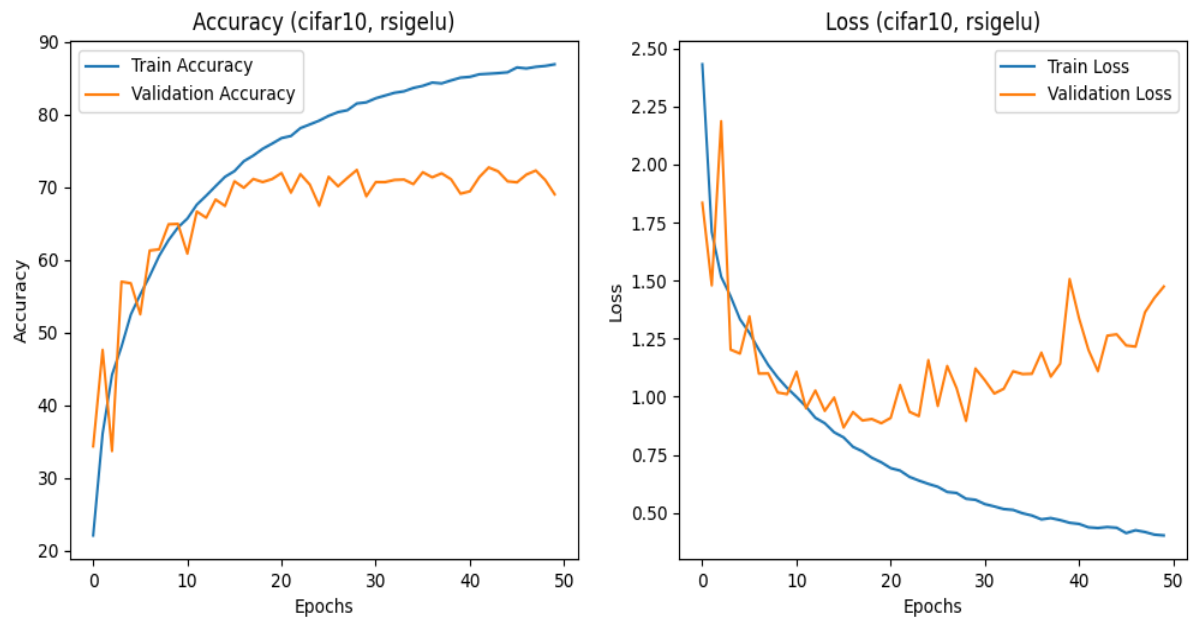


Figure 4.7: rsigelu Function on CIFAR-10 Dataset

### 4.1.8 Sigmoid Function

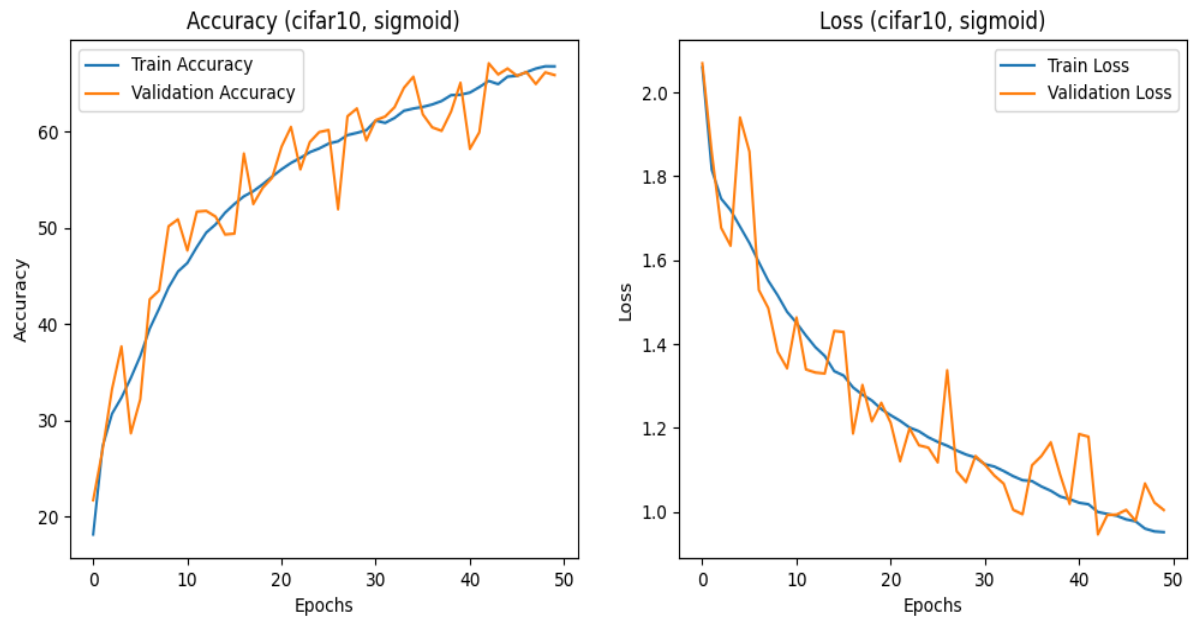


Figure 4.8: Sigmoid Function on CIFAR-10 Dataset

### 4.1.9 Tanh Exp Function

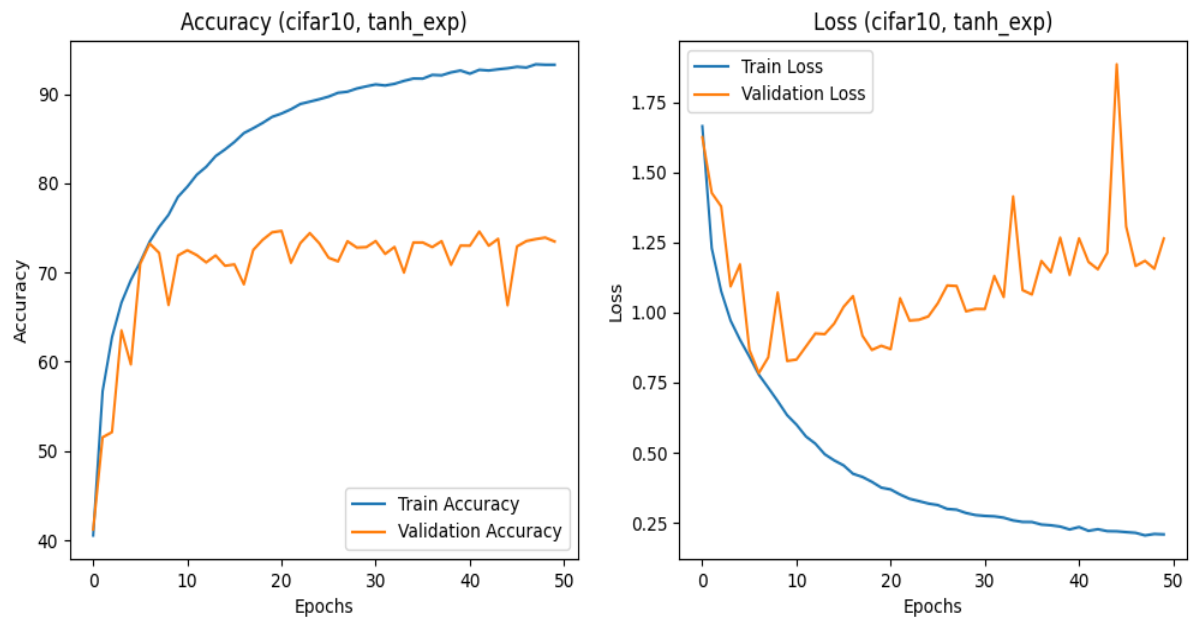


Figure 4.9: Tanh Exp Function on CIFAR-10 Dataset

#### 4.1.10 Tanh Function

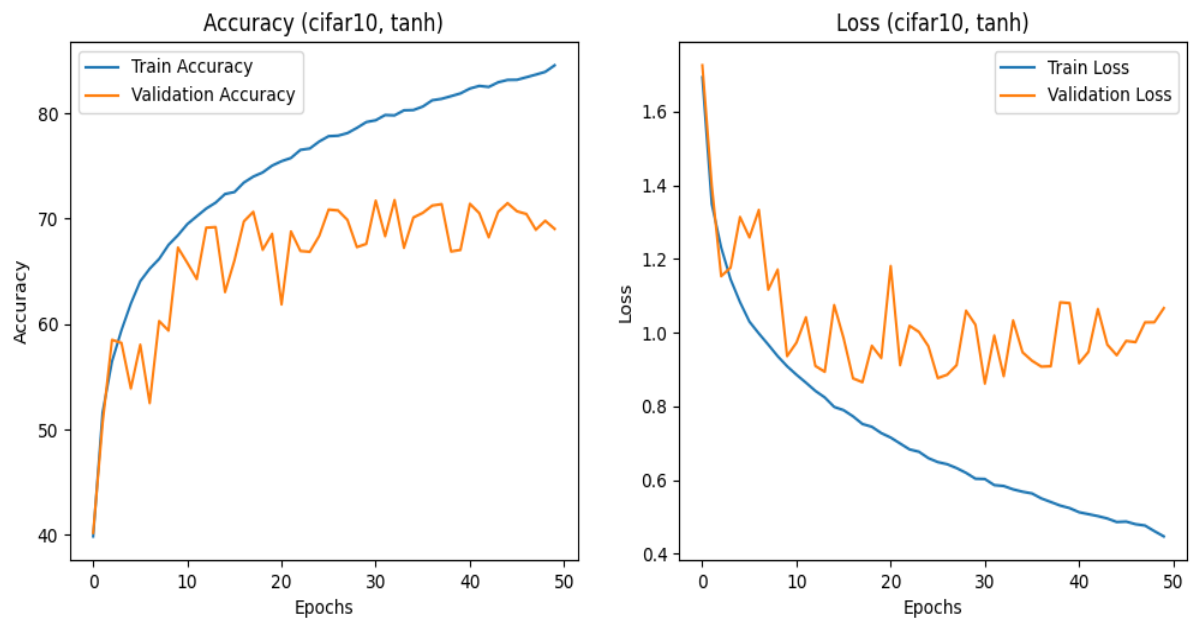


Figure 4.10: Tanh Function on CIFAR-10 Dataset



## 4.2 Performance on CIFAR-100 Dataset:

The CIFAR-100 dataset, which has 100 detailed categories and is harder to work with, showed Penalized Tanh leading the performance charts. It reached a test accuracy of 41.19%. Tanh followed behind at 35.85% while Mish scored 33.51%. The popular ReLU function achieved a test accuracy of 32.19%, which was rather low. HcLSH performed to ReLU with a test accuracy of 32.44%. However, it did not show clear benefits during this specific test. EliSH and GCU struggled a lot. EliSH hit 1% accuracy, and GCU behaved very poorly, with its test loss crossing 121. This stressed how some activation functions struggle more when faced with deeper and more detailed classification problems.

The plots of each activation function's performance on CIFAR-100 Dataset is as follows:

### 4.2.1 GCU Function



Figure 4.11: GCU Function on CIFAR-100 Dataset

### 4.2.2 eLiSH Function

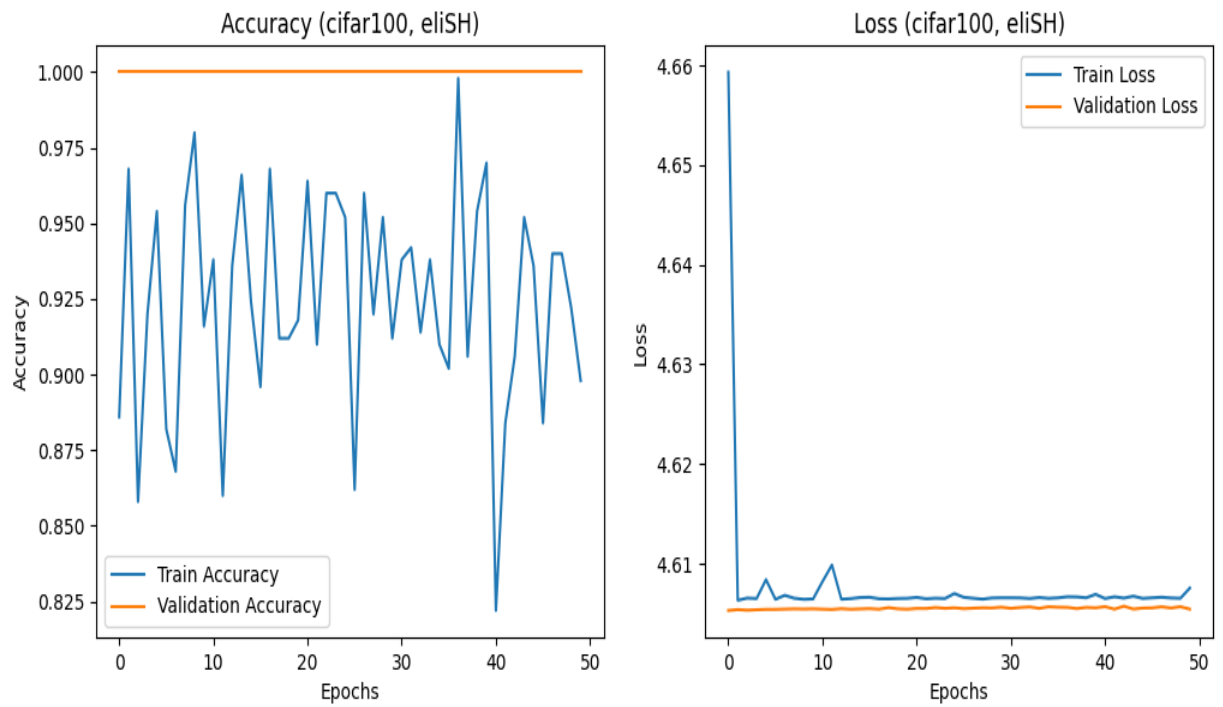


Figure 4.12: eLiSH Function on CIFAR-100Dataset

### 4.2.3 hcLSH Function

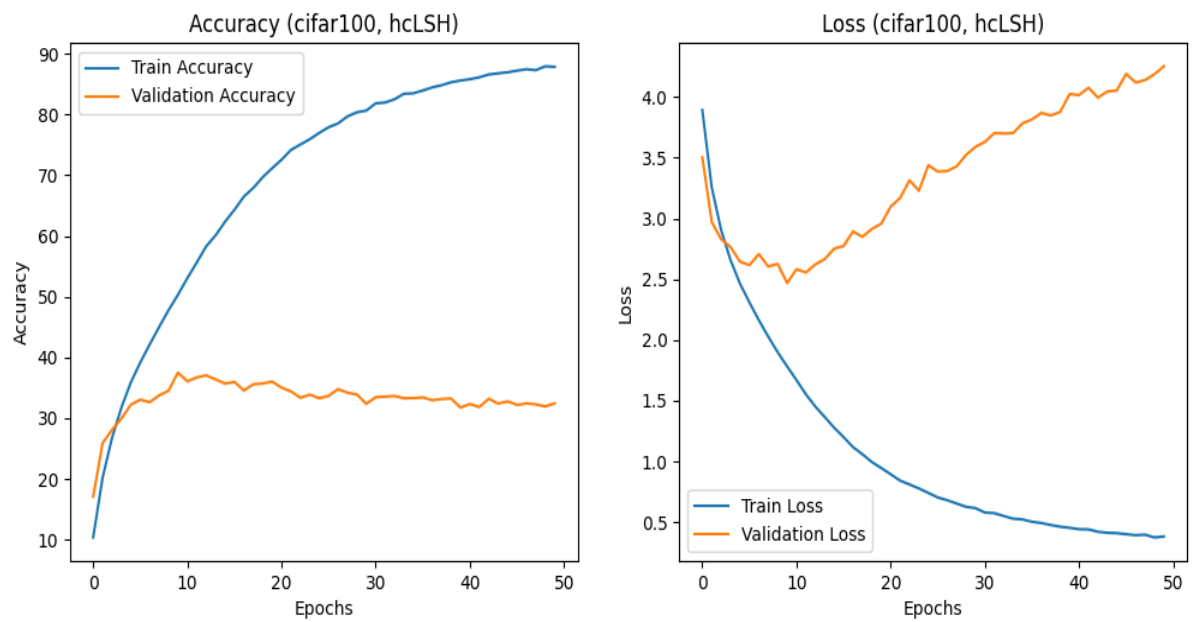


Figure 4.13: hcLSH Function on CIFAR-100 Dataset

## 4.2.4 mish Function

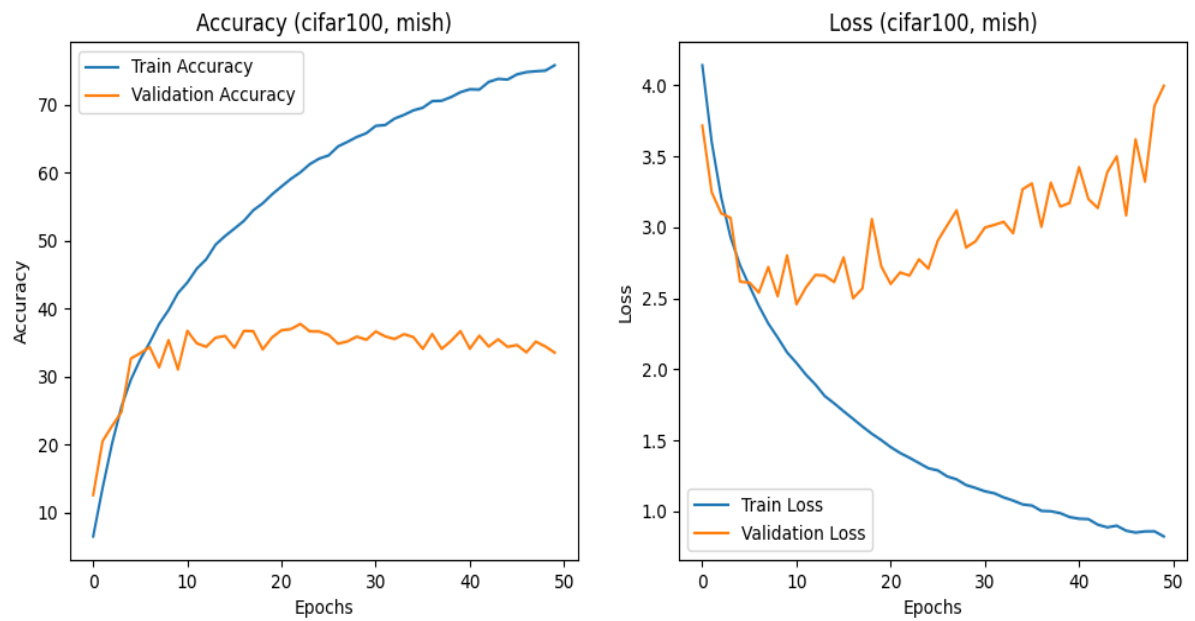


Figure 4.14: mish Function on CIFAR-100 Dataset

## 4.2.5 Penalized Tanh Function

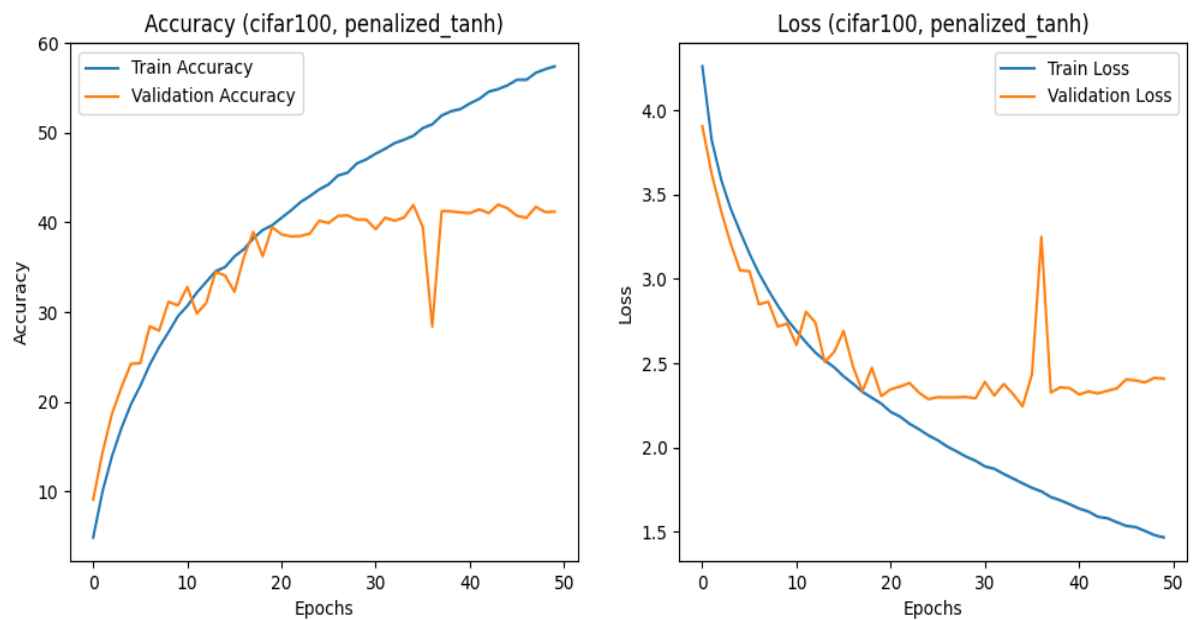


Figure 4.15: Penalized Tanh Function on CIFAR-100 Dataset

## 4.2.6 Relu Function

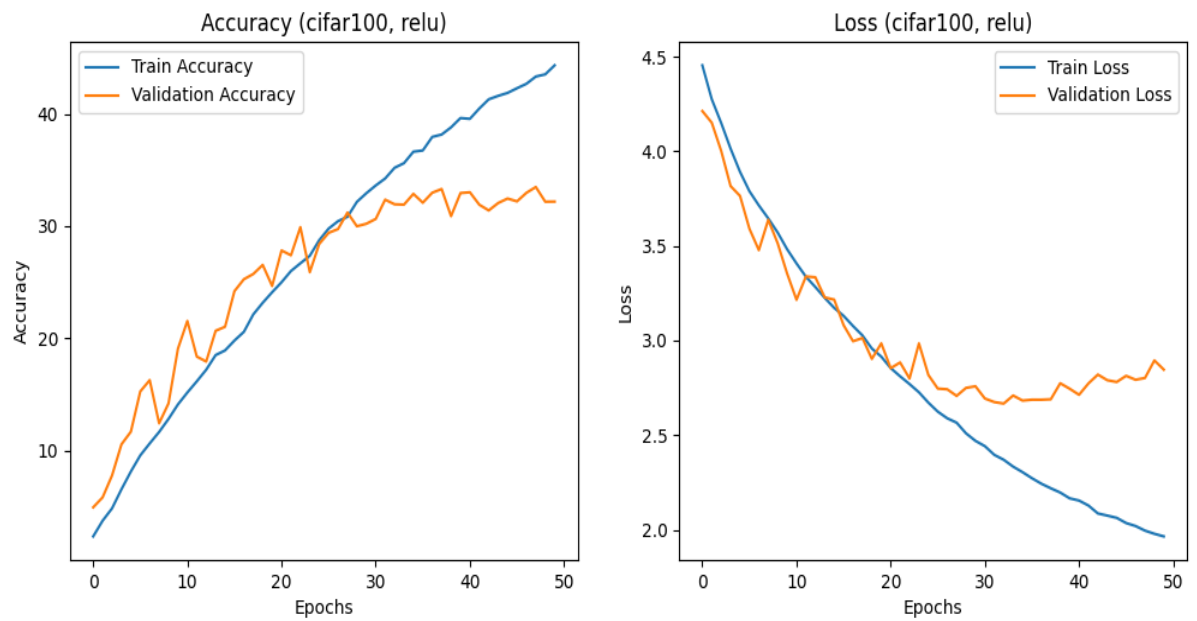


Figure 4.16: Relu Function on CIFAR-100 Dataset

## 4.2.7 rsigelu Function

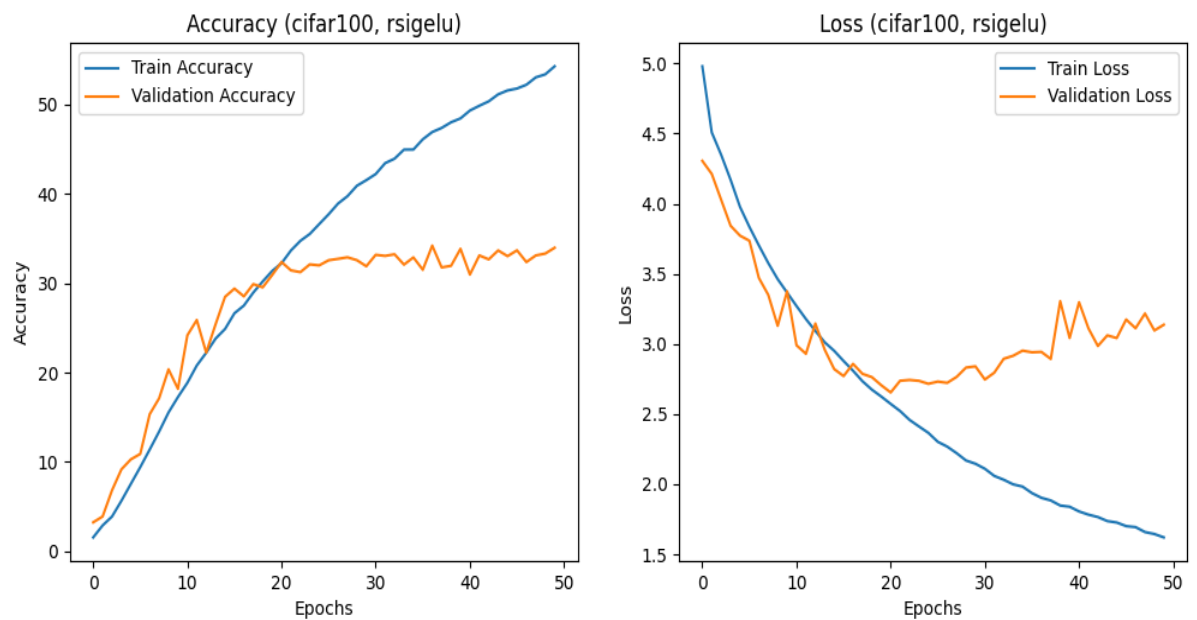


Figure 4.17: rsigelu Function on CIFAR-100 Dataset

### 4.2.8 Sigmoid Function

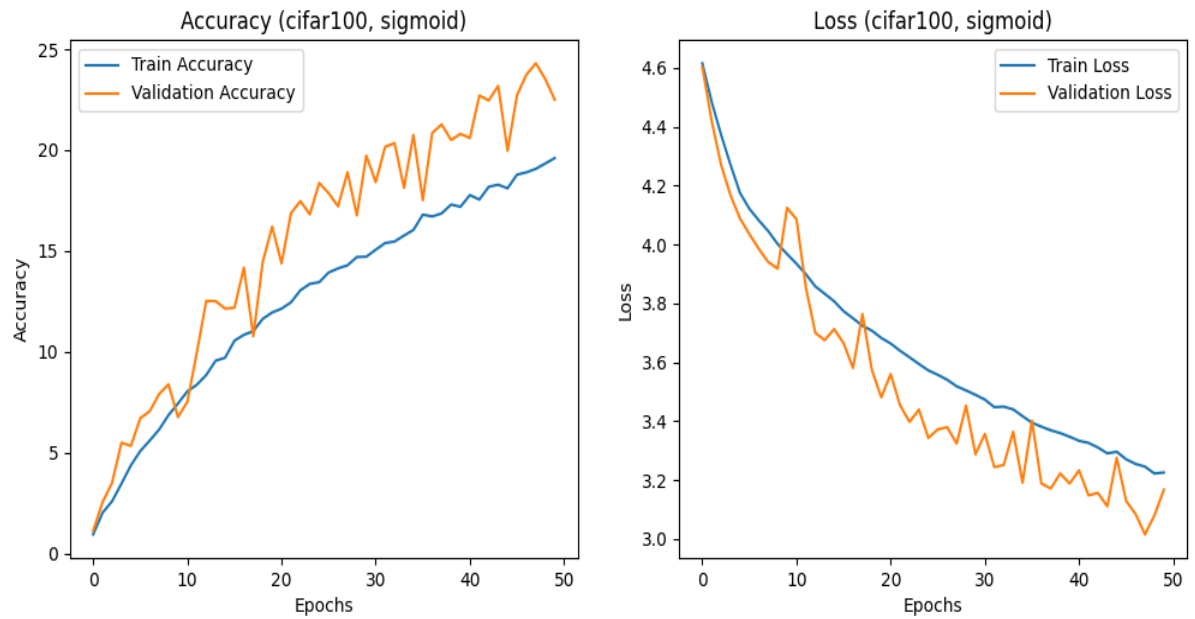


Figure 4.18: Sigmoid Function on CIFAR-100 Dataset

### 4.2.9 Tanh Exp Function

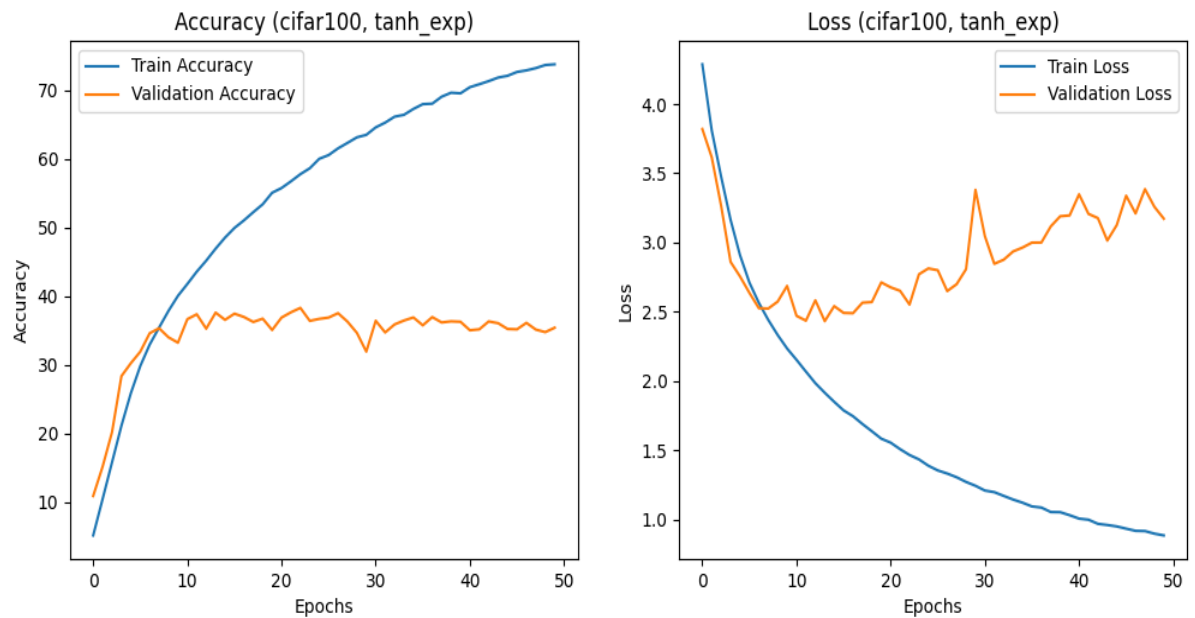


Figure 4.19: Tanh Exp Function on CIFAR-100 Dataset

### 4.2.10 Tanh Function

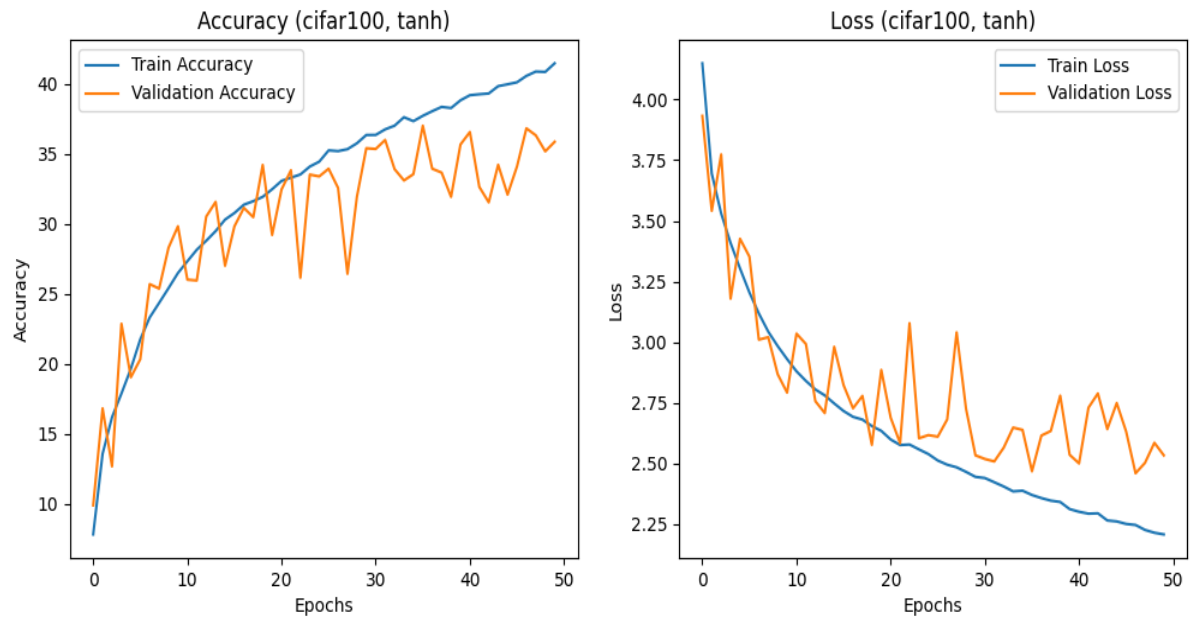


Figure 4.20: Tanh Function on CIFAR-100 Dataset

### 4.3 Performance on Fashion-MNIST Dataset:

The Fashion MNIST dataset contains 70,000 grayscale images of clothing items split into 10 classes. Most activation functions worked better on this dataset compared to CIFAR datasets because it is less complicated. Penalized Tanh led the way with the highest accuracy at about 91.99%. ReLU Rsgelu, and HcLSH weren't far behind scoring 91.86%, 91.89%, and 91.82% . On the other hand, Tanh-Exp was a complete failure here. It hit a non-numeric loss (NaN) and stayed stuck at 10% accuracy, which is no better than random guessing. GCU didn't impress either, with a bloated test loss over 53 and accuracy above random levels at 10.93%.

**The plots of each activation function's performance on Fashion-MNIST Dataset is as follows:**

#### 4.3.1 GCU Function

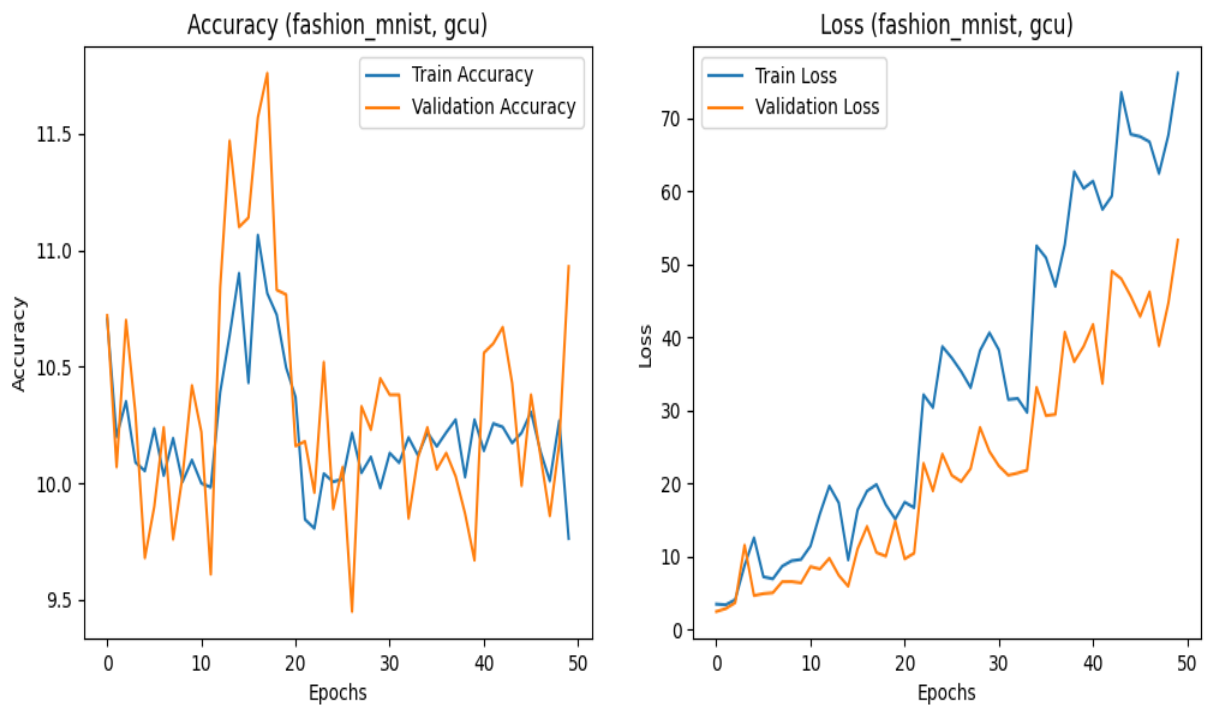


Figure 4.21: GCU Function on Fashion-MNIST Dataset

### 4.3.2 eLiSH Function

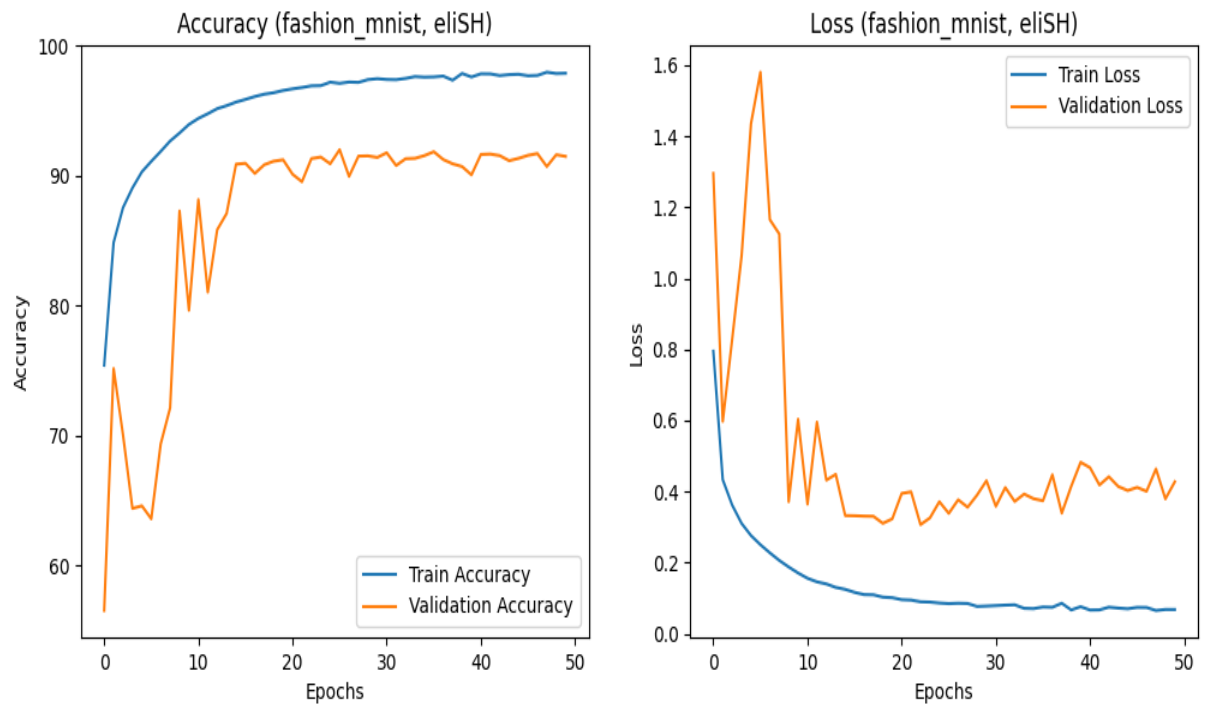


Figure 4.22: eLiSH Function on Fashion-MNIST Dataset

### 4.3.3 hcLSH Function

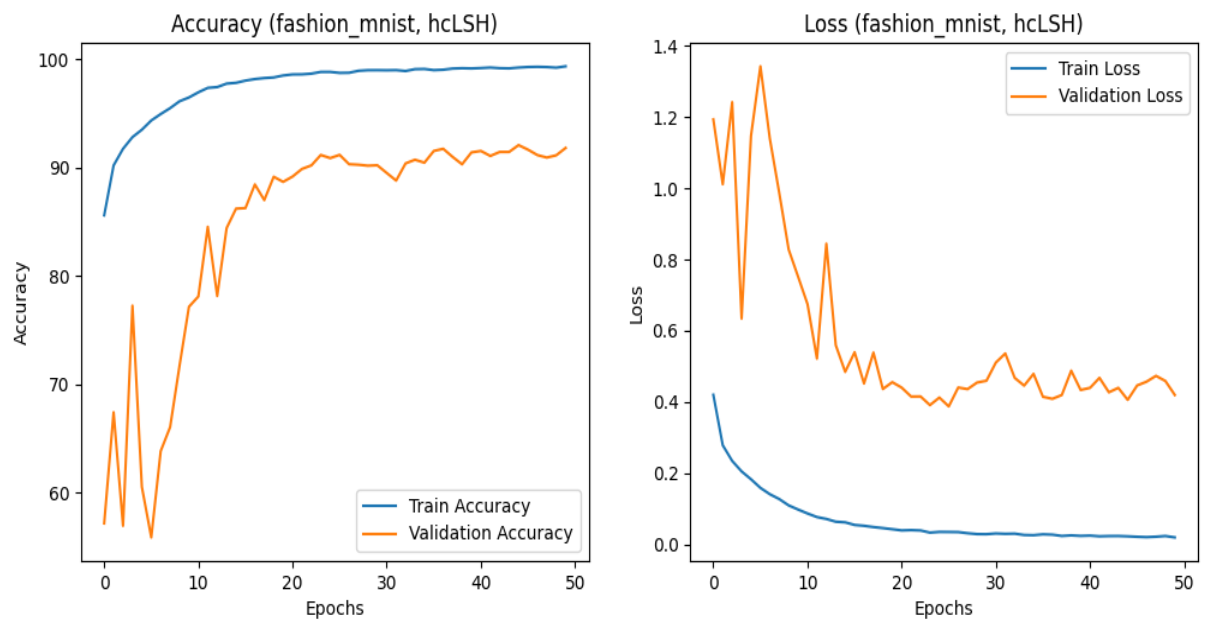


Figure 4.23: hcLSH Function on Fashion-MNIST Dataset



### 4.3.4 mish Function

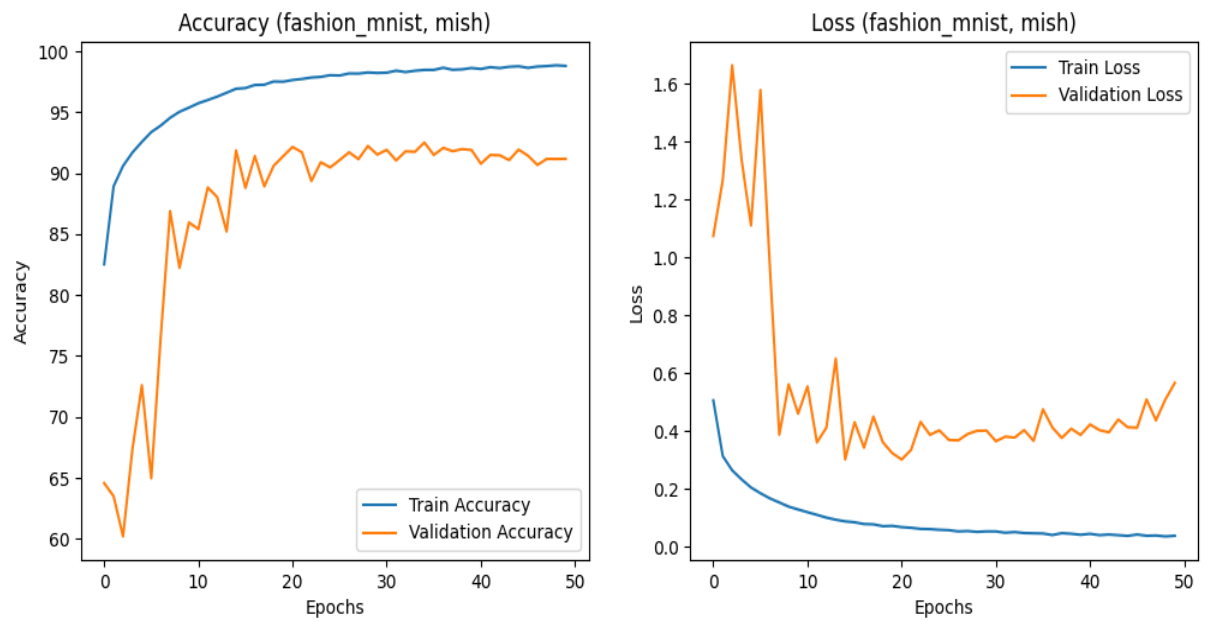


Figure 4.24: mish Function on Fashion-MNIST Dataset

### 4.3.5 Penalized Tanh Function



Figure 4.25: Penalized Tanh Function on Fashion-MNIST Dataset

### 4.3.6 Relu Function

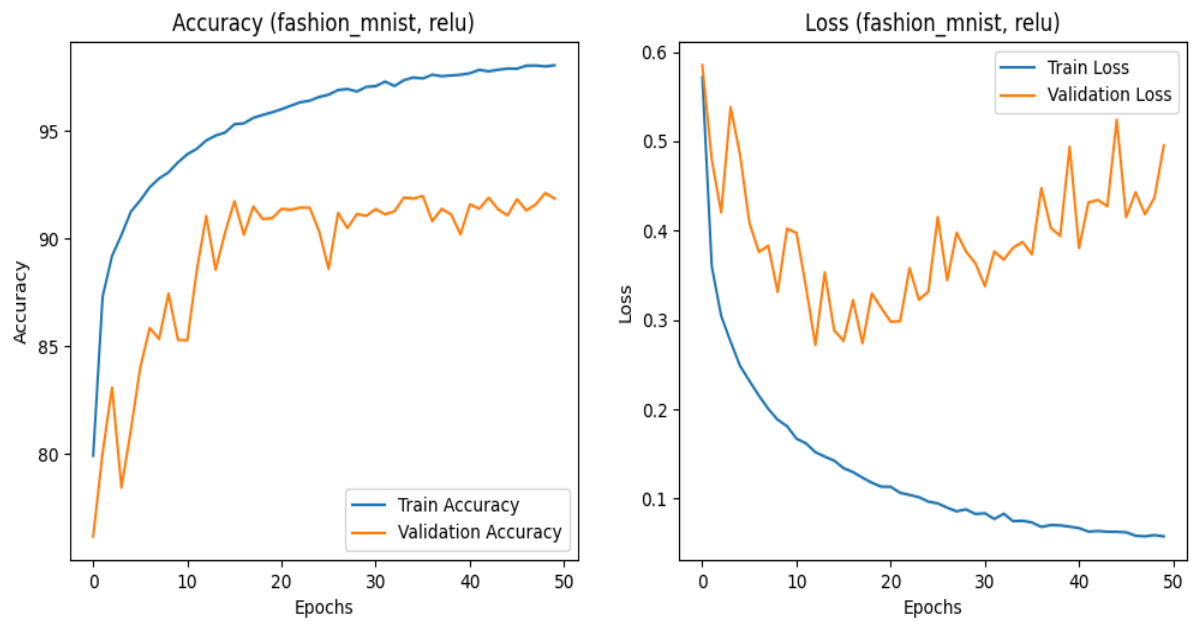


Figure 4.26: Relu Function on Fashion-MNIST Dataset

### 4.3.7 rsigelu Function

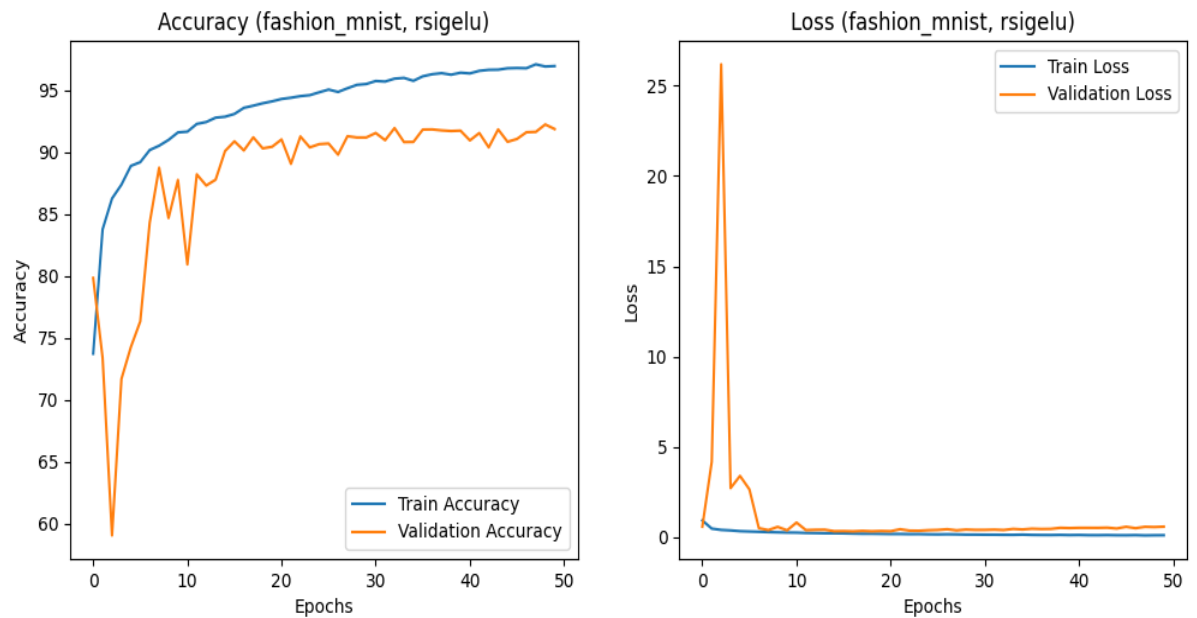


Figure 4.27: rsigelu Function on Fashion-MNIST Dataset

### 4.3.8 Sigmoid Function

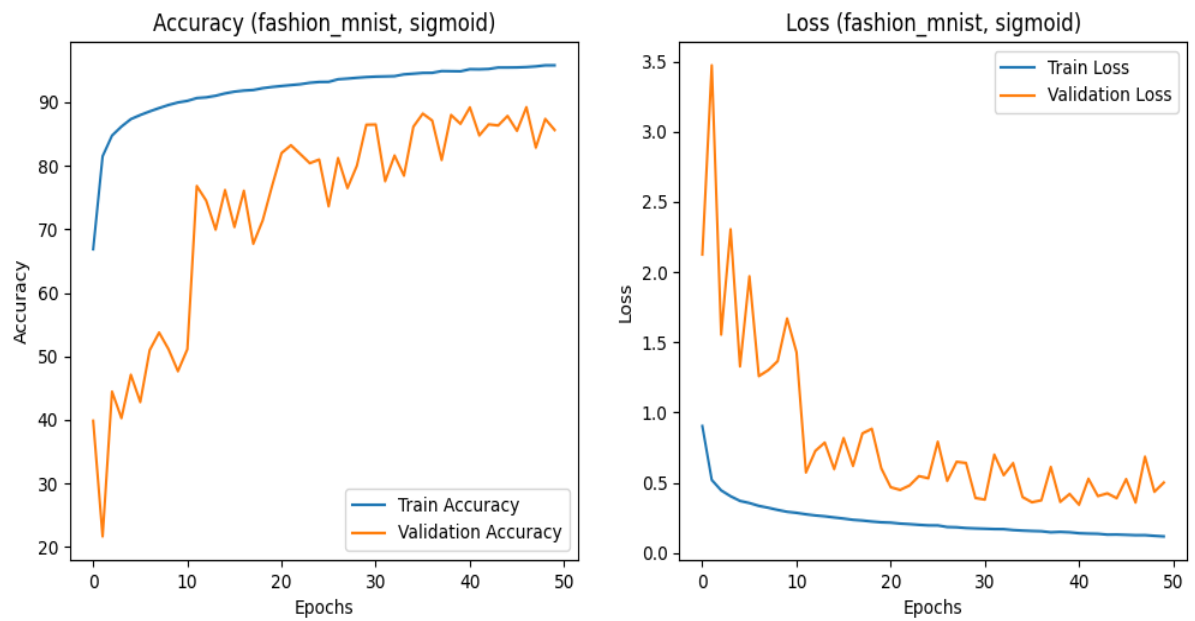


Figure 4.28: Sigmoid Function on Fashion-MNIST Dataset

### 4.3.9 Tanh Exp Function

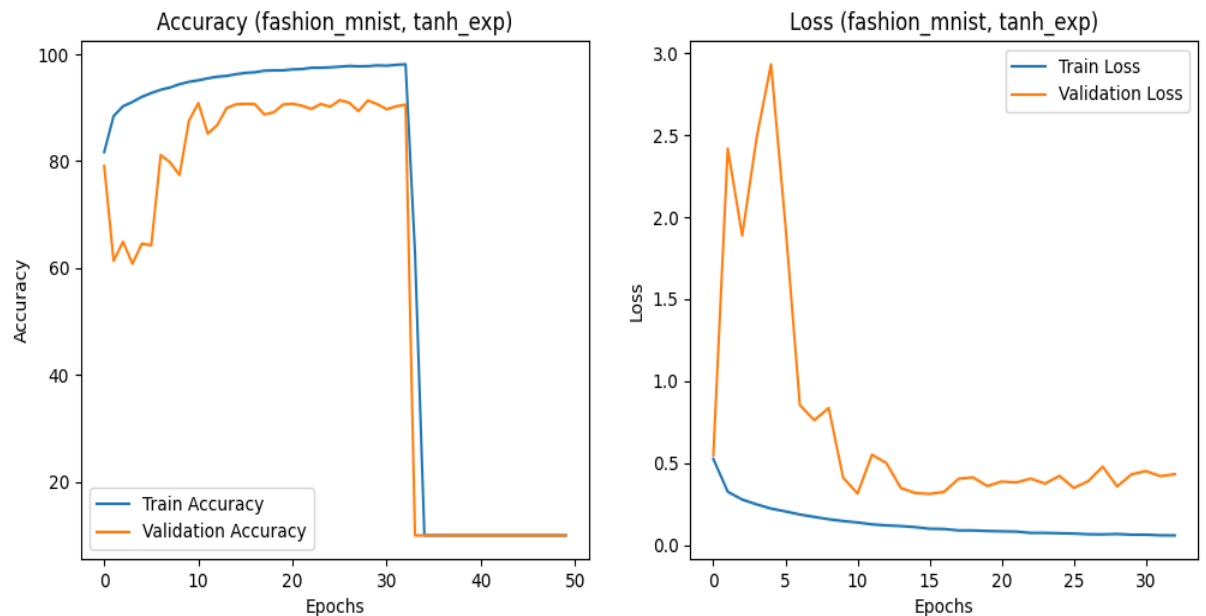


Figure 4.29: Tanh Exp Function on Fashion-MNIST Dataset

### 4.3.10 Tanh Function

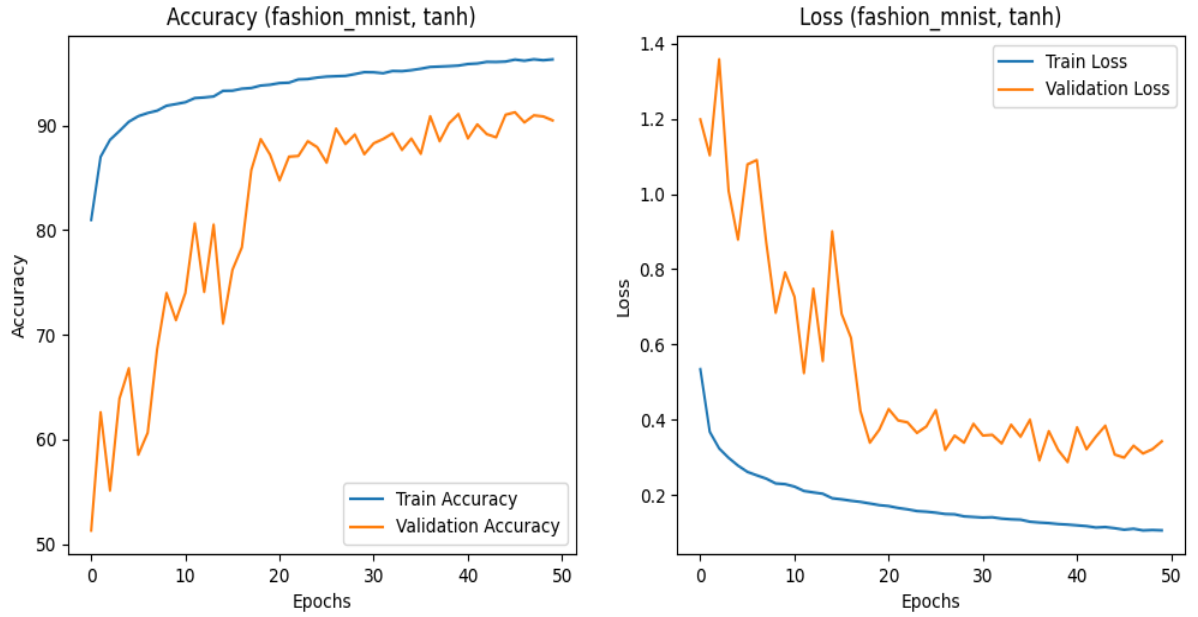


Figure 4.30: Tanh Function on Fashion-MNIST Dataset

The results showed that newer activation functions such as Mish, EliSH, and Rsgelu seemed promising but failed to match the dependability of Penalized Tanh or even the older ReLU in a wide range of classification tasks. The proposed HcLSH function performed well on easier datasets but fell short when compared to well-known alternatives on challenging datasets like CIFAR-100 with many categories. The experiments also highlighted how differently activation functions perform depending on the dataset, which underscored the need to tune activation functions specifically to suit each dataset in real-world deep learning tasks.

## 4.4 Conclusion

This study looks at how different activation functions in deep neural networks work and compares them. It focuses on the Hybrid Continuous Leaky Sigmoid Hyperbolic (HcLSH) function alongside both older and newer methods. The findings show that classic functions like ReLU and its variations, while popular for being simple and quick still face challenges like dead neurons and non-zero-centered outputs. These are the issues which can make training deeper networks very much harder. In contrast most of the modern functions like Swish, Mish, PFLU, and HcLSH show much better results. They allows models to train much faster than before, improve the flow of gradient , and even perform better overall when tested on various datasets present.

The HcLSH function is unique because it combines the reliable nature of sigmoidal functions with the adaptable design of hyperbolic activations. This combination solves key problems in training deep neural networks. Research demonstrates that HcLSH helps reduce challenges like exploding and vanishing gradients. It ensures models stay effective at managing complex tasks making it a good fit for advanced designs. Its ability to adjust saturation levels and maintain smooth flow of gradient which in turn provides stability during training that surpasses older approaches by a huge margin. Studies also confirm it speeds up training and enhances model accuracy.

These results provide valuable insights to help deep learning experts and professionals. Choosing the correct activation function requires understanding how large the network is and how challenging the task might be. Recent functions like HcLSH appear effective in solving complicated issues. The results from these modern activation methods show exciting possibilities to explore later. This might include testing them in fields like reinforcement learning and generative modeling or pairing them with neural architecture search strategies. Hybrid activation methods highlight their potential showing there are still many ways to create new ideas in building neural networks.

Looking ahead, researchers have to investigate some important questions. Limited-resource scenarios require more study on balancing performance gains of advanced activation functions with their computational demands. Efforts could very likely shift toward examining how the features of activation functions shape model behavior to better understand them. With deep learning growing, picking or designing the right activation functions will be the key to boosting the model performance across various applications. This study hugely contributes by providing useful tips and tricks and examples to help make these crucial design choices.

# Chapter 5

## Future Work

Our experiments with both old and new activation functions in image classification point toward interesting areas to explore further. This area is growing with activation designs that might speed up training, help gradients flow better, and make models more accurate. Future studies should look at functions like SwiGLU with its gated linear units, GEGLU’s gating system, FReLU’s ability to adjust to spatial features, MetaAcon’s flexible thresholds, DY-ReLU’s context-aware behavior, and new memristor-based methods. Each of these approaches offers unique strengths that could be useful for certain tasks.

Our research explored computer vision tasks, but the logical next move is to test these activation functions on other types of data. Text applications in natural language processing, time-series predictions using sequential data, and classification tasks with structured data offer unique challenges. These challenges could respond depending on the activation function used. Possible areas of use include predictive systems for financial markets, voice recognition technologies, healthcare diagnostics, and even emotion-based computing. In these fields adaptive activation behaviors might help manage shifting input patterns.

Practical considerations during deployment need further examination. Using complex activation functions like Mish or HcLSH might not be worth it in devices with limited resources such as IoT gadgets or smartphones, as the computational cost could outweigh any accuracy improvements. Engineers building actual systems might benefit from a detailed analysis of memory usage, energy needs, and the time taken for inference when

trying different activation functions.

There are several ways to improve how researchers study activation functions in the future:

- Using neural architecture search to find the best activation strategies
- Exploring new optimizer and activation function combinations instead of just relying on Adam
- Developing better tools to study how activation patterns and gradients behave
- Creating methods to visualize and interpret how each activation function contributes to learning

Our findings also does create a solid base for future generations to compare activation functions in vision-related tasks using various and numerous open source libraries for that purposes like OpenCV etc. Future studies which will be done in the future need to build on this by adding newer types of activation functions and do hard and rigorous testing on them across different use cases, and using a variety of evaluation methods while doing research in the future. This should aim not to improve how models work in real-world settings but also to provide a clearer understanding of the role nonlinearities play in neural networks.



# Chapter 6

## References

- G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006, doi: 10.1162/neco.2006.18.7.1527.
- F. Shao, L. Chen, J. Shao, W. Ji, S. Xiao, L. Ye, Y. Zhuang, and J. Xiao, “Deep learning for weakly-supervised object detection and localization: A survey,” *Neurocomputing*, vol. 496, pp. 192–207, Jul. 2022, doi: 10.1016/j.neucom.2022.01.095.
- Y. Mo, Y. Wu, X. Yang, F. Liu, and Y. Liao, “Review of state-of-the-art technologies of semantic segmentation based on deep learning,” *Neurocomputing*, vol. 493, pp. 626–646, Jul. 2022, doi: 10.1016/j.neucom.2022.01.005.
- S. R. Dubey, “A decade survey of content-based image retrieval using deep learning,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 32, no. 5, pp. 2687–2704, May 2022, doi: 10.1109/TCSVT.2021.3080920.
- L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, “DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 4, pp. 834–848, Apr. 2018, doi: 10.1109/TPAMI.2017.2699184.
- W. Duch and N. Jankowski, “Survey of neural transfer functions,” *Neural Comput. Surveys*, vol. 2, no. 1, pp. 163–212, 1999.


- L. Datta, “A survey on activation functions and their relation with Xavier and He normal initialization,” 2020, arXiv:2004.06632.
- X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, Jun. 2011, pp. 315–323. [Online]. Available: <https://proceedings.mlr.press/v15/glorot11a.html>.
- Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994, doi: 10.1109/72.279181.
- F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, “Learning activation functions to improve deep neural networks,” 2014, arXiv:1412.6830.
- A. Apicella, F. Donnarumma, F. Isgrò, and R. Prevete, “A survey on modern trainable activation functions,” *Neural Netw.*, vol. 138, pp. 14–32, Jun. 2021, doi: 10.1016/j.neunet.2021.01.026.
- E. Chai, W. Yu, T. Cui, J. Ren, and S. Ding, “An efficient asymmetric nonlinear activation function for deep neural networks,” *Symmetry*, vol. 14, no. 5, p. 1027, May 2022, doi: 10.3390/sym14051027.
- 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, doi: 10.1016/j.neucom.2022.06.111.
- Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient BackProp,” in *Neural Networks: Tricks of the Trade*, Springer, 2012, pp. 9–48, doi: 10.1007/978-3-642-35289-8\_3.
- X. Wang, Y. Qin, Y. Wang, S. Xiang, and H. Chen, “ReLTanh: An activation function with vanishing gradient resistance for SAE-based DNNs and its application to rotating machinery fault diagnosis,” *Neurocomputing*, vol. 363, pp. 88–98, Oct. 2019, doi: 10.1016/j.neucom.2019.07.017.

- A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *Proc. ICML*, vol. 30, no. 1, 2013, p. 3.
- B. Xu, R. Huang, and M. Li, “Revise saturated activation functions,” 2016, arXiv:1602.05980.
- V. Nair and G. Hinton, “Rectified linear units improve restricted Boltzmann machines,” in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814.
- A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.
- P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” 2017, arXiv:1710.05941.
- M. Basirat and P. M. Roth, “The quest for the golden activation function,” 2018, arXiv:1808.00783.
- M. Zhu, W. Min, Q. Wang, S. Zou, and X. Chen, “PFLU and FPFLU: Two novel non-monotonic activation functions in convolutional neural networks,” *Neurocomputing*, vol. 429, pp. 110–117, Mar. 2021, doi: 10.1016/j.neucom.2020.11.068.
- D. Misra, “Mish: A self regularized non-monotonic activation function,” 2019, arXiv:1908.08681.
- X. Wang, H. Ren, and A. Wang, “Smish: A novel activation function for deep learning methods,” *Electronics*, vol. 11, no. 4, p. 540, Feb. 2022, doi: 10.3390/electronics11040540.
- X. Liu and X. Di, “TanhExp: A smooth activation function with high convergence speed for lightweight neural networks,” *IET Comput. Vis.*, vol. 15, no. 2, pp. 136–150, Mar. 2021, doi: 10.1049/cvi2.12020.
- S. Kiliçarslan and M. Celik, “RSigELU: A nonlinear activation function for deep neural networks,” *Exp. Syst. Appl.*, vol. 174, Jul. 2021, Art. no. 114805, doi: 10.1016/j.eswa.2021.114805.

- D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (ELUs),” 2015, arXiv:1511.07289.
- M. Mithra Noel, A. L, A. Trivedi, and P. Dutta, “Growing cosine unit: A novel oscillatory activation function that can speedup training and reduce parameters in convolutional neural networks,” 2021, arXiv:2108.12943.

# Chapter 7

## Plagiarism Report



**btp\_8th\_sem\_activation\_functions**

Goals 73 Overall score

MAY, 2025


viii


DECLARATION

Department of Information Technology Delhi-110078, India


We, Aryan Sinha (2021UIT3019), Jatin Nagarwal (2021UIT3033), and Muddit Lal (2021UIT3060) of B. Tech., Department of Information Technology, hereby declare that the Project II - report titled "Comparative Analysis of Activation Functions" which is submitted by us to the Department of Information Technology, Netaji Subhas University of Technology, is original and not copied from source without proper citation. This work has not previously formed the basis for the award of any Degree.


Place: Delhi


 ?

 Formatting tools are not available.


12,374 words ^


 Review suggestions


 Write with generative AI


 Check for AI text & plagiarism


**Plagiarism and AI text check** APA ▾


 This section resembles AI text


 This section resembles AI text


 This section resembles AI text

 This section resembles AI text


 This section resembles AI text


 This section resembles AI text

 This section resembles AI text

 This text matches **Generative AI: The basics**

**Detected Reference**

 1% of your text matches external sources  
Matches were found on the web or in academic databases

 8% of your text has patterns that resemble AI text  
These patterns may show AI text or occur in your writing

# Chapter 8

## Appendix

Following are the Code Snippets of the script used in this project:

```
# Custom activation functions
def penalized_tanh(x, alpha=0.25): # alpha is a hyperparameter (default 0.25)
    return tf.where(x >= 0, tf.tanh(x), alpha * tf.tanh(x))

def elISH(x):
    return x * tf.nn.sigmoid(x) * (1 + tf.nn.tanh(x))

def mish(x):
    return x * tf.nn.tanh(K.softplus(x)) # softplus(x) = log(1 + exp(x))

def rsigelu(x):
    return x * tf.nn.sigmoid(x) + tf.nn.elu(x)

def tanh_exp(x):
    return x * tf.tanh(K.exp(x))

def pflu(x, alpha=0.1, beta=1.0):
    return tf.where(x >= 0, x + beta, alpha * (tf.exp(x) - 1))

def gcu(x):
    return x * tf.cos(x)

def hcLSH(x):
    return x * tf.nn.sigmoid(x) * K.log(K.sigmoid(x) + K.epsilon())

activation_functions = {
    "relu": tf.nn.relu,
    "tanh": tf.nn.tanh,
    "sigmoid": tf.nn.sigmoid,
    "hcLSH": hcLSH,
    "penalized_tanh": penalized_tanh,
    "elISH": elISH,
    "mish": mish,
    "rsigelu": rsigelu,
    "tanh_exp": tanh_exp,
    "pflu": pflu,
    "gcu": gcu
}
```

```

activation_colors = {
    "relu": "blue",
    "tanh": "green",
    "sigmoid": "red",
    "hcLSH": "purple",
    "penalized_tanh": "orange",
    "eliSH": "cyan",
    "mish": "magenta",
    "rsigelu": "brown",
    "tanh_exp": "pink",
    "pflu": "gray",
    "gcu": "olive"
}

def load_svhn():
    train_data = loadmat("path/to/train_32x32.mat")
    test_data = loadmat("path/to/test_32x32.mat")

    X_train = np.moveaxis(train_data["X"], -1, 0) # Move axis to match TF format
    y_train = train_data["y"].flatten() % 10 # Normalize labels (SVHN uses 1-10)

    X_test = np.moveaxis(test_data["X"], -1, 0)
    y_test = test_data["y"].flatten() % 10

    return X_train, X_test, y_train, y_test

```

```

def load_dataset(dataset_name):
    if dataset_name == "fashion_mnist":
        (X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
        X_train = X_train[..., np.newaxis] # Add channel dimension
        X_test = X_test[..., np.newaxis]

    elif dataset_name == "cifar10":
        (X_train, y_train), (X_test, y_test) = cifar10.load_data()

    elif dataset_name == "cifar100":
        (X_train, y_train), (X_test, y_test) = cifar100.load_data()

    elif dataset_name == "svhn":
        X_train, X_test, y_train, y_test = load_svhn()

    else:
        raise ValueError("Dataset not supported!")

    # Normalize data
    X_train, X_test = X_train / 255.0, X_test / 255.0

    return X_train, X_test, y_train, y_test

```



```

def create_cnn_model(input_shape, activation_name, num_classes):
    activation_func = activation_functions[activation_name]

    model = Sequential([
        Conv2D(filters=32, kernel_size=(3, 3), padding="same", input_shape=input_shape),
        BatchNormalization(),
        Lambda(activation_func),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(filters=64, kernel_size=(3, 3), padding="same"),
        BatchNormalization(),
        Lambda(activation_func),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(256),
        Lambda(activation_func),
        Dropout(0.3),
        Dense(128),
        Lambda(activation_func),
        Dropout(0.4),
        Dense(num_classes, activation="softmax") # Multi-class classification
    ])

    model.compile(optimizer=Adam(learning_rate=0.001), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return model


def create_rnn_model(activation_name, input_shape, num_classes):
    activation_fn = activation_functions[activation_name] # Default to Tanh if not found

    model = tf.keras.Sequential([
        tf.keras.layers.SimpleRNN(64, activation=activation_fn, return_sequences=True, input_shape=input_shape),
        tf.keras.layers.SimpleRNN(128, activation=activation_fn, return_sequences=False),
        tf.keras.layers.Dense(64, activation=activation_fn),
        tf.keras.layers.Dense(128, activation=activation_fn),
        tf.keras.layers.Dense(num_classes, activation="softmax") # Output layer
    ])

    model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return model

```

```

# Function to save model properly
def save_model(model_dir: str, dataset_name, activation_name):
    model_dir = os.path.join("saved_models", dataset_name)
    os.makedirs(model_dir, exist_ok=True) # Ensure directory exists

    # Save model weights (only weights, NOT full model)
    model_path = os.path.join(model_dir, f"cnn_{dataset_name}_{activation_name}.h5")
    model.save_weights(model_path) # Save weights
    print(f"Model weights saved: {model_path}")

    # Save metadata separately (since model structure cannot be serialized)
    metadata = {
        "dataset": dataset_name,
        "activation": activation_name,
    }
    metadata_path = os.path.join(model_dir, f"metadata_{dataset_name}_{activation_name}.json")
    with open(metadata_path, "w") as json_file:
        json.dump(metadata, json_file, indent=4)

    print(f"Metadata saved: {metadata_path}")

# Function to reload the model
def load_saved_model(dataset_name, activation_name, input_shape, num_classes):
    model_dir = os.path.join("saved_models", dataset_name)

    # Rebuild the model
    model = create_cnn_model(input_shape, activation_name, num_classes)

    # Load saved weights
    model_path = os.path.join(model_dir, f"cnn_{dataset_name}_{activation_name}.h5")
    model.load_weights(model_path)
    print(f"Model weights loaded: {model_path}")

    return model

```

```

# Function to plot training results
def plot_training(history, dataset_name, activation_name):
    fig, ax = plt.subplots(1, 2, figsize=(12, 5))

    acc = [a * 100 for a in history.history["accuracy"]]
    val_acc = [a * 100 for a in history.history["val_accuracy"]]

    loss = [a * 1 for a in history.history["loss"]]
    val_loss = [a * 1 for a in history.history["val_loss"]]

    epochs = range(len(acc))

    # Accuracy Plot
    ax[0].plot(epochs, acc, label="Train Accuracy")
    ax[0].plot(epochs, val_acc, label="Validation Accuracy")
    ax[0].set_title(f"Accuracy ({dataset_name}, {activation_name})")
    ax[0].set_xlabel("Epochs")
    ax[0].set_ylabel("Accuracy")
    ax[0].legend()

    # Loss Plot
    ax[1].plot(epochs, loss, label="Train Loss")
    ax[1].plot(epochs, val_loss, label="Validation Loss")
    ax[1].set_title(f"Loss ({dataset_name}, {activation_name})")
    ax[1].set_xlabel("Epochs")
    ax[1].set_ylabel("Loss")
    ax[1].legend()

    # Save plot
    plot_filename = f"{PLOT_DIR}/training_{dataset_name}_{activation_name}.png"
    plt.savefig(plot_filename)
    plt.close()
    print(f"📄 Plot saved as {plot_filename}")

```

```

datasets = ["fashion_mnist", "cifar10", "cifar100"]
activation_functions = {
    "relu": tf.nn.relu,
    "tanh": tf.nn.tanh,
    "sigmoid": tf.nn.sigmoid,
    "hclsh": hclsh,
    "penalized_tanh": penalized_tanh,
    "eliSH": eliSH,
    "mish": mish,
    "rsigelu": rsigelu,
    "tanh_exp": tanh_exp,
    "gcu": gcu
}
for dataset in datasets:
    for activation in activation_functions:
        print(f" ♦ Training on {dataset} with activation: {activation}")

        X_train, X_test, y_train, y_test = load_dataset(dataset)
        num_classes = len(np.unique(y_train))

        model = create_cnn_model(X_train.shape[1:], activation, num_classes)
        model.summary()

        history=model.fit(X_train, y_train, epochs=50, validation_data=(X_test, y_test))

        save_model(model, dataset, activation)
        plot_training(history, dataset, activation)

        test_loss, test_acc = model.evaluate(X_test, y_test)
        print(f" 📊 Test Accuracy on {dataset} with {activation}: {test_acc:.4f}")

        # Save test accuracy results
        save_test_results(dataset, activation, test_loss, test_acc)

```