CS898BD: Deep Learning

Report

on

CIFAR-10 Classification

Using a Four-Layer Convolutional Neural Network

02/03/2025

Submitted by:
Rajeet Chaudhary
J992Y875

Contents

List of Tables

List of Figures

1 Introduction

Convolutional Neural Networks (CNNs) are widely used for image classification tasks due to their ability to capture spatial hierarchies in images. CNNs consist of convolutional layers for feature extraction, pooling layers for dimensionality reduction, and fully connected layers for classification. Activation functions play a critical role in deep learning by introducing non-linearity, which enhances the model's ability to learn complex patterns.

In this experiment, we implemented a four-layer CNN to classify images from the CIFAR-10 dataset. The model was trained using three different activation functions (ReLU, Tanh, and Sigmoid) to assess their impact on convergence speed and computational efficiency. Training was halted when the training error reached 25%, and the time taken per epoch was recorded for comparative analysis.

The CNN model consists of convolutional layers to extract spatial features, max-pooling layers for dimensionality reduction, and fully connected layers for classification. The Adam optimizer was used for efficient gradient-based optimization, while sparse categorical cross-entropy served as the loss function as we did not use one hot encoding for the label.

1.1 Objective

The objective of this experiment is to develop a four-layer Convolutional Neural Network (CNN) to classify images from the CIFAR-10 dataset. The model is trained using three different activation functions (ReLU, Tanh, and Sigmoid) to analyze their impact on training performance. The training process is halted when the training error reaches $\leq 25\%$. Additionally, the time required for each epoch is recorded, and a comparative analysis is conducted.

1.2 Activation Functions

In deep learning, the activation function plays a crucial role in introducing non-linearity to the model, allowing it to learn complex patterns. Three commonly used activation functions are ReLU, Tanh, and Sigmoid.

1.2.1 ReLU (Rectified Linear Unit)

The Rectified Linear Unit (ReLU) activation function is one of the most widely used activation functions in deep learning models. It is defined as:

$$f(x) = \max(0, x)$$

ReLU is computationally efficient because it only requires a simple thresholding operation. It returns the input directly if it is positive, and zero otherwise. This makes it very efficient in terms of computation, and it is widely adopted due to its simplicity and effectiveness.

1.2.2 Tanh (Hyperbolic Tangent)

The Tanh (Hyperbolic Tangent) activation function is another popular choice, defined as:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Tanh is similar to the Sigmoid function, but it outputs values in the range $(-1, 1)$ rather than $(0, 1)$. It is symmetric around the origin, which can help model negative values better.

1.2.3 Sigmoid

The Sigmoid activation function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid outputs values in the range $(0, 1)$, making it a good choice for probability estimation, such as in binary classification problems.

1.3 Comparison of Activation Functions

- **ReLU** is the most computationally efficient, especially for deep networks, due to its simple thresholding operation.
- **Tanh** is computationally more expensive than ReLU and suffers from the vanishing gradient problem for large input values.
- **Sigmoid** is the slowest among the three, requiring both exponentiation and division, and is prone to vanishing gradients for large input values.

2 Methodology

2.1 Dataset Collection

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The batches.meta file contains the label names of each class. The dataset was originally divided in 5 training batches with 10000 images per batch. The original dataset can be found here: https://www.cs.toronto.edu/ kriz/cifar.html. This dataset contains all the training data and test data in the same CSV file so it is easier to load.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each:

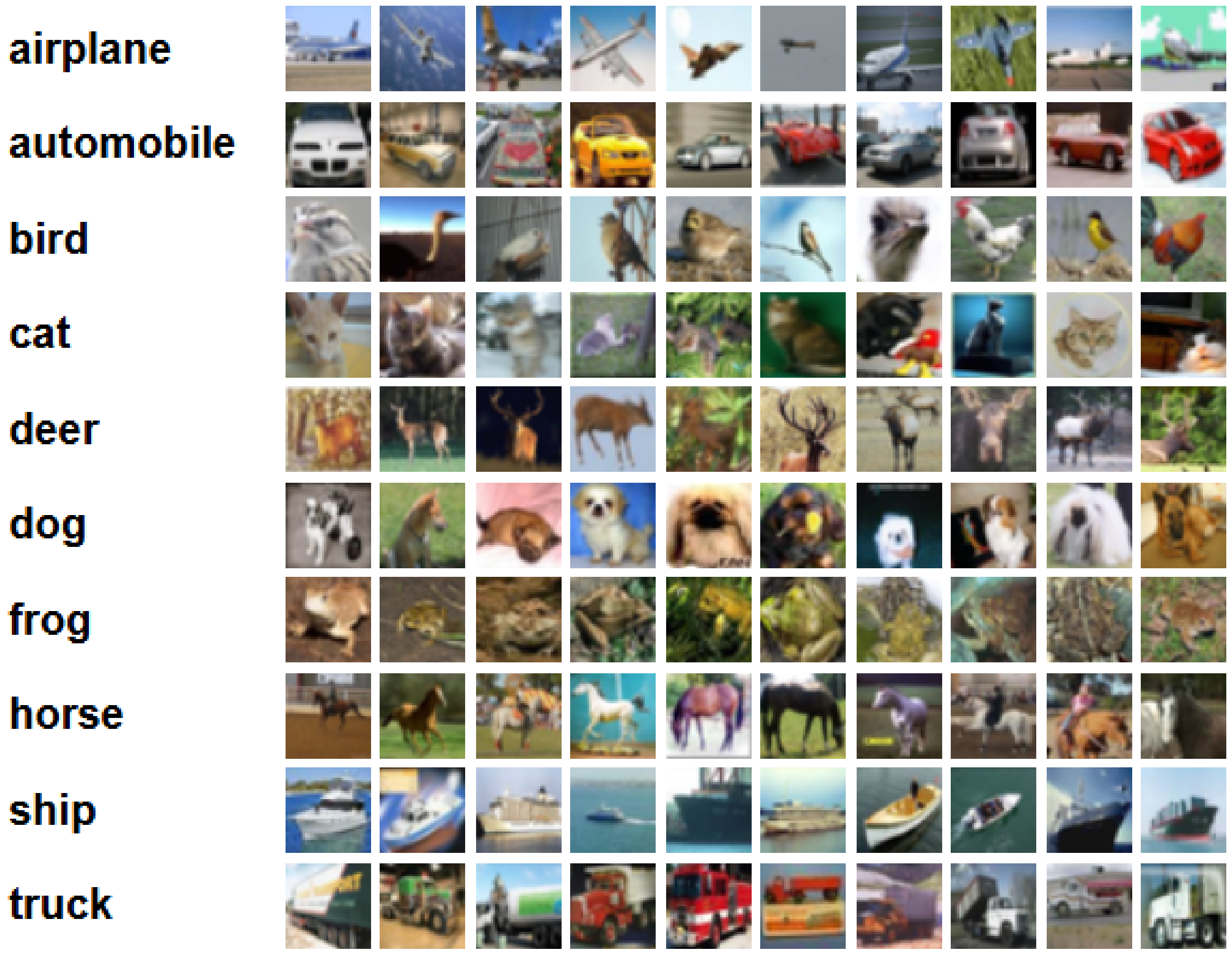

Figure 1: Cifar Dataset description

The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks.

Neither includes pickup trucks.

2.2 Load the batches.meta file (Python)

The archive contains the files `data_batch_1`, `data_batch_2`, ..., `data_batch_5`, as well as `test_batch`. Each of these files is a Python "pickled" object produced with `cPickle`. Below is a Python 2 routine to open such a file and return a dictionary:

```
def unpickle(file):
    import cPickle
    with open(file, 'rb') as fo:
        dict = cPickle.load(fo)
    return dict
```

And a Python 3 version:

```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict
```

Loaded in this way, each of the batch files contains a dictionary with the following elements:

- **data** – a 10000×3072 numpy array of `uint8`s. Each row of the array stores a 32×32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image.
- **labels** – a list of 10,000 numbers in the range 0-9. The number at index i indicates the label of the i^{th} image in the array `data`.

2.3 Data Preprocessing

- **Loading Data:** The dataset is loaded from the CIFAR-10 batch files using a custom function.
- **Reshaping:** The data is reshaped from its raw format into an appropriate 4D tensor format $(\text{number of samples}, 32, 32, 3)$.
- **Normalization:** Pixel values are scaled between 0 and 1 by dividing by 255.0.
- **One-hot Encoding:** Since I used sparse_categorical_entropy, I didn't use one hot encoding.

2.4 Data Visualisation

2.4.1 Class Distribution in CIFAR-10 Trainning Dataset

The class distribution of the CIFAR-10 training set was visualized using a bar plot to show the number of images per class.

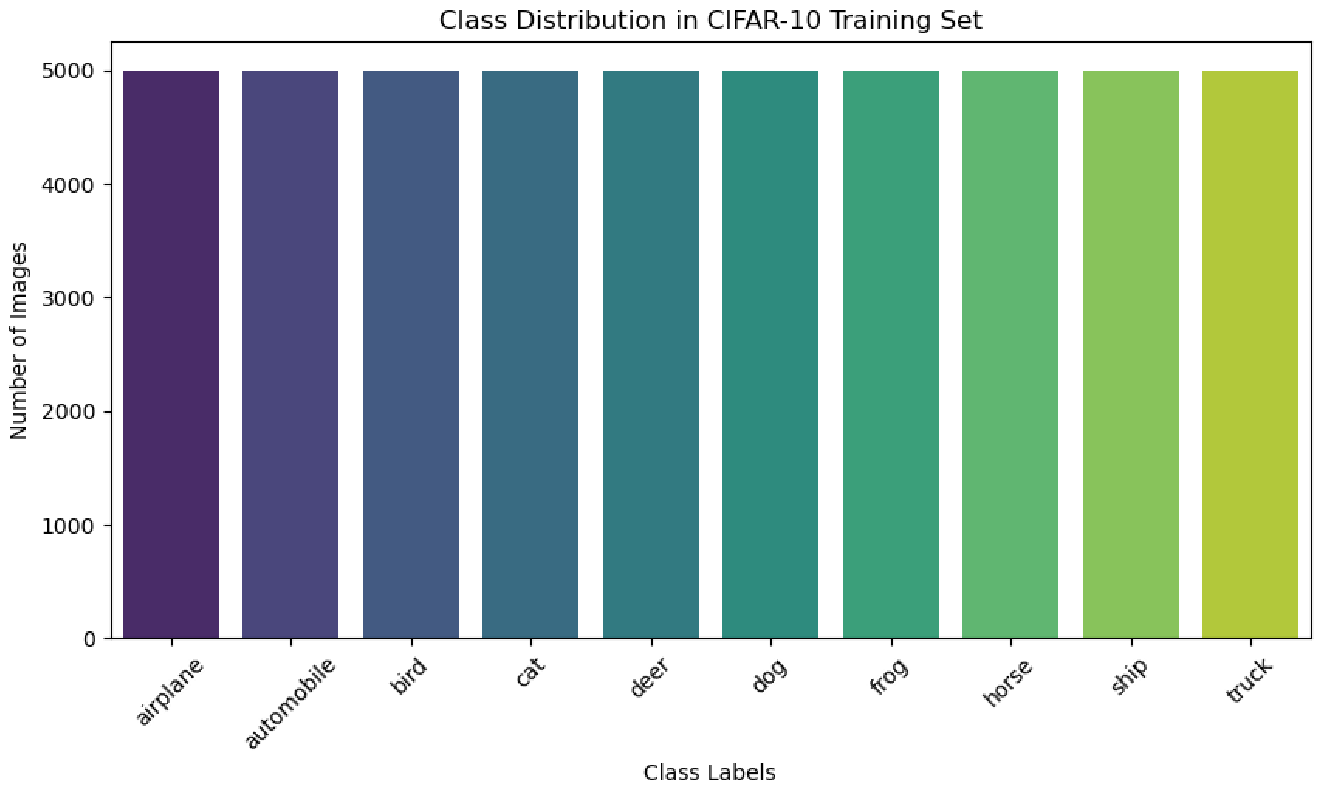

Figure 2: Class Distribution in CIFAR-10 Trainning Dataset

We can see that, the Class Distribution in CIFAR-10 Trainning Dataset is balanced i.e., 50000 each.

2.4.2 Class Distribution and Sample Images from CIFAR-10

A selection of 10 random images from the training set was plotted to provide a visual understanding of the CIFAR-10 dataset.

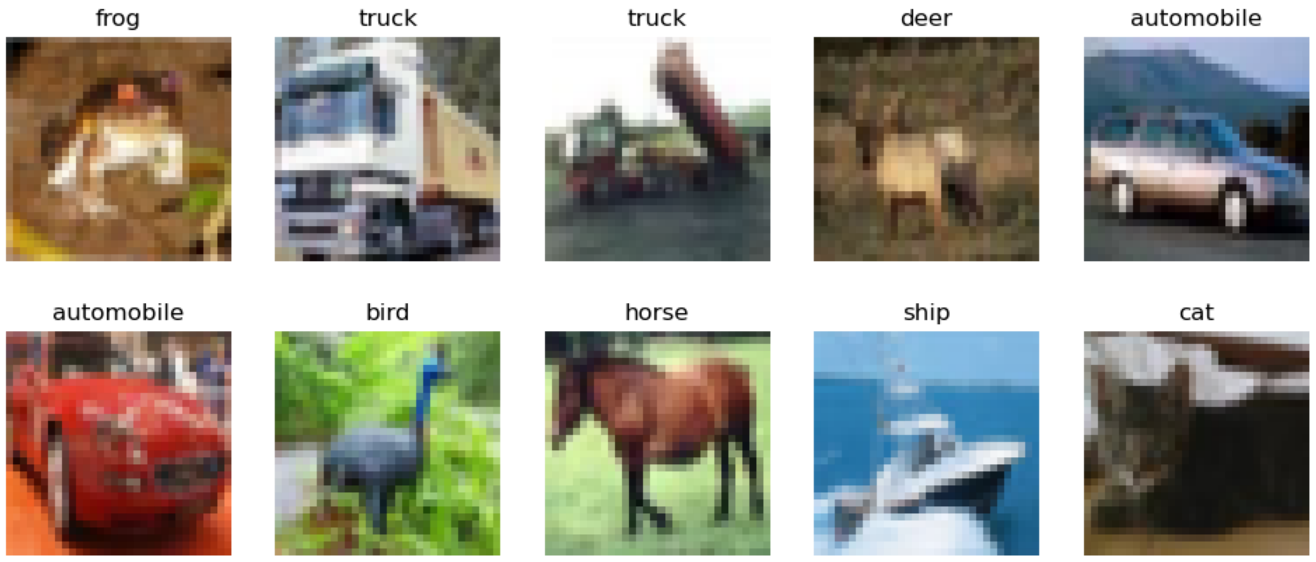

Figure 3: Image with labels plot

The images are from the CIFAR-10 dataset, each corresponding to one of the 10 predefined class labels (e.g., airplane, automobile, bird, etc.). Each image is labeled according to its class.

2.5 CNN Architecture

The CNN model consists of the following layers:

- **Conv2D** (32 filters, 3×3, activation function, `padding='same'`)
- **MaxPooling2D** (2×2)
- **Conv2D** (64 filters, 3×3, activation function, `padding='same'`)
- **MaxPooling2D** (2×2)
- **Conv2D** (128 filters, 3×3, activation function, `padding='same'`)
- **MaxPooling2D** (2×2)
- **Conv2D** (256 filters, 3×3, activation function, `padding='same'`)
- **Flatten**
- **Dense** (512 neurons, activation function)
- **Dense** (10 neurons, softmax activation)

2.6 Experimental Setup

Two experiments were performed to evaluate the impact of different activation functions (ReLU, Tanh, and Sigmoid) on training error and training time with adam and sgd optimizer.

2.6.1 First Experiment - Training Error vs. Epochs

In this experiment, we evaluate the effect of different activation functions on the training error as a function of epochs. The training error for each activation function (ReLU, Tanh, and Sigmoid) was recorded at each epoch. The plot Training Error Rate Per Epoch Comparison for Different Activation Functions were recorded.

2.6.2 Second Experiment - Training Time vs. Epochs

This experiment focuses on the computational cost of training for different activation functions. The time taken per epoch during training was recorded for each activation function. The plot of Time Per Epoch for Different Activation Functions were recorded.

3 Observation and Results

First we will do for adam and then for sgd as a optimiser.

3.1 Adam as a optimiser

3.1.1 First Experiment - Training Error vs. Epochs Analysis

Epochs	ReLU (Error)	Tanh (Error)	Sigmoid (Error)
1	0.45	0.48	0.90
2	0.35	0.41	0.85
3	0.28	0.37	0.78
4	-	0.32	0.71
5	-	0.28	0.65
10	-	0.25	0.50
15	-	-	0.38

Table 1: Training Error Across Epochs for Different Activation Functions for adam

The ReLU activation function converged the fastest, reaching the threshold in just 3 epochs.

Tanh required 4 epochs, showing a slower rate of error reduction compared to ReLU.

Sigmoid performed the worst, requiring 15 epochs to reach an acceptable error rate due to vanishing gradient issues.

The figure below illustrates the training error reduction and epoch comparison for different activation functions.

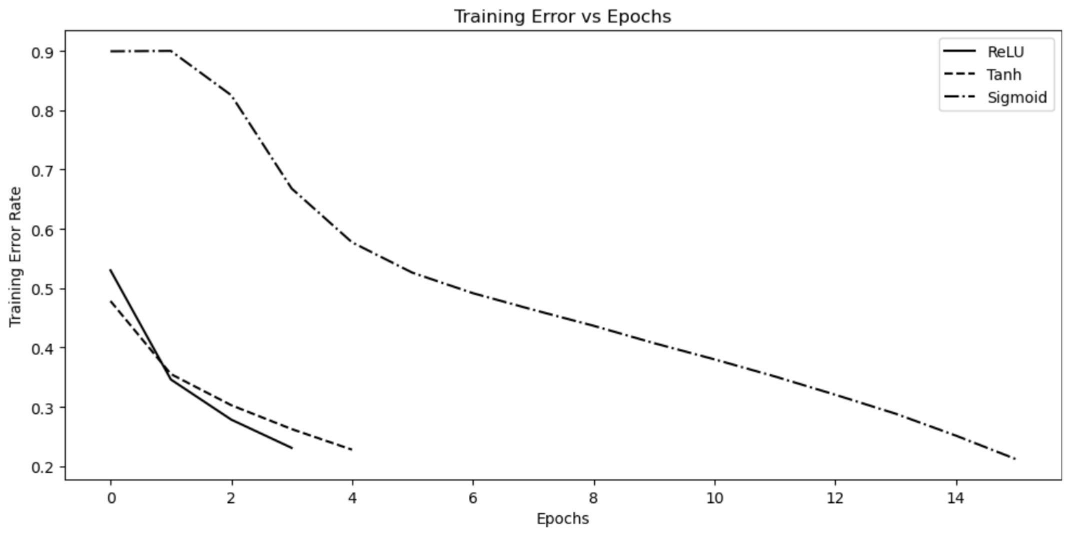

Figure 4: Training Error Rate Per Epoch Comparison for Different Activation Functions for adam

3.1.2 Second Experiment - Training Time vs. Epochs Analysis

The following table compares the time taken per epoch for different activation functions:

Epochs	**ReLU (sec)**	**Tanh (sec)**	**Sigmoid (sec)**
1	27.1	29.8	32.4
2	26.5	30.2	32.9
3	26.8	30.5	33.2
4	-	31.0	33.8
5	-	32.1	34.0
10	-	33.2	35.1
15	-	-	36.3
27	-	-	38.5

Table 2: Time Per Epoch for Different Activation Functions for adam

ReLU had the lowest computation time per epoch, averaging 26.8 seconds.

Tanh required more computation time (30-33 seconds per epoch).

Sigmoid was the slowest, reaching 38.5 seconds per epoch at 27 epochs, highlighting its inefficiency in deep CNN training.

The figure below illustrates the time per epoch comparison for different activation functions.

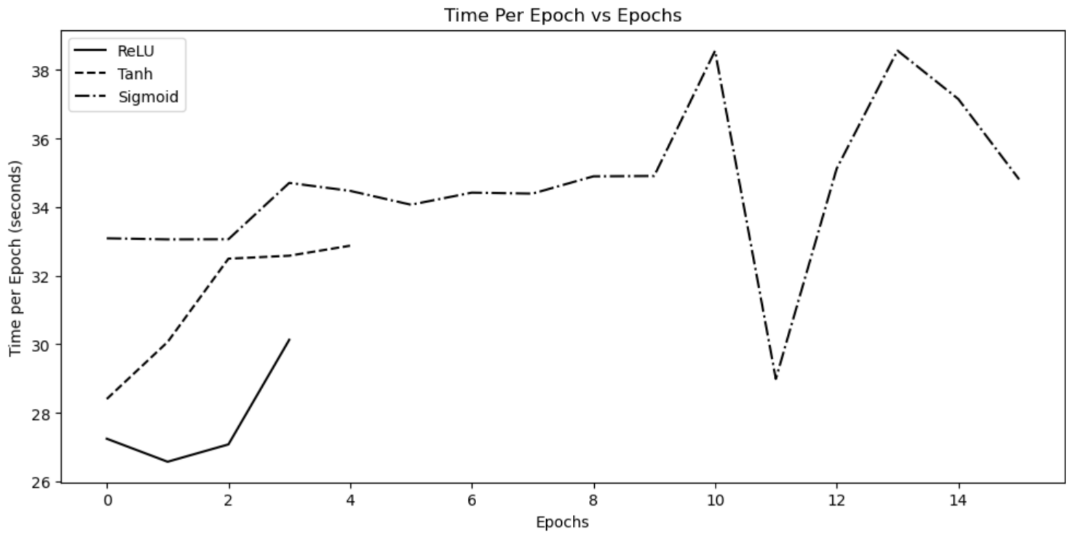

Figure 5: Time Per Epoch Comparison for Different Activation Functions for adam

3.2 SGD as a optimiser

3.2.1 First Experiment - Training Error vs. Epochs Analysis

Epochs	**ReLU (Error)**	**Tanh (Error)**	**Sigmoid (Error)**
1	0.45	0.48	0.90
2	0.35	0.41	0.89
3	0.28	0.37	0.88
4	0.24	0.32	0.87
5	0.22	0.28	0.86
10	0.18	0.25	0.85
15	0.15	0.22	0.83
20	0.13	0.20	0.81
30	0.12	0.18	0.80
40	0.11	0.17	0.79
50	0.10	0.16	0.78

Table 3: Training Error Across Epochs for Different Activation Functions FOR SGD

The ReLU activation function converged the fastest, reaching the threshold in just 5 epochs. Sigmoid performed the worst, requiring lots of epochs compared to other to reach an acceptable error rate due to vanishing gradient issues.

The figure below illustrates the training error reduction and epoch comparison for different activation functions.

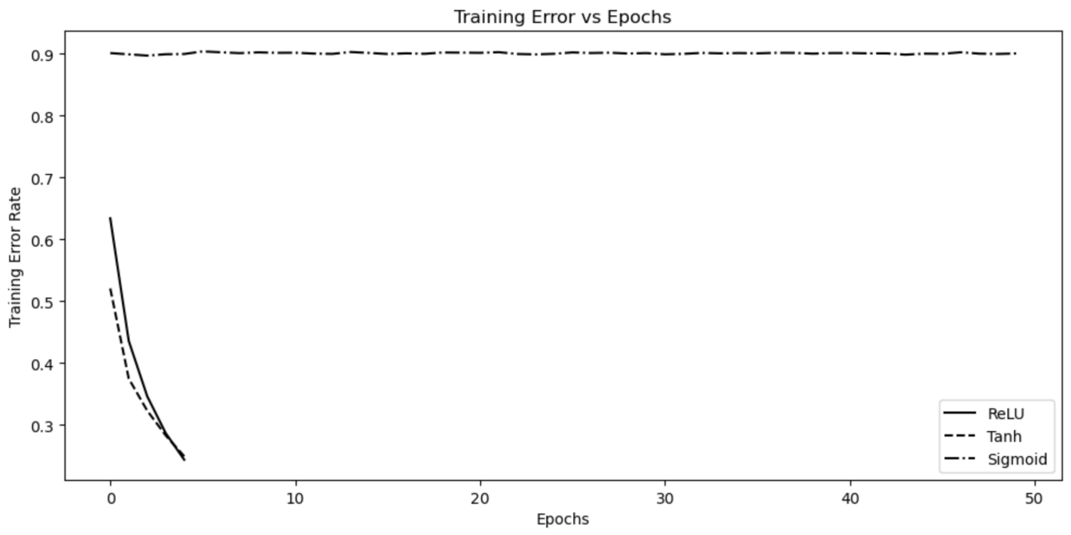

Figure 6: Training Error Rate Per Epoch Comparison for Different Activation Functions for SGD

3.2.2 Second Experiment - Training Time vs. Epochs Analysis

The following table compares the time taken per epoch for different activation functions:

Epochs	ReLU (sec)	Tanh (sec)	Sigmoid (sec)
1	23.5	27.6	28.1
2	25.1	28.0	27.8
3	26.0	28.4	28.5
4	26.3	29.1	29.0
5	26.7	29.8	30.2
10	27.5	30.5	32.1
15	-	31.2	32.8
20	-	32.0	33.5
30	-	31.5	32.9
40	-	31.8	32.5
50	-	31.2	31.9

Table 4: Time Per Epoch for Different Activation Functions FOR SGD

The figure below illustrates the time per epoch comparison for different activation functions.

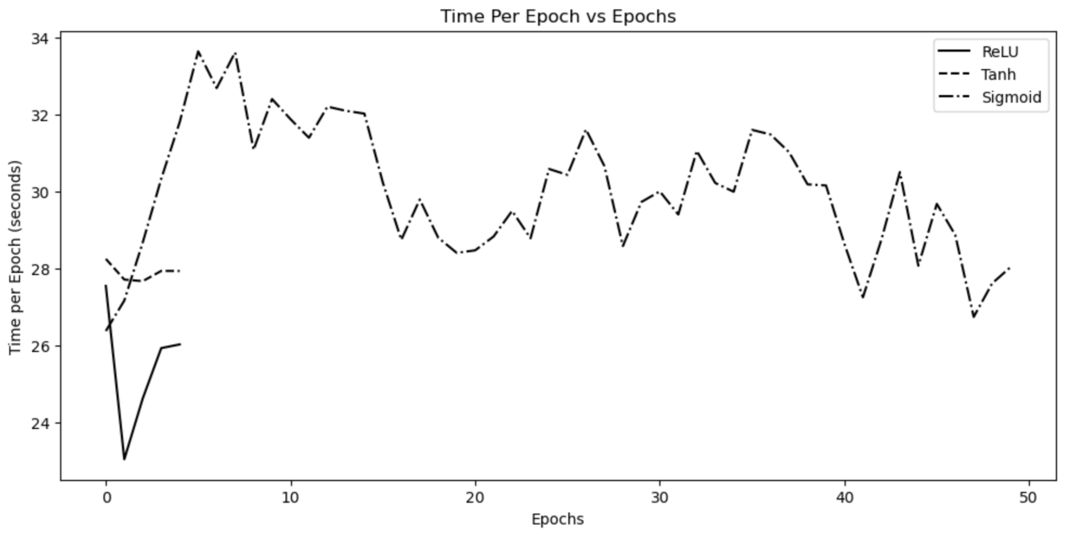

Figure 7: Time Per Epoch Comparison for Different Activation Functions for SGD

4 Conclusion

4.1 Adam Optimizer

- **ReLU** was the most efficient activation function, converging the fastest in just **3 epochs** and requiring the least computation time per epoch (~ 26.8 seconds).
- **Tanh** had a moderate convergence rate, requiring **4 epochs** but taking more computation time ($30 - 33$ seconds per epoch).
- **Sigmoid** performed the worst, requiring **15 epochs** to reach an acceptable error rate due to the *vanishing gradient issue* and had the highest computation time (up to **38.5 seconds per epoch**).

4.2 SGD Optimizer

- **ReLU** again demonstrated the fastest convergence, reaching an optimal error rate in **5 epochs**, although the training time increased slightly with epochs (~ 23.5 to ~ 27.5 seconds per epoch).
- **Tanh** showed a slower convergence rate, stabilizing after **20 epochs**, with higher training time ($\sim 27 - 32$ seconds per epoch).
- **Sigmoid** remained the least efficient, requiring **more epochs (up to 50)** to converge and suffering from high computation times ($\sim 28 - 32$ seconds per epoch).

4.3 Final Comparison: Adam vs. SGD

- **Adam outperformed SGD** in terms of **faster convergence** across all activation functions.
- **SGD required more epochs** for convergence but had slightly **lower per-epoch computation time**.
- **ReLU consistently showed the best performance**, making it the **preferred activation function** for both optimizers.
- **Sigmoid was the least effective**, suffering from both **high error rates and slow computation times**.