Lab1: Back-propagation

Yu-Hsiang, Chen - 313553004
July 10, 2024

1 Introduction

In this lab, we aim to understand and implement a simple neural network with two hidden layers, incorporating both forward pass and backpropagation algorithms. The goal is to train this neural network to perform binary classification tasks using generated datasets.
The structure of the neural network consists of:

- **Input layer**: Takes in the input features.
- **Hidden layers**: Two layers with activation functions to introduce non-linearity.
- **Output layer**: Provides the final prediction.

Each hidden layer must include at least one transformation (e.g., convolutional layer, linear layer) and one activation function (e.g., Sigmoid, Tanh, ReLU).
The training process involves:

1. **Forward Pass**: Computing the output of the network given the inputs.
2. **Backpropagation**: Calculating the gradients of the loss function with respect to the network parameters and updating the weights.

The mathematical foundation of the forward pass and backpropagation is detailed as follows:

1.1 Loss Function

The loss function used is Mean Squared Error (MSE), defined as:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

where y_i is the true label and $\hat{y}_i$ is the predicted label.

1.2 Dataset

The datasets used in this lab are:

- **Linear Dataset**: Generated using uniformly distributed random points.
- **XOR Dataset**: A classic dataset for binary classification.

The neural network's performance is evaluated by visualizing the predicted results against the ground truth and plotting the training loss over epochs. Different configurations of the neural network, such as varying learning rates, the number of hidden units, and activation functions, are tested to observe their effects on the model's performance.

2 Experiment Setups

2.1 A. Sigmoid Functions

The sigmoid function is a commonly used activation function in neural networks. It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

The sigmoid function has several desirable properties:

- It is differentiable, which is crucial for backpropagation.
- It squashes the input to the range (0, 1), introducing non-linearity into the model.
- Its derivative is simple to compute:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{3}$$

In our implementation, we utilized the sigmoid function as the activation function for the hidden layers and the output layer. This helps the neural network to learn complex patterns in the data by introducing non-linearity.

2.2 B. Neural Network

The neural network architecture we implemented consists of the following components:

- **Input Layer**: Accepts the input features.
- **Hidden Layer 1**: Applies a linear transformation followed by the sigmoid activation function.
- **Hidden Layer 2**: Applies another linear transformation followed by the sigmoid activation function.
- **Output Layer**: Produces the final output using the sigmoid activation function.

The forward pass of the network can be summarized by the following equations:

$$\mathbf{Z}_1 = \sigma(\mathbf{X}\mathbf{W}_1) \tag{4}$$

$$\mathbf{Z}_2 = \sigma(\mathbf{Z}_1\mathbf{W}_2) \tag{5}$$

$$\hat{y} = \sigma(\mathbf{Z}_2\mathbf{W}_3) \tag{6}$$

Here, $\mathbf{X}$ is the input matrix, $\mathbf{W}_1, \mathbf{W}_2$,$\mathrm{W}_3$ are the weight matrices, and $\hat{y}$ is the output of the network.

2.3 C. Backpropagation

Backpropagation is the process of calculating the gradients of the loss function with respect to the weights of the network and updating the weights to minimize the loss. The steps involved in backpropagation are as follows:

1. Compute the error at the output layer:

$$\delta_3 = (\hat{y} - y) \cdot \sigma'(\hat{y}) \tag{7}$$

2. Compute the gradient of the loss with respect to the weights of the output layer:

$$\mathbf{dW}_3 = \mathbf{Z}_2^T \delta_3 \tag{8}$$

3. Propagate the error back to the second hidden layer:

$$\delta_2 = (\delta_3 \mathbf{W}_3^T) \cdot \sigma'(\mathbf{Z}_2) \tag{9}$$

4. Compute the gradient of the loss with respect to the weights of the second hidden layer:

$$\mathbf{dW}_2 = \mathbf{Z}_1^T \delta_2 \tag{10}$$

5. Propagate the error back to the first hidden layer:

$$\delta_1 = (\delta_2 \mathbf{W}_2^T) \cdot \sigma'(\mathbf{Z}_1) \tag{11}$$

6. Compute the gradient of the loss with respect to the weights of the first hidden layer:

$$\mathbf{dW}_1 = \mathbf{X}^T \delta_1 \tag{12}$$

7. Update the weights using gradient descent:

$$\mathbf{W}_i \leftarrow \mathbf{W}_i - \eta \mathbf{dW}_i \tag{13}$$

where η is the learning rate.

```python
# Define the structure of the neural network
class NeuralNetwork:
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        # Create the weight sturcture
        self.W1 = np.random.randn(input_size, hidden_size1)
        self.W2 = np.random.randn(hidden_size1, hidden_size2)
        self.W3 = np.random.randn(hidden_size2, output_size)

    def forward(self, X):
        # Forward Process dot product
        self.Z1 = sigmoid(np.dot(X, self.W1))
        self.Z2 = sigmoid(np.dot(self.Z1, self.W2))
        self.y_hat = sigmoid(np.dot(self.Z2, self.W3))
        return self.y_hat

    def backward(self, X, y, learning_rate):
        self.error = y - self.y_hat
        # weight 3 back prop
        self.delta3 = self.error * derivative_sigmoid(self.y_hat)
        self.dW3 = np.dot(self.Z2.T, self.delta3)
        # weight 2 back prop
        self.delta2 = np.dot(self.delta3, self.W3.T) * derivative_sigmoid(self.Z2)
        self.dW2 = np.dot(self.Z1.T, self.delta2)
        # weight 1 back prop
        self.delta1 = np.dot(self.delta2, self.W2.T) * derivative_sigmoid(self.Z1)
        self.dW1 = np.dot(X.T, self.delta1)
        # update weight
        self.W3 += learning_rate * self.dW3
        self.W2 += learning_rate * self.dW2
        self.W1 += learning_rate * self.dW1
```

Figure 1: Neural Network and BackPropagation

By iteratively performing these steps, the network learns to minimize the loss function and improve its predictions.

3 Results of your testing

3.1 A. Screenshot and comparison figure

To evaluate the performance of our neural network, we trained it on two datasets: a linear dataset and the XOR dataset. Below are the screenshots showing the ground truth vs predicted results for both datasets.

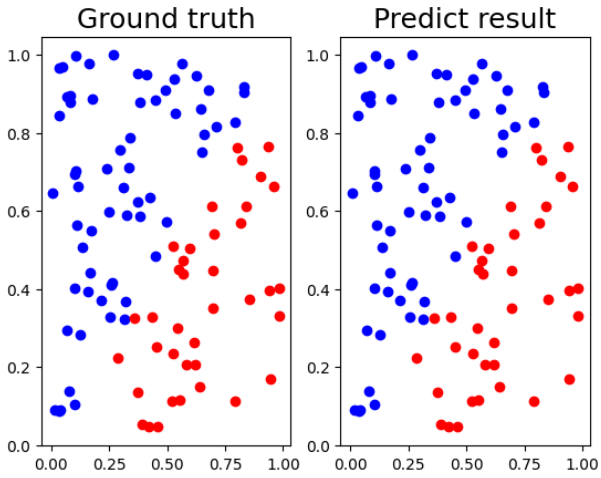

Figure 2: Linear Dataset: (Left) Ground Truth, (Right) Predicted Results

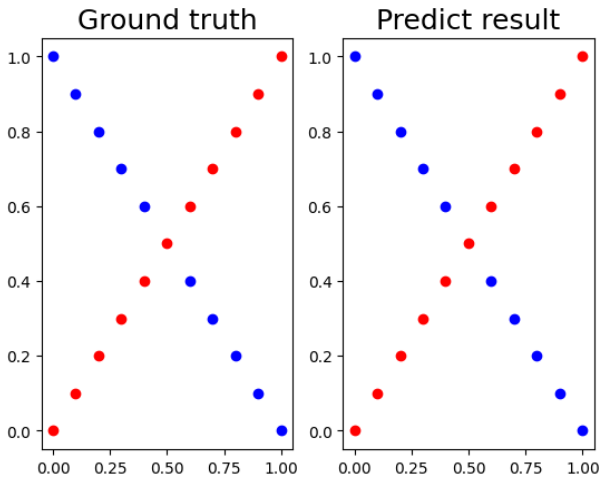

Figure 3: XOR Dataset: (Left) Ground Truth, (Right) Predicted Results

3.2 B. Show the accuracy of your prediction

The neural network was able to achieve 100% accuracy on both the linear and XOR datasets after 100,000 epochs of training (learning rate = 0.1,with hidden layer size =5) . The final loss values and accuracy for both datasets are summarized in the table below.

Dataset	Final Loss	Accuracy
Linear	3.4907e-06	100%
XOR	1.9741075e-05	100%

Table 1: Final Loss and Accuracy after 100,000 Epochs

```
Epoch 2000, Loss: 0.0009336127102596124
Epoch 2100, Loss: 0.00086722873241259
Epoch 2200, Loss: 0.0008078526487478307
Epoch 2300, Loss: 0.0007545454998155104
Epoch 2400, Loss: 0.0007065177773579269
...
Iter 97 |   Ground Truth: 1.0 |    Prediction: 1.0 |
Iter 98 |   Ground Truth: 1.0 |    Prediction: 0.99999975 |
Iter 99 |   Ground Truth: 1.0 |    Prediction: 1.0 |
loss= 3.490758320426767e-06 accuracy =100.0%
```

Figure 4: Training and Testing on Linear

```
Epoch 2000, Loss: 0.009183461144920984
Epoch 2100, Loss: 0.0074074769727153185
Epoch 2200, Loss: 0.006128921163515841
Epoch 2300, Loss: 0.005179582371155046
Epoch 2400, Loss: 0.0044551227651461345
...
Iter 18 |   Ground Truth: 1.0 |    Prediction: 0.99747074 |
Iter 19 |   Ground Truth: 0.0 |    Prediction: 0.0038072 |
Iter 20 |   Ground Truth: 1.0 |    Prediction: 0.99745845 |
loss= 1.9741075071635543e-05 accuracy =100.0%
```

Figure 5: Training and Testing on XOR

3.3 C. Learning curve (loss, epoch curve)

The learning curves for both datasets show the loss over epochs during training. Both datasets exhibit rapid convergence within the first 3,000 epochs.

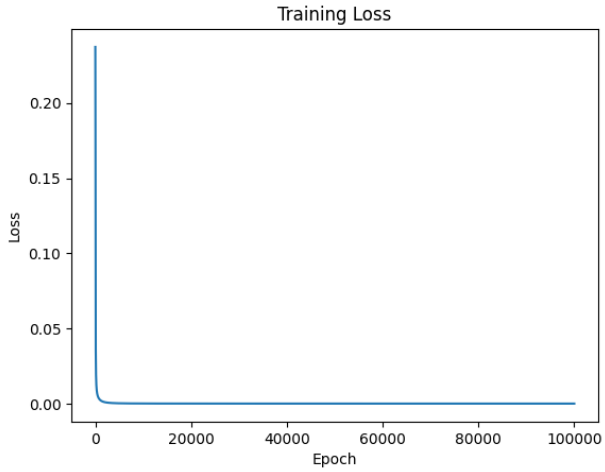

Figure 6: Learning Curve for Linear Dataset

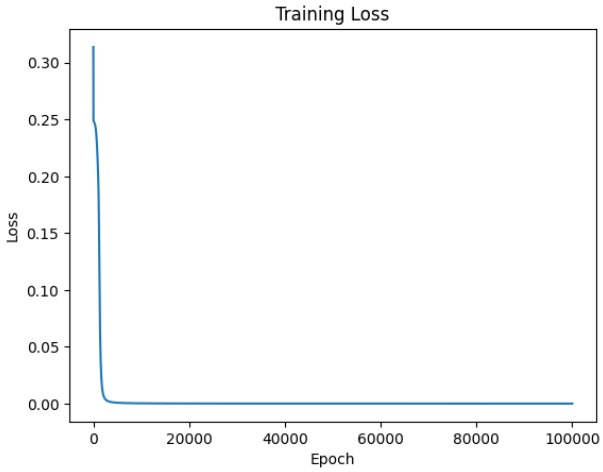

Figure 7: Learning Curve for XOR Dataset

3.4 D. Anything you want to present

During the experiment, it was observed that the choice of learning rate and the number of hidden units significantly affected the convergence rate and final accuracy. We tried various configurations and noted that while the network converged quickly within the first 3,000 epochs, further fine-tuning of hyperparameters could potentially improve the training efficiency and stability.

4 Discussion (30%)

4.1 A. Try different learning rates

We experimented with different learning rates to observe their impact on the training performance of the neural network. The results for the linear dataset are as follows:

- **Learning rate: 0.01**
  Loss: 0.004586824924178942, Accuracy: 100.0%
- **Learning rate: 0.05**
  Loss: 0.0006019188915593869, Accuracy: 100.0%
- **Learning rate: 0.1**
  Loss: 0.0001733321686487683, Accuracy: 100.0%
- **Learning rate: 0.5**
  Loss: 1.2816842010983653e-05, Accuracy: 100.0%

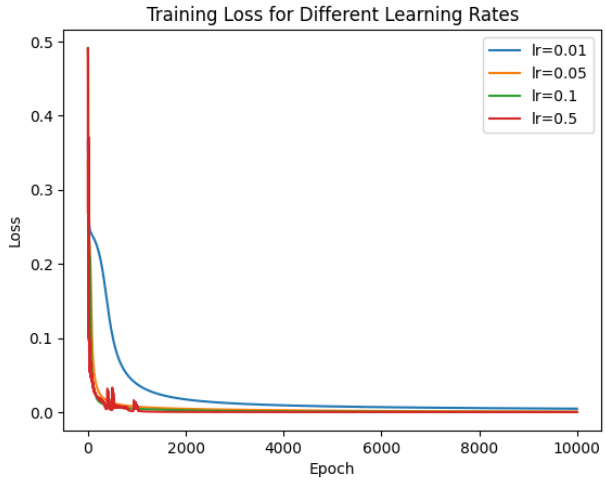

Figure 8: Training Loss for Different Learning Rates on Linear Dataset

From the results, we observe that a learning rate of 0.5 is the most effective for the linear dataset, achieving the lowest loss value.
For the XOR dataset, the results are:

- **Learning rate: 0.01**
  Loss: 0.10505482146976779, Accuracy: 90.47619047619048%
- **Learning rate: 0.05**
  Loss: 0.0006879595818460072, Accuracy: 100.0%
- **Learning rate: 0.1**
  Loss: 0.00022480598810870633, Accuracy: 100.0%
- **Learning rate: 0.5**
  Loss: 3.6959905340741546e-05, Accuracy: 100.0%

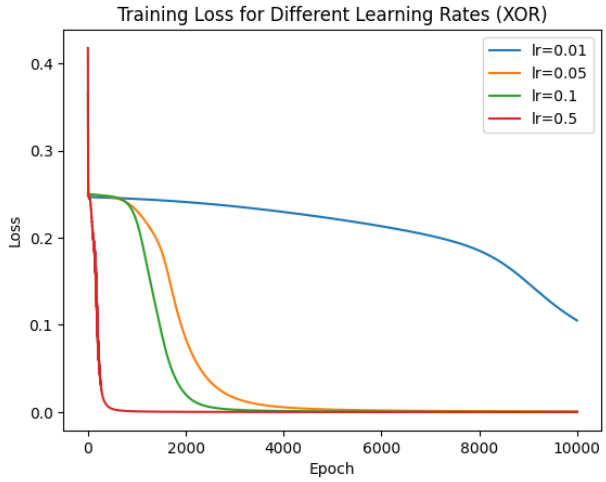

Figure 9: Training Loss for Different Learning Rates on XOR Dataset

For the XOR dataset, a learning rate of 0.5 again proves to be the most effective, achieving the lowest loss and highest accuracy.

4.2 B. Try different numbers of hidden units

Next, we evaluated the impact of varying the number of hidden units on the network's performance. The results for the linear dataset are:

- **Hidden units: 3, 3**
  Loss: 0.00017665160583931638, Accuracy: 100.0%
- **Hidden units: 5, 5**
  Loss: 0.00017113793049810752, Accuracy: 100.0%
- **Hidden units: 10, 10**
  Loss: 0.00016523825933427043, Accuracy: 100.0%
- **Hidden units: 20, 20**
  Loss: 0.00015697992778381488, Accuracy: 100.0%

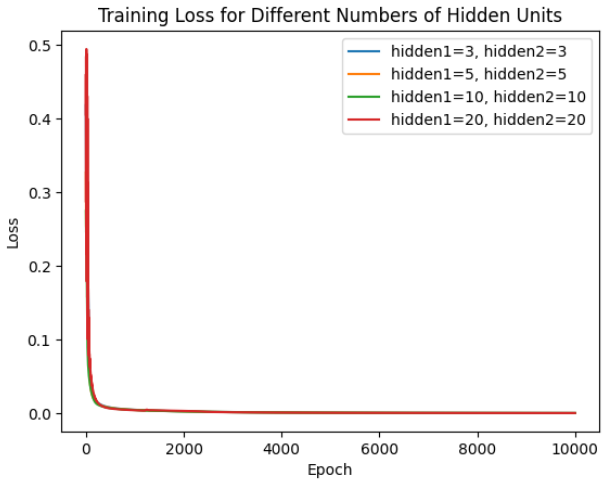

Figure 10: Training Loss for Different Numbers of Hidden Units on Linear Dataset

We observe that increasing the number of hidden units generally improves the performance, with the lowest loss achieved with 20 hidden units per layer.
For the XOR dataset, the results are:

- **Hidden units: 3, 3**
  Loss: 0.0003344252655832893, Accuracy: 100.0%
- **Hidden units: 5, 5**
  Loss: 0.0002934011844768291, Accuracy: 100.0%
- **Hidden units: 10, 10**
  Loss: 0.00025018082626211387, Accuracy: 100.0%
- **Hidden units: 20, 20**
  Loss: 0.00018257290067721544, Accuracy: 100.0%

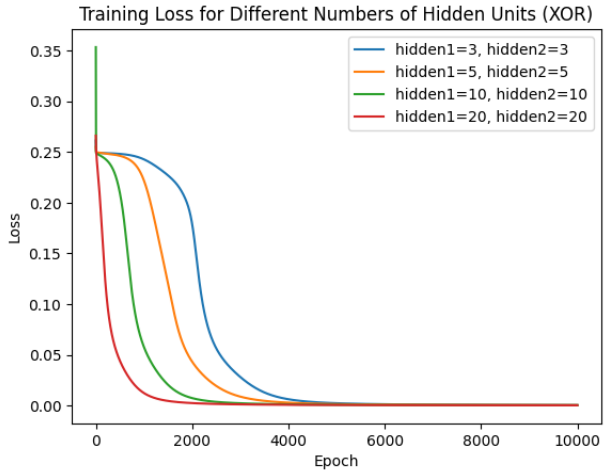

Figure 11: Training Loss for Different Numbers of Hidden Units on XOR Dataset

Similar to the linear dataset, increasing the number of hidden units improves the network's performance on the XOR dataset.

4.3 C. Try without activation functions

We also experimented with removing the activation functions from the network. The results showed a significant drop in performance:

- **Linear Dataset**
  Final Loss: 0.09542922275533261, Accuracy: 88.0%
- **XOR Dataset**
  Final Loss: 0.2887139107611549, Accuracy: 33.33333333333333%

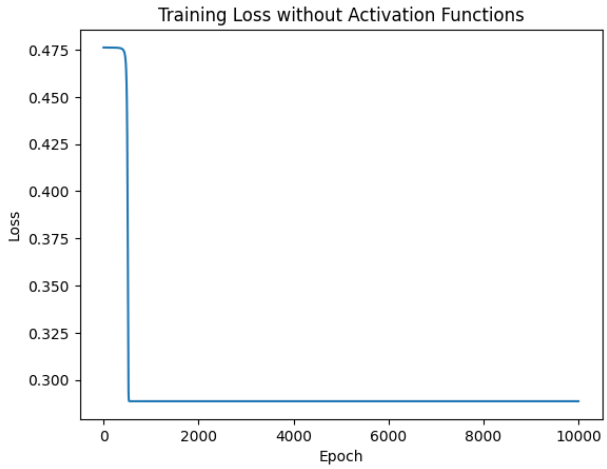

Figure 12: Training Loss without Activation Functions (We can see the loss is linear and not effective)

Without activation functions, the network's ability to learn complex patterns is severely hindered, highlighting the importance of non-linearity in neural networks.

4.4 D. Anything you want to share

During the experiments, we observed that the choice of hyperparameters such as learning rate and the number of hidden units significantly affects the convergence rate and the final accuracy of the model. Moreover, the results emphasize the importance of non-linearity introduced by activation functions, as networks without them perform poorly on both datasets.

5 Extra (10%)

5.1 A. Implement different optimizers

In addition to the basic Stochastic Gradient Descent (SGD) optimizer, we implemented the Adam optimizer. The SGD optimizer updates the weights using the following rule:

$$\mathbf{W}_i \leftarrow \mathbf{W}_i - \eta \mathbf{dW}_i \tag{14}$$

where η is the learning rate and $\mathbf{dW}_i$ is the gradient of the loss with respect to the weights.

The Adam optimizer combines the benefits of both SGD and RMSProp, incorporating momentum and adaptive learning rates. The update rule for Adam is:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{15}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{16}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{17}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{18}$$

$$\mathbf{W}_i \leftarrow \mathbf{W}_i - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{19}$$

where m_t and v_t are the first and second moment estimates, β_1 and β_2 are the exponential decay rates for these estimates, and ϵ is a small constant to prevent division by zero.

The results of evaluating different optimizers on the linear dataset are as follows:

- **Optimizer: SGD**
  Final Loss: 0.005538515725124356, Accuracy: 100.0%
- **Optimizer: Adam**
  Final Loss: 7.693262328713195e-07, Accuracy: 100.0%

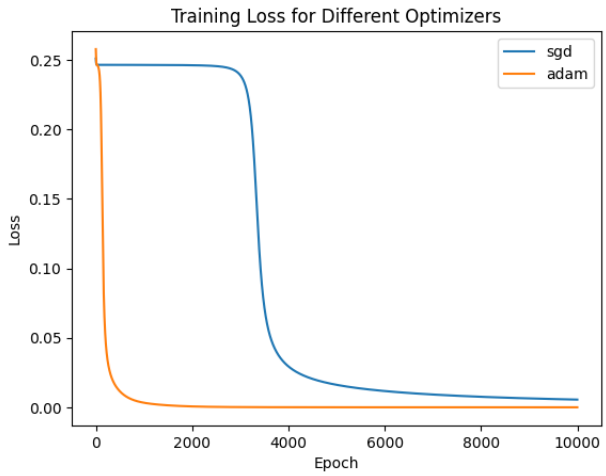

Figure 13: Training Loss for Different Optimizers on Linear Dataset

From these results, it is evident that Adam performs better than SGD on the linear dataset, achieving a significantly lower loss.

5.2 B. Implement different activation functions

We tested three different activation functions: Sigmoid, Tanh, and ReLU.
The Sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{20}$$

Its derivative is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{21}$$

The Tanh function is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{22}$$

Its derivative is:

$$\tanh'(x) = 1 - \tanh^{2}(x) \tag{23}$$

The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{24}$$

Its derivative is:

$$\text{ReLU}'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{25}$$

The results of evaluating different activation functions on the linear dataset are as follows:

- **Activation: Sigmoid**
  Final Loss: 0.0053130012637674, Accuracy: 100.0%
- **Activation: Tanh**
  Final Loss: 0.0032075458939290597, Accuracy: 100.0%
- **Activation: ReLU**
  Final Loss: 0.08799288611066064, Accuracy: 90.0%

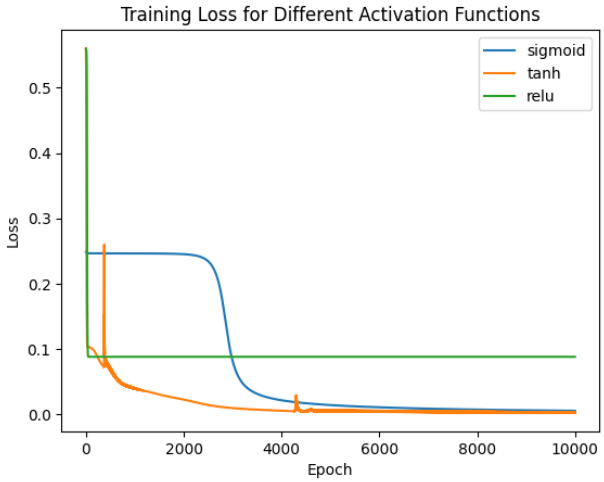

Figure 14: Training Loss for Different Activation Functions on Linear Dataset

Tanh performed the best in this case, followed by Sigmoid. ReLU had a lower accuracy and a higher final loss, indicating that it was not as effective for this dataset.

5.3 C. Implement convolutional layers

To add more complexity to our model, we implemented a convolutional layer. Convolutional layers apply a set of filters to the input data to extract features. Each filter slides over the input data and computes the dot product between the filter and a portion of the input data. The forward pass of the convolutional layer can be described by the following equation:

$$\mathbf{O}_{i,j} = \sum_{k,l} \mathbf{I}_{i+k,j+l} \mathbf{F}_{k,l} \tag{26}$$

where $\mathbf{O}$ is the output matrix, $\mathbf{I}$ is the input matrix, and $\mathbf{F}$ is the filter matrix.
The backward pass involves computing the gradients of the loss with respect to the filters and updating the filters accordingly.
The results of evaluating different activation functions with convolutional layers are as follows:

- **Activation: Sigmoid**
  Accuracy: 92.0%
- **Activation: Tanh**
  Accuracy: 85.0%
- **Activation: ReLU**
  Accuracy: 92.0%

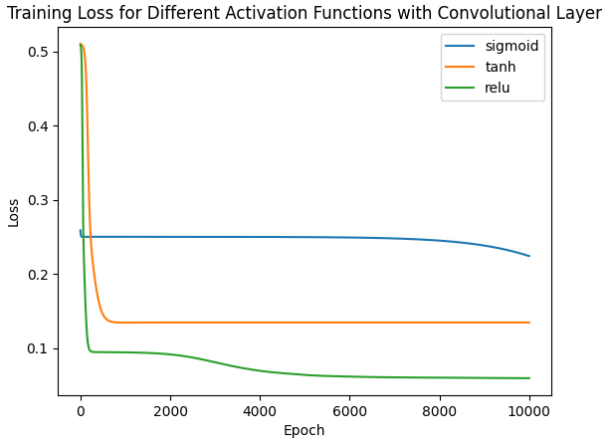

Figure 15: Training Loss for Different Activation Functions with Convolutional Layer

The results indicate that the convolutional layer with the Sigmoid and ReLU activations performed better than with Tanh. However, the overall performance of the convolutional layer was not as good as the fully connected neural network for the linear dataset.

5.4 Overall Findings

In summary, the Adam optimizer outperformed SGD, achieving a lower loss and higher accuracy on the linear dataset. Among the activation functions, Tanh provided the best performance, followed by Sigmoid, while ReLU showed suboptimal results. The convolutional layer, while useful for more complex patterns, did not perform as well as the fully connected network for the linear dataset. These findings highlight the importance of choosing appropriate optimizers and activation functions based on the specific characteristics of the dataset and the problem at hand.