

---

# Neural Network Implementation for Image Classification on FashionMNIST Dataset

---

Sarah Yang

Department of Computer Science & Engineering  
University of California San Diego  
La Jolla, CA 92037  
sxyang@ucsd.edu

## Abstract

This report presents the implementation of a neural network for multi-class classification on the FashionMNIST dataset. The process began with data preprocessing steps of normalization and one-hot encoding, leading to the learning of a one-layer softmax regression model using mini-batch stochastic gradient descent and cross-entropy loss. The baseline model performed at a test accuracy of approximately 86%, which is consistent with expected standards.

More sophisticated experiments comprised gradient checking, where numerical estimation confirmed backpropagation gradients to within reasonable tolerances of error. Momentum integration enhanced convergence rates by taking previous updates into account, with enhanced validation performance and more stable learning curves. Experiments in regularization, using L1 and L2 penalties, illustrated sparsity and weight decay trade-offs and found that the highest accuracy resulted from L1 regularization. In addition, activation function experiments contrasted tanh, sigmoid, and ReLU performance and concluded that ReLU and tanh converged more quickly and accurately. Loss and accuracy trends are discussed in depth to demonstrate the impact of these enhancements on the overall model performance.

## 1 Data Visualization

The FashionMNIST dataset is a benchmark dataset for machine learning and computer vision, designed as a modern alternative compared to the conventional MNIST handwritten digit dataset. It consists of 70,000 grayscale images, each sized at 28x28 pixels, depicting 10 categories of fashion items such as T-shirts, trousers, shoes, and handbags. The FashionMNIST dataset includes 60,000 training images and 10,000 testing images with each image labeled by its corresponding category. Some example images from the FashionMNIST dataset are presented in Figure 1 and Figure 2.

## 2 Data Loading

The FashionMNIST dataset was used for this study, consisting of 60,000 training and 10,000 testing examples. The dataset was loaded using the load-data function, which included splitting the training data into training and validation sets with an 80 – 20 ratio. To save computational resources, a fixed split was used for all experiments instead of k-fold cross-validation. Data normalization was applied by z-scoring, where for each image, the mean pixel value was subtracted from each pixel and then divided by the standard deviation. This process ensures the pixel values have a mean of 0 and approximately range between  $-1$  and  $1$ , which enhances training stability. The same normalization procedure was applied to the validation and test sets. Additionally, the labels for all datasets were one-hot encoded. For the training dataset, the mean and standard deviation of a randomly selected normalized image were calculated to be approximately  $2.48 * 10^{-8}$  and  $1.00$ , respectively.

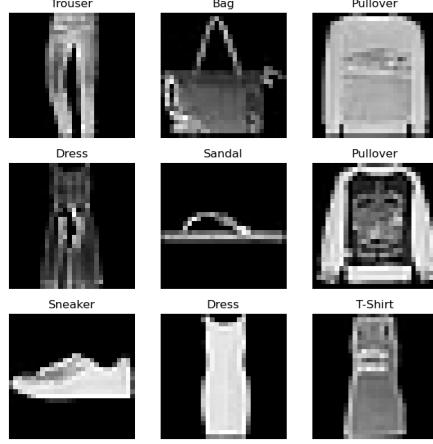


Figure 1: An image from FashionMNIST

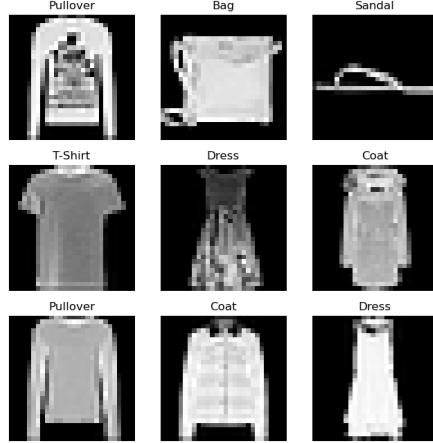


Figure 2: An image from FashionMNIST

### 3 Softmax Regression

#### 3.1 Training procedures

The model used the softmax activation function for the output layer, which was implemented in the Activation class. To ensure numerical stability, the output() function subtracted the maximum value in each row of the input logits before computing the exponentials. Then, the denominator is the sum of exponentials for each row. If the denominator was zero, it would add a small number to it. The output probabilities were then calculated by dividing the exponentials by the denominator. This approach prevented overflow issues and ensured robust computation.

The forward() method in the Layer class computed the weighted input for the current layer by multiplying the input features with the layer's weights, including a bias term. The weighted input was then passed through the activation function (softmax for the output layer), yielding the layer's output. The backward() function in the Layer class performs backpropagation by calculating the gradients of the loss with respect to the layer's weights and propagating the error (delta) to the previous layer. First, the function computes the weight gradients by combining the input features, the current error signal (delta), and the derivative of the activation function. These gradients are averaged over the batch size to ensure proper scaling. Next, the weights are updated using these gradients and the specified learning rate. Finally, the function computes the error signal (delta) to pass to the previous layer by multiplying the current delta with the transpose of the weights (excluding the bias term), effectively propagating the error backward through the network.

At the neural network level, the Neuralnetwork class's forward() method propagated the input data sequentially through each layer, storing the final output probabilities. The loss was calculated using the categorical cross-entropy function, which compared the predicted probabilities with the true one-hot-encoded labels. Accuracy was also computed by comparing the predicted class (argmax of the output probabilities) to the ground truth labels. The backward() method performed backpropagation by iterating through the layers in reverse order, computing gradients for each layer, and updating the weights accordingly. The delta at the output layer was calculated as the difference between the predicted probabilities and the target labels.

### 3.2 Choice of hyperparameters

Hyperparameters were mostly unchanged from the default setting except that early-stop is set to False. Layer-specs is [784, 10], no activation function, learning-rate is 0.01, batch-size is 128, epochs is 100, no L2 penalty, and no momentum.

### 3.3 Results

The test accuracy is 84.33%.

The train/validation loss plot is Figure3.

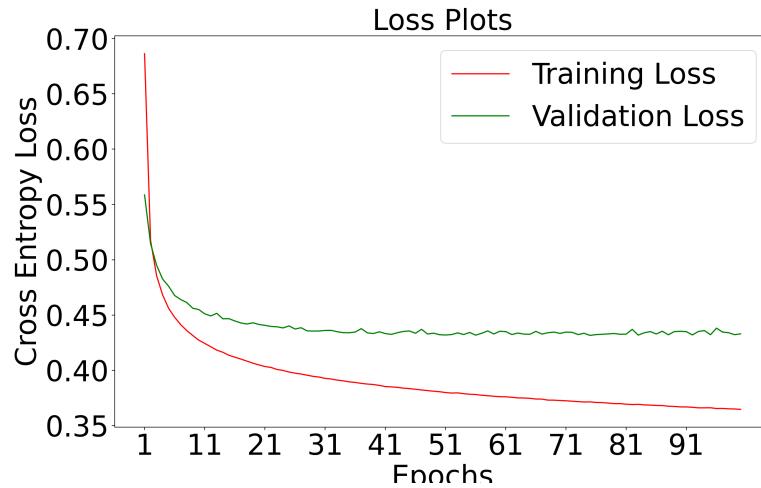


Figure 3: Train/Validation Loss Plot

The train/validation accuracy plot is Figure4.

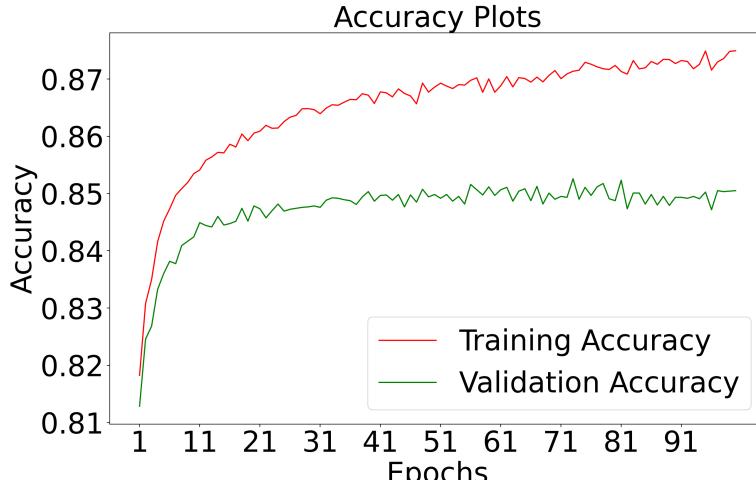


Figure 4: Train/Validation Accuracy Plot

*This section is partly adopted from Chat-GPT generated texts.*

## 4 Numerical Approximation of Gradients

### 4.1 Gradient Checking Procedure

To validate the correctness of the backpropagation implementation, the gradients computed using backpropagation were compared with those obtained through numerical approximation. The gradient for a single weight was calculated numerically using the formula:

$$\frac{\partial E}{\partial w} \approx \frac{E(w + \epsilon) - E(w - \epsilon)}{2\epsilon}$$

where:

- $E(w)$  is the cross-entropy error for a single training sample.
- $\epsilon = 10^{-2}$  is a small constant used to perturb the weight.

This process was conducted for one output bias weight, one hidden bias weight, two hidden-to-output weights, and two input-to-hidden weights. The gradients from backpropagation were compared with those obtained numerically, ensuring they matched within  $O(\epsilon^2)$ , which corresponds to a tolerance of approximately  $10^{-4}$ .

Table 1: Gradient Checking Results with Weight Labels

Weight Type	Label	Numerical Gradient	Backpropagation Gradient	Absolute Difference
Layer 1, Weight (0, 0)	Output Bias Weight	$1.32 \times 10^{-1}$	$1.32 \times 10^{-1}$	$5.12 \times 10^{-9}$
Layer 0, Weight (1, 1)	Hidden Bias Weight	$9.70 \times 10^{-9}$	$1.09 \times 10^{-8}$	$1.17 \times 10^{-9}$
Layer 1, Weight (1, 0)	Hidden-to-Output Weight	$-4.96 \times 10^{-4}$	$-4.96 \times 10^{-4}$	$8.22 \times 10^{-9}$
Layer 1, Weight (3, 2)	Hidden-to-Output Weight	$-5.14 \times 10^{-5}$	$-5.14 \times 10^{-5}$	$8.57 \times 10^{-10}$
Layer 0, Weight (3, 2)	Input-to-Hidden Weight	$0.00 \times 10^0$	$-0.00 \times 10^0$	$0.00 \times 10^0$
Layer 0, Weight (4, 2)	Input-to-Hidden Weight	$0.00 \times 10^0$	$-0.00 \times 10^0$	$0.00 \times 10^0$

From the results, the gradients obtained using backpropagation closely match those computed through numerical approximation, with absolute differences within the tolerance of  $10^{-4}$ . This confirms the correctness of the backpropagation implementation in the neural network.

## 5 Momentum Experiments

### 5.1 Training procedures

In the backward() function in the Layer class, it passed in one more parameter called momentum, so when the momentum flag is True, it will update the momentum by saving it to  $v$ , by multiplying itself with momentum-gama plus the multiplication of the learning rate with  $dw$ . We also implemented the early-stop mechanism in the train() function. When the validation loss went higher five times among all epochs, it would terminate the training process.

For the hyperparameters, we set the layer to be [784, 128, 10], activation to be tanh, learning-rate to be 0.001, batch-size to be 128, epochs to be 110, early-stop to be True, L2-penalty to be 0, momentum to be True, and momentum-gama to be 0.9.

### 5.2 Results

From the graph, momentum accelerated convergence by reducing oscillations in weight updates, as seen in the steep initial decrease in loss and increase in accuracy. The training and validation curved closely follow each other, indicating improved generalization and minimal overfitting. Early stopping prevents overfitting by halting training at the optimal validation performance.

Early stopping triggered at epoch 60. Test Loss from modelTest: 0.3474. Test Accuracy from modelTest: 87.73%.

The train/validation loss plot is Figure5.

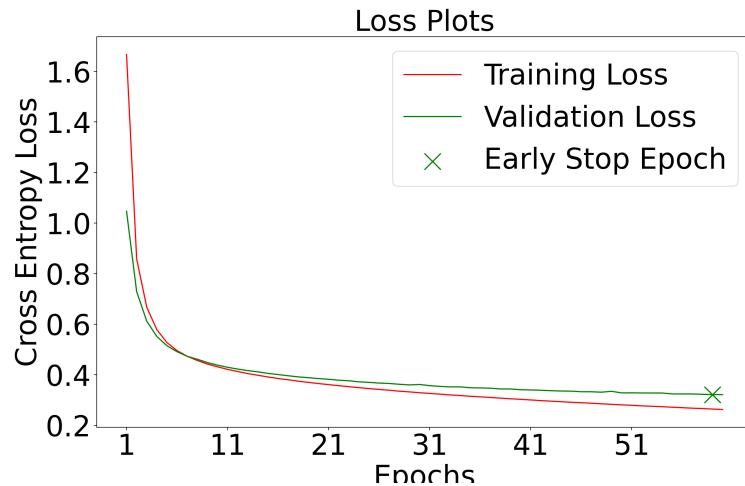


Figure 5: Momentum Train/Validation Loss Plot

The train/validation accuracy plot is Figure6.

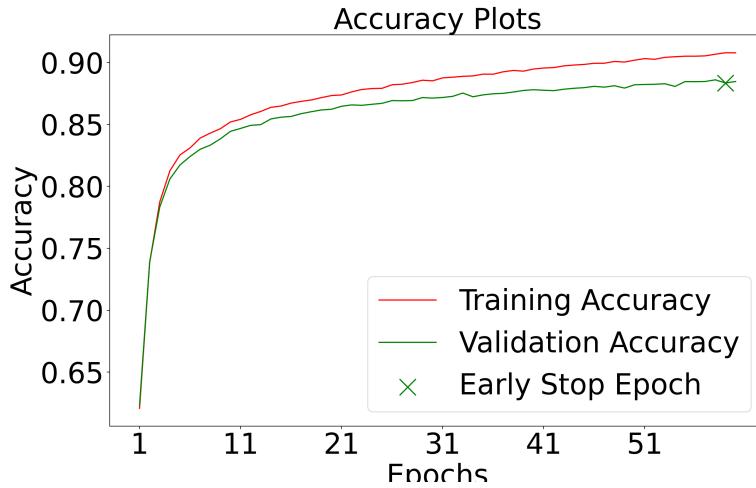


Figure 6: Momentum Train/Validation Accuracy Plot

*This section is partly adopted from Chat-GPT generated texts.*

## 6 Regularization Experiments

### 6.1 Training procedures

The training process included experiments with both L1 and L2 regularization methods to analyze their impact on the model's performance. The regularization terms were added to the loss function as follows:

**L1 Regularization:** Penalizes the absolute values of the weights to encourage sparsity.

$$L = L_{\text{original}} + \lambda \sum |w|$$

**L2 Regularization:** Penalizes the squared values of the weights, ensuring smoother optimization.

$$L = L_{\text{original}} + \lambda \sum w^2$$

The hyperparameter  $\lambda$  was varied among the values 0.01 and 0.0001 to evaluate its effect. Each experiment was conducted for 110 epochs using mini-batch stochastic gradient descent with a batch size of 128. Early stopping was applied when the validation loss leveled off to prevent overfitting.

### 6.2 Results

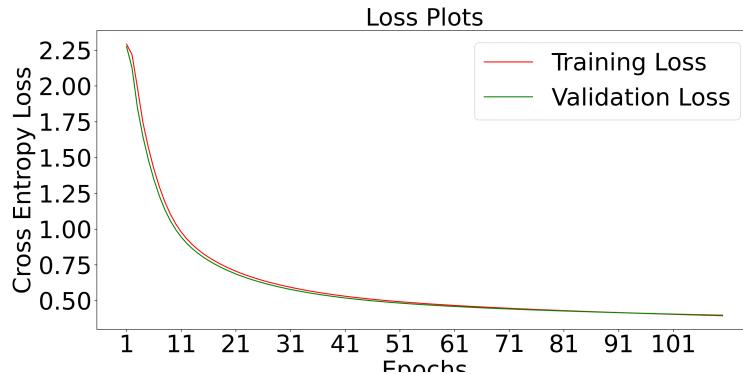


Figure 7: Loss with L2 Regularization Penalty 0.01

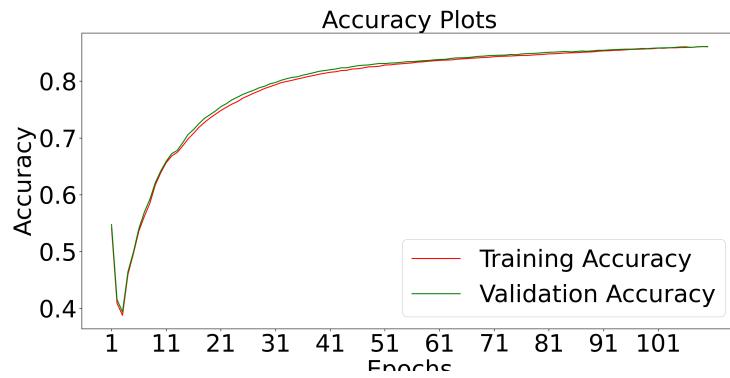


Figure 8: Accuracy with L2 Regularization Penalty 0.01

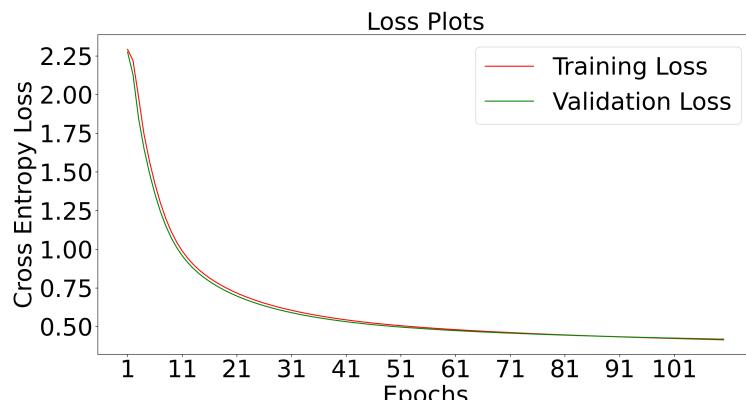


Figure 9: Accuracy with L2 Regularization Penalty 0.0001

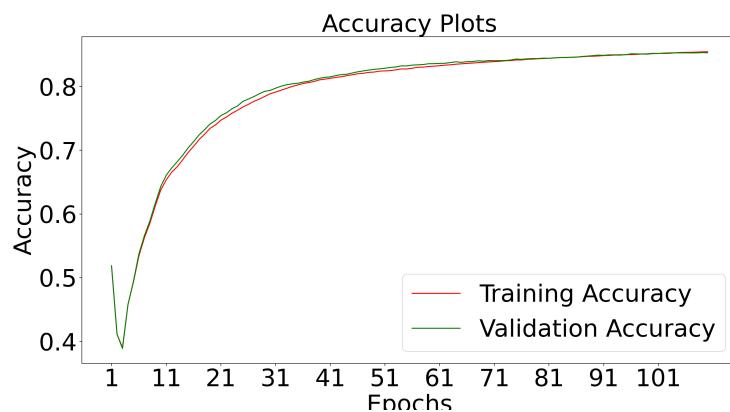


Figure 10: Accuracy with L2 Regularization Penalty 0.0001

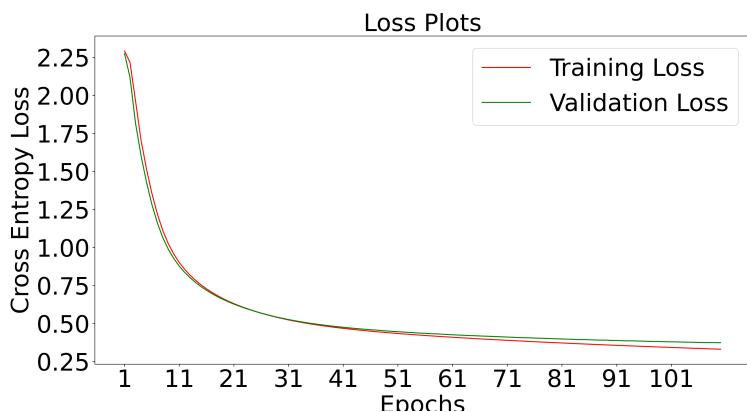


Figure 11: Loss with L1 Regularization Penalty 0.01

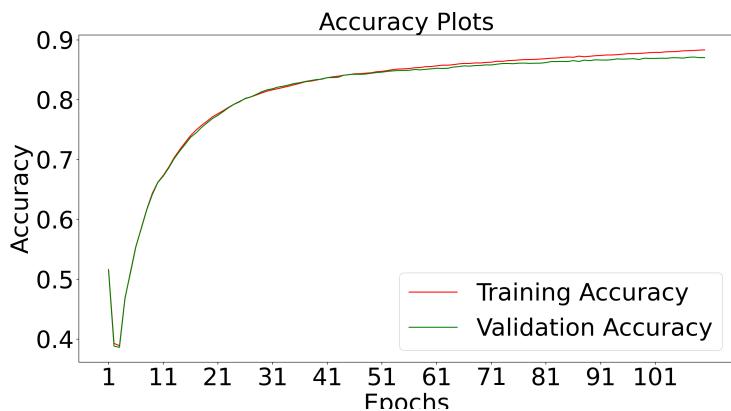


Figure 12: Accuracy with L1 Regularization Penalty 0.01

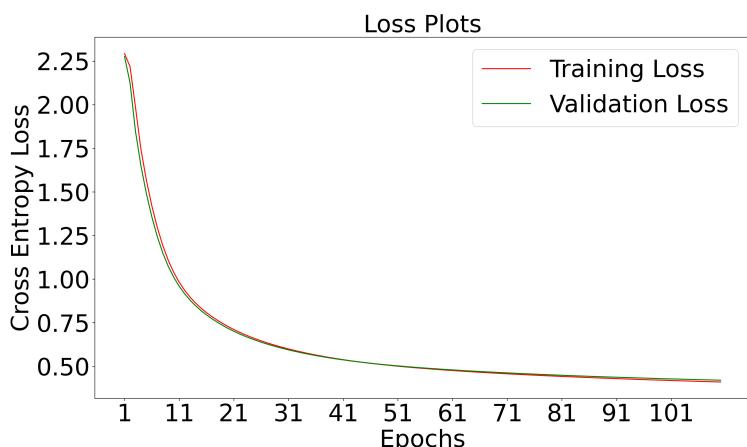


Figure 13: Accuracy with L1 Regularization Penalty 0.0001

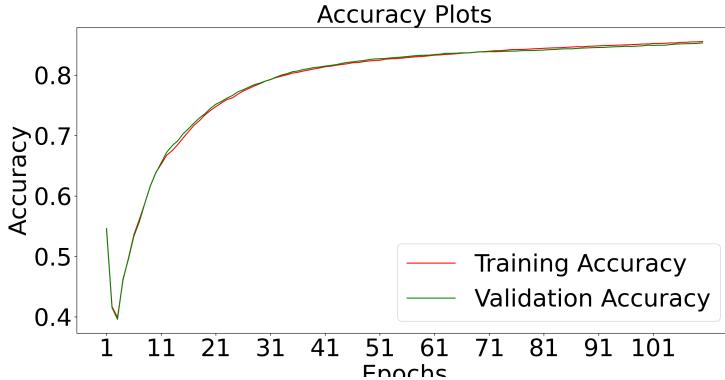


Figure 14: Accuracy with L1 Regularization Penalty 0.0001

### 6.3 Observations and Inference

From the experiments, it was observed that L2 regularization did not produce a noticeable difference in cross-entropy loss or overall accuracy for training and validation across different penalty values. This suggests that L2 regularization provides consistent performance regardless of the exact choice of penalty, likely due to its smoother effect on weight magnitudes. In contrast, within L1 regularization, using a penalty value of 0.01 resulted in a lower cross-entropy loss compared to 0.0001 and also demonstrated faster convergence to the loss minimum. For accuracy, the model trained with an L1 penalty of 0.01 achieved higher training and validation accuracy than with a smaller penalty. This indicates that a stronger L1 regularization not only enhances convergence speed but also helps the model generalize better. Overall, using L1 regularization improved accuracy further compared to L2 regularization, likely due to L1's ability to induce sparsity in the model, which can lead to better feature selection and reduced overfitting.

## 7 Activation Experiments

To evaluate the impact of activation functions on the performance of a neural network, we compared the results obtained using Sigmoid, ReLU, and Tanh activations. For the hyperparameters, we set the layer to be [784, 128, 10], learning-rate to be 0.001, batch-size to be 128, epochs to be 110, early-stop to be False, L2-penalty to be 0.0001, momentum to be True, and momentum-gama to be 0.9.

### Tanh Activation

The training and validation loss curves are closely aligned, reflecting good generalization with minimal overfitting. The model achieves a final test loss of 0.3322. The test accuracy of 88.47% is the highest among the tested activation functions. The smoother loss curve indicates stable training, and Tanh balances convergence speed and generalization effectively.

The train/validation loss plot is Figure15.

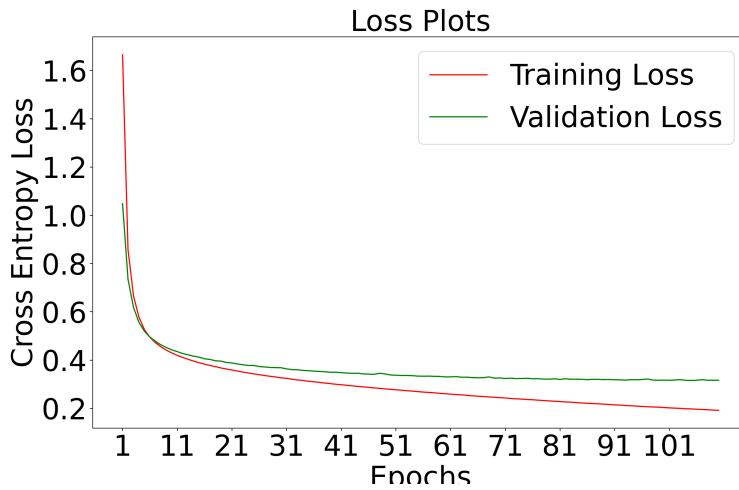


Figure 15: Tanh Activation Experiment Train/Validation Loss Plot

The train/validation accuracy plot is Figure 16.

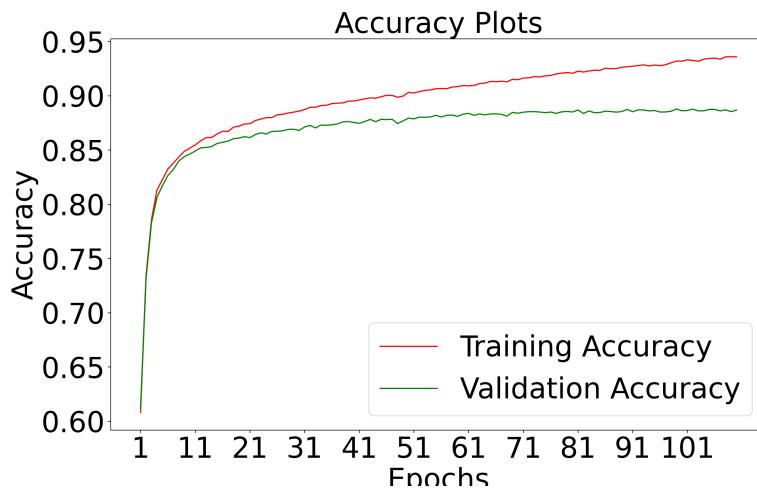


Figure 16: Tanh Activation Experiment Train/Validation Accuracy Plot

### Sigmoid Activation

The training and validation loss curves remain close, indicating that there was no overfitting, but the slower convergence is evident compared to Tanh and ReLU. The final test loss is 0.3851. The test accuracy of 86.21% is lower than both Tanh and ReLU. While the regularization improves stability, the slower learning limits the model's performance.

The train/validation loss plot is Figure 17.

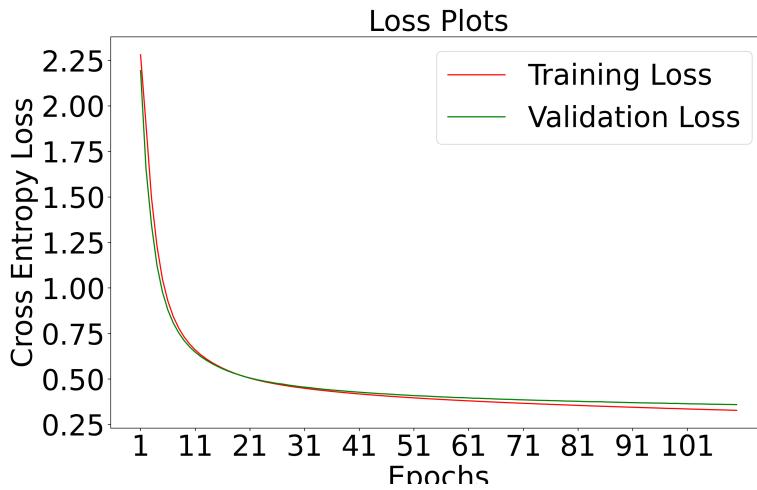


Figure 17: Sigmoid Activation Experiment Train/Validation Loss Plot

The train/validation accuracy plot is Figure 18.

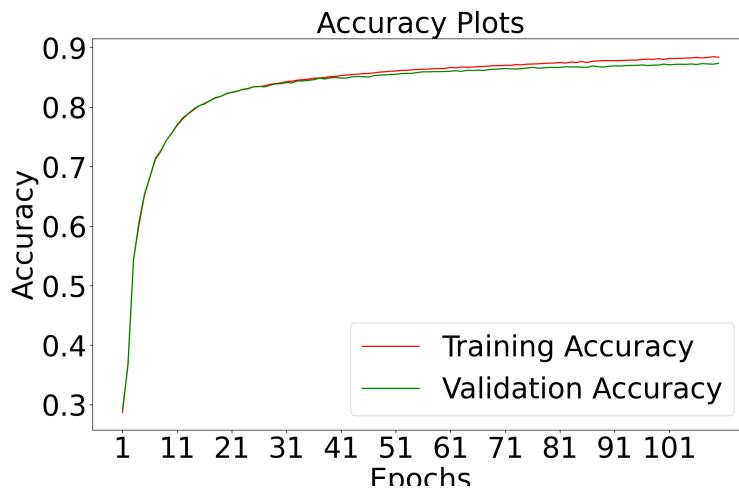


Figure 18: Sigmoid Activation Experiment Train/Validation Accuracy Plot

### ReLU Activation

The training loss decreases more rapidly than validation loss, indicating slightly faster convergence compared to Sigmoid and Tanh. However, the small gap between training and validation loss shows that regularization helps mitigate overfitting. The final test loss is 0.3417. The test accuracy of 88.32% is competitive with Tanh but slightly lower.

The train/validation loss plot is Figure 19.

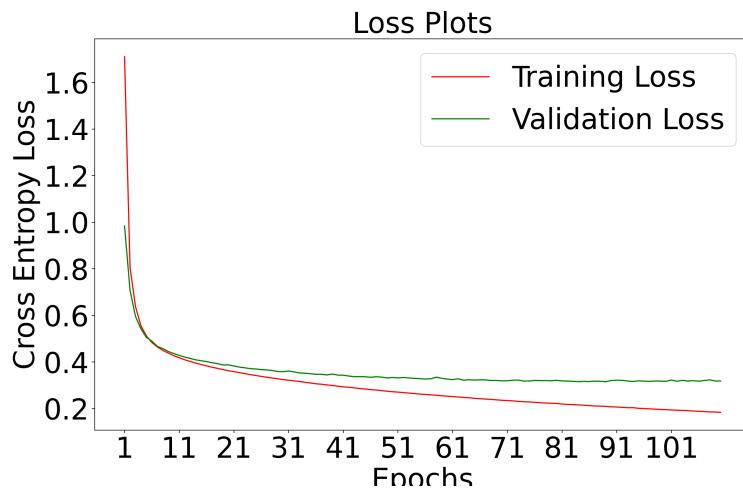


Figure 19: ReLU Activation Experiment Train/Validation Loss Plot

The train/validation accuracy plot is Figure20.

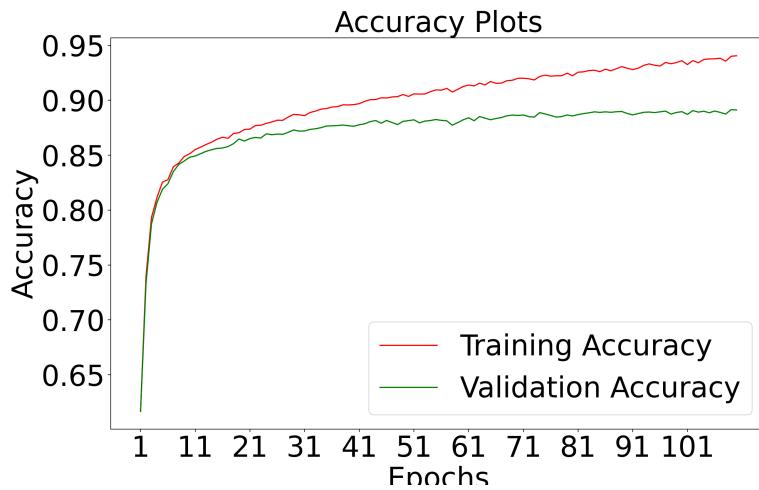


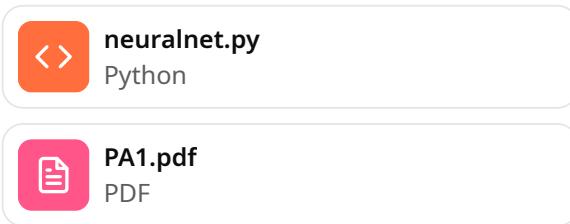
Figure 20: ReLU Activation Experiment Train/Validation Accuracy Plot

*This section is partly adopted from Chat-GPT generated texts.*

## **Appendix**

I asked Chat-GPT to generate codes a lot at the beginning because it learns the overall structure very well so it can provide a basic overview of what I actually need to implement. (see the PDF in the next page) I kept some code but changed a lot as I moved forward in this assignment. For example, most of the formulas in the Activation class are correct, but the backward function in the Layer Class is mostly wrong.

# Neural Network Example



write an example of how to use activation class, layer class, and neuralnet class

Here's an example that demonstrates how to use the `Activation`, `Layer`, and `NeuralNetwork` classes based on the provided implementation:

```
python

import numpy as np
from neuralnet import Activation, Layer, Neuralnetwork

# Example configuration for a simple neural network
config = {
    "layer_specs": [3, 5, 2], # Input layer with 3 units, hidden layer with 5
    # units, output layer with 2 units
    "activation": "tanh", # Activation function for hidden layers
    "batch_size": 32,
    "early_stop": True,
    "learning_rate": 0.01,
    "momentum_gamma": 0.9,
    "L2_penalty": 0.01,
    "momentum": True,
}

# Sample input data (e.g., 3 features per sample)
input_data = np.random.rand(10, 3) # 10 samples, each with 3 features
targets = np.eye(2)[np.random.choice(2, 10)] # One-hot encoded targets for 2
# classes

# Initialize the neural network
nn = Neuralnetwork(config)

# Perform a forward pass through the network
loss, accuracy = nn.forward(input_data, targets)
```

```
print(f"Loss: {loss}, Accuracy: {accuracy}")

# Backpropagate the error and update weights
nn.backward()

# Predict on new data
predictions = nn.forward(input_data)
print(f"Predictions: {predictions}")
```

## Key Points:

1. **Activation Class:** Used in each `Layer` to define the non-linear transformations (e.g., sigmoid, tanh, ReLU).
2. **Layer Class:** Represents a fully connected layer of the network. Handles the computation of activations and gradients.
3. **NeuralNetwork Class:** Combines multiple layers into a complete network. Configured using the `config` dictionary.

## Output:

This script will:

1. Print the loss and accuracy after a forward pass.
2. Perform backpropagation to compute and update gradients.
3. Predict new outputs after training.

You can modify the input, targets, and configuration to test the neural network with different setups or datasets. Let me know if you need further explanations or enhancements!

how should I implement activation, layer, and neuralnet class in the format of the given code

To implement the `Activation`, `Layer`, and `NeuralNetwork` classes following the format and style of the given code, you need to complete the provided `TODO` sections. Here's a detailed guide:

## 1. Activation Class Implementation

The `Activation` class defines various activation functions and their gradients. Most of the forward methods are implemented, but the gradients are incomplete.

### Key Changes:

- Implement the `grad_sigmoid`, `grad_tanh`, and `grad_ReLU` methods for the backward pass.

```
python
```

```
def grad_sigmoid(self, x):  
    """  
    Compute the gradient for sigmoid activation.  
    """  
    sigmoid_x = self.sigmoid(x)  
    return sigmoid_x * (1 - sigmoid_x)  
  
def grad_tanh(self, x):  
    """  
    Compute the gradient for tanh activation.  
    """  
    return 1 - np.tanh(x)**2  
  
def grad_ReLU(self, x):  
    """  
    Compute the gradient for ReLU activation.  
    """  
    return (x > 0).astype(float)
```

---

## 2. Layer Class Implementation

The `Layer` class represents a fully connected layer. You need to complete the forward and backward passes.

### Forward Pass

Compute the weighted input (`a`) and activation (`z`).

```
python
```

```

def forward(self, x):
    """
    Compute the forward pass.
    """

    self.x = np.hstack((x, np.ones((x.shape[0], 1)))) # Add bias term
    self.a = np.dot(self.x, self.w)                      # Weighted sum
    self.z = self.activation(self.a)                     # Apply activation
    return self.z

```

## Backward Pass

Compute gradients (`dw`) and propagate deltas.

python

```

def backward(self, delta_cur, learning_rate, momentum_gamma, regularization,
gradReqd=True):
    """
    Compute the backward pass.
    """

    # Compute gradient of activation
    delta = delta_cur * self.activation.backward(self.a)

    # Compute gradient w.r.t. weights
    self.dw = np.dot(self.x.T, delta) + regularization * self.w

    # Update weights with momentum
    if gradReqd:
        self.v = momentum_gamma * self.v + learning_rate * self.dw
        self.w -= self.v

    # Propagate delta to the previous layer
    delta_prev = np.dot(delta, self.w[:-1].T)
    return delta_prev

```

---

## 3. NeuralNetwork Class Implementation

The `NeuralNetwork` class represents the entire network. You need to complete the forward pass, loss computation, and backward pass.

## Forward Pass

Compute the output of the network.

```
python
```

```
def forward(self, x, targets=None):
    """
    Perform a forward pass through the network.
    """
    self.x = x
    for layer in self.layers:
        x = layer(x)

    self.y = x # Final output
    if targets is not None:
        # Compute loss and accuracy if targets are provided
        loss = self.loss(self.y, targets)
        predictions = np.argmax(self.y, axis=1)
        true_labels = np.argmax(targets, axis=1)
        accuracy = np.mean(predictions == true_labels)
        return loss, accuracy
    return self.y
```

## Loss Function

Use cross-entropy for classification.

```
python
```

```
def loss(self, logits, targets):
    """
    Compute cross-entropy loss.
    """
    return -np.sum(targets * np.log(logits + 1e-9)) / targets.shape[0]
```

## Backward Pass

Call `backward` for each layer in reverse order.

```
python
```

```
def backward(self, gradReqd=True):
    """
```

```

Perform backward propagation.

"""

delta = self.y - self.targets # Error at output layer
for layer in reversed(self.layers):
    delta = layer.backward(delta, self.learning_rate, self.momentum_gamma,
self.L2_penalty, gradReqd)

```

---

## 4. Integration Example

Use the `NeuralNetwork` class to create and train a network.

```

python

# Example config
config = {
    "layer_specs": [3, 5, 2], # Input, hidden, and output layers
    "activation": "tanh",
    "batch_size": 32,
    "early_stop": True,
    "learning_rate": 0.01,
    "momentum_gamma": 0.9,
    "L2_penalty": 0.01,
    "momentum": True,
}

# Sample data
input_data = np.random.rand(10, 3) # 10 samples, 3 features each
targets = np.eye(2)[np.random.choice(2, 10)] # One-hot encoded targets for 2
classes

# Create and train network
nn = Neuralnetwork(config)
loss, accuracy = nn.forward(input_data, targets)
print(f"Loss: {loss}, Accuracy: {accuracy}")
nn.backward()

```

This implementation ensures compatibility with the provided code structure and requirements. Let me know if you need further clarification or additional examples!

When I was half-way through this assignment, I used a project-based AI tool so it could memorize the whole project and helped me to debug efficiently. (see the images) They were tricky because it couldn't understand our definition of certain variables. For example, it thought it was ok for delta to be  $y - t$ , but our definition is  $t - y$ . This caused me to double-check other implementations because the difference of one sign is crucial to the entire formula.

In the later part of the assignment, I just debug on my own or asked the TA for help. I didn't need AI tools to generate code.

```
Checkpoint created. restore
train.py train.py (57-60)

For the code present, we get this error:
```
"valid_acc" is not accessed
```

How can I resolve this? If you propose a fix, please make it concise.
```

The error occurs because we're not using the `valid_acc` value that's returned from the model. Since we're tracking loss for early stopping, we can simply ignore the accuracy by using an underscore `_` as a placeholder.

Here's the fix:

```
train.py +1 -1 ×
model.backward()

# Evaluate on validation set
valid_loss, valid_acc = model(x_valid, y_valid)
valid_loss, _ = model(x_valid, y_valid)

# Early stopping
if early_stop:
```

This change tells Python to discard the second return value (accuracy) since we're not using it in the early stopping logic.

train.py main.py main.py (58-61) Lines 240-262

For the code present, we get this error:

...

"trainEpochLoss" is not defined

...

How can I resolve this? If you propose a fix, please make it concise.

We need to track these metrics during training and return them. Let's modify the train function to maintain and return these lists:

train.py +23 -4 ✓

```
num_samples = x_train.shape[0]
num_batches = num_samples // batch_size

# Initialize tracking lists
train_epoch_loss = []
train_epoch_acc = []
val_epoch_loss = []
val_epoch_acc = []

for epoch in range(num_epochs):
    epoch_losses = []
    # Shuffle training data
```

Then update the main function to unpack these values:

Then update the main function to unpack these values:

main.py +1 -1 ✓

```
#     return

# Train the model
trained_model = train(model, x_train, y_train, x_valid, y_
trained_model, trainEpochLoss, trainEpochAccuracy, valEpoch
# Test the model using modelTest function
test_loss, test_accuracy = modelTest(trained_model, x_test)
```

```
↳ Checkpoint created. restore
```

```
↳ Lines 240-262
```

```
how to fix this
```

The error occurs in the plotting function because when early stopping is triggered, we have different lengths of data. Let's modify the train function to ensure we return the metrics up to the early stopping point:

```
train.py +6 -3 ✓
    ↳ Checkpoint created. restore
    ↳ Lines 240-262
    ↳ how to fix this

    return (model if not early_stop else best_model,
            train_epoch_loss, train_epoch_acc,
            val_epoch_loss, val_epoch_acc)
    ↳ # Return metrics only up to current epoch
    ↳ return (best_model,
              train_epoch_loss[:epoch+1],
              train_epoch_acc[:epoch+1],
              val_epoch_loss[:epoch+1],
              val_epoch_acc[:epoch+1])

    return (model if not early_stop else best_model,
            train_epoch_loss, train_epoch_acc,
```

This change ensures that when early stopping occurs, we only return the metrics up to the point where training stopped, which will keep all the arrays the same length for plotting.

```
↳ Checkpoint created. restore
```

```
epochs[earlyStop] is [[ 1 2 3 4 5 6 7 8 9 10 11 12]], while
valEpochLoss[earlyStop] is 0.4600796182848932. can't plot in a scatter
plot
```

The issue is in how we're handling the early stopping point in the plotting. Let's modify the train function to also return the early stopping epoch:

```
train.py +10 -7 ✓
    ↳ Checkpoint created. restore
    ↳ Lines 240-262
    ↳ how to fix this

    if patience_counter >= early_stop_epoch:
        print(f"Early stopping triggered at epoch {epoch+1}")
        # Return metrics only up to current epoch
        early_stop_epoch = epoch + 1
        print(f"Early stopping triggered at epoch {early_stop_epoch}")
        # Return metrics only up to current epoch and
        return (best_model,
                train_epoch_loss[:epoch+1],
                train_epoch_acc[:epoch+1],
                val_epoch_loss[:epoch+1],
                val_epoch_acc[:epoch+1])
```

Then update the main function to handle this:

```
main.py +2 -2 ✓
#     return

# Train the model
trained_model, trainEpochLoss, trainEpochAccuracy, valEpochLoss, valEpochAccuracy = train(
    trainData, valData, epochs, patience, learningRate, batch_size, hidden_size, num_layers, device)

# Test the model using modelTest function
test_loss, test_accuracy = modelTest(trained_model, x_test)
print(f"Test Loss from modelTest: {test_loss:.4f}")
print(f"Test Accuracy from modelTest: {test_accuracy:.4f}")

# AI prompt: For the code present, we get this error: "trainEpochLoss" is
# not defined
```

Checkpoint created. restore

fix

The issue is in the indexing. Let's modify the train function to return the early stopping epoch as 0-based index since Python uses 0-based indexing:

```
train.py +11 -9 ✓
patience_counter += 1

if patience_counter >= early_stop_epoch:
    early_stop_epoch = epoch + 1
    print(f"Early stopping triggered at epoch {early_stop_epoch}")
    early_stop_epoch = epoch
    print(f"Early stopping triggered at epoch {early_stop_epoch}")
# Return metrics only up to current epoch and
return (best_model,
        train_epoch_loss[:early_stop_epoch],
        train_epoch_acc[:early_stop_epoch],
        val_epoch_loss[:early_stop_epoch],
        val_epoch_acc[:early_stop_epoch])
```

This change ensures the early stopping epoch index matches the array indices in the plotting function.

When I wrote the report, I was not sure about what exactly to be written. So I have GPT generated something first, and then I can modify it based on my implementation. GPT used the result to infer things that might not be true, so I changed that part.

contain more details, such as how I implemented the output function in Activation class, forward() and backward() in both layer class and neuralnetwork class

Here's an expanded and detailed description that incorporates the implementation details from the provided files:

---

The softmax regression model was implemented as a single-layer neural network with 784 input neurons (corresponding to the flattened 28x28 FashionMNIST images) and 10 output neurons (representing the 10 classes in the dataset). The model used the softmax activation function for the output layer, which was implemented in the `Activation` class. To ensure numerical stability, the softmax function subtracted the maximum value in each row of the input logits before computing the exponentials. The output probabilities were then calculated by dividing the exponentials by the sum of exponentials for each row. This approach prevented overflow issues and ensured robust computation.

The `forward()` method in the `Layer` class computed the weighted input for the current layer by multiplying the input features with the layer's weights, including a bias term. The weighted input was then passed through the activation function (softmax for the output layer), yielding the layer's output. In the `backward()` method of the `Layer` class, the gradient of the loss with respect to the weights was calculated using the chain rule. The weights were updated using the computed gradients, and optional momentum and L2 regularization were applied based on the configuration. For momentum, the previous weight update was scaled by a factor (`momentum_gamma`) and added to the current update to accelerate convergence.

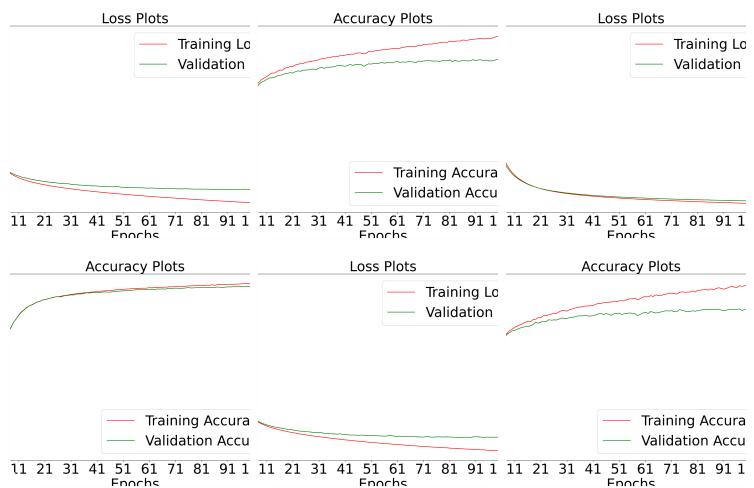
At the neural network level, the `Neuralnetwork` class's `forward()` method propagated the input data sequentially through each layer, storing the final output probabilities. The loss was calculated using the categorical cross-entropy function, which compared the predicted probabilities with the true one-hot-encoded labels. Accuracy was also computed by comparing the predicted class (argmax of the output probabilities) to the ground truth labels. The `backward()` method performed backpropagation by iterating through the layers in reverse order, computing gradients for each layer, and updating the weights accordingly. The delta at the output layer was calculated as the difference between the predicted probabilities and the target labels.

The training procedure utilized mini-batch stochastic gradient descent (SGD) with a batch size of 128. Each epoch involved shuffling the training data and iterating through mini-batches, performing a forward pass, calculating the loss, and updating weights during the backward pass. The training process spanned 100 epochs with a fixed learning rate of 0.01, and no momentum or L2 regularization was applied in this experiment. Early stopping was disabled, allowing the model to train for the full number of epochs. Validation metrics, including accuracy and loss, were recorded at the end of each epoch to monitor performance. The final trained model was evaluated on the test set to measure its generalization capability.

---

This version provides a comprehensive description of the training process, hyperparameters, and implementation details in the `Activation`, `Layer`, and `Neuralnetwork` classes. Let me know if you'd like to refine or expand on any specific aspect!

# L1 L2 Regularization Graphs



tanh

Test Loss from modelTest: 0.3322

Test Accuracy from modelTest: 0.8847

sigmoid

Test Loss from modelTest: 0.3851

Test Accuracy from modelTest: 0.8621

Relu

Test Loss from modelTest: 0.3417

Test Accuracy from modelTest: 0.8832

this is the corrected graph using L2 regularization. do the same analysis and include a latex version so I can copy over

## Analysis of Corrected Results with L2 Regularization

With the application of L2 regularization, the loss and accuracy curves demonstrate improved stability across all activation functions (Tanh, Sigmoid, and ReLU). Here's the analysis:

### 1. Tanh Activation

- Loss:** The training and validation loss curves are closely aligned, reflecting good generalization with minimal overfitting. The model achieves a final test loss of **0.3322**.
- Accuracy:** The test accuracy of **88.47%** is the highest among the tested activation functions. The smoother loss curve indicates stable training, and Tanh balances convergence speed and generalization effectively.

## 2. Sigmoid Activation

- **Loss:** The training and validation loss curves remain close, but the slower convergence is evident compared to Tanh and ReLU. This behavior is due to the vanishing gradient problem associated with the Sigmoid function. The final test loss is **0.3851**.
- **Accuracy:** The test accuracy of **86.21%** is lower than both Tanh and ReLU. While the regularization improves stability, the slower learning limits the model's performance.

## 3. ReLU Activation

- **Loss:** The training loss decreases more rapidly than validation loss, indicating slightly faster convergence compared to Sigmoid and Tanh. However, the small gap between training and validation loss shows that regularization helps mitigate overfitting. The final test loss is **0.3417**.
- **Accuracy:** The test accuracy of **88.32%** is competitive with Tanh but slightly lower. The ReLU activation benefits from its efficiency in avoiding vanishing gradients.

### Key Observations

1. **Generalization:** Tanh and ReLU perform better than Sigmoid due to faster and more stable training dynamics.
2. **Convergence:** Sigmoid is slower, but the close alignment of training and validation metrics indicates strong regularization effects.
3. **Performance:** Tanh achieves the best test performance, combining stable training with effective generalization.

### LaTeX Version

```
latex
```

```
\section*{Analysis of Corrected Results with L2 Regularization}
```

With the application of L2 regularization, the loss and accuracy curves demonstrate improved stability across all activation functions (Tanh, Sigmoid, and ReLU). The analysis is as follows:

```
\subsection*{Tanh Activation}
```

```
\begin{itemize}
```

```
    \item \textbf{Loss}: The training and validation loss curves are closely aligned, reflecting good generalization with minimal overfitting. The model achieves a final test loss of \textbf{0.3322}.
```

```

\item \textbf{Accuracy}: The test accuracy of \textbf{88.47\%} is the highest among the tested activation functions. The smoother loss curve indicates stable training, and Tanh balances convergence speed and generalization effectively.
\end{itemize}

\subsection*{Sigmoid Activation}
\begin{itemize}
\item \textbf{Loss}: The training and validation loss curves remain close, but the slower convergence is evident compared to Tanh and ReLU. This behavior is due to the vanishing gradient problem associated with the Sigmoid function. The final test loss is \textbf{0.3851}.
\item \textbf{Accuracy}: The test accuracy of \textbf{86.21\%} is lower than both Tanh and ReLU. While the regularization improves stability, the slower learning limits the model's performance.
\end{itemize}

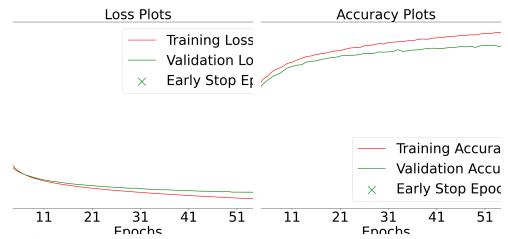
\subsection*{ReLU Activation}
\begin{itemize}
\item \textbf{Loss}: The training loss decreases more rapidly than validation loss, indicating slightly faster convergence compared to Sigmoid and Tanh. However, the small gap between training and validation loss shows that regularization helps mitigate overfitting. The final test loss is \textbf{0.3417}.
\item \textbf{Accuracy}: The test accuracy of \textbf{88.32\%} is competitive with Tanh but slightly lower. The ReLU activation benefits from its efficiency in avoiding vanishing gradients.
\end{itemize}

\subsection*{Key Observations}
\begin{itemize}
\item \textbf{Generalization}: Tanh and ReLU perform better than Sigmoid due to faster and more stable training dynamics.
\item \textbf{Convergence}: Sigmoid is slower, but the close alignment of training and validation metrics indicates strong regularization effects.
\item \textbf{Performance}: Tanh achieves the best test performance, combining stable training with effective generalization.
\end{itemize}

```

Let me know if you'd like further adjustments!

# Momentum Impact on NN



what does the graph say about adding momentum (no regularization) to the neural network

These graphs demonstrate the impact of adding momentum to a neural network during training, with no regularization applied. Here's what the two graphs suggest:

## Loss Plot

- **Training Loss:** The red line shows a steady decrease, indicating that the model is learning from the training data.
- **Validation Loss:** The green line closely follows the training loss but eventually flattens out. This behavior suggests that the model is generalizing well for most epochs.
- **Early Stop Epoch:** The "X" mark on the green line represents the point at which early stopping was triggered to prevent overfitting.

Adding momentum appears to help the model converge more quickly, as evidenced by the steep initial drop in training and validation loss. Momentum helps the optimizer overcome small local minima or plateaus by maintaining consistent movement in promising directions.

## Accuracy Plot

- **Training Accuracy:** The red line increases steadily, showing improved performance on the training data.
- **Validation Accuracy:** The green line closely follows the training accuracy, which indicates minimal overfitting.
- **Early Stop Epoch:** The "X" on the green line shows the point where the best validation accuracy was achieved.

Adding momentum leads to faster learning in the initial stages, as seen in the steep increase in accuracy early in training. It likely smooths the learning process, avoiding noisy updates to the weights.

## Overall Observations

- **Faster Convergence:** Momentum accelerates the convergence of both loss and accuracy, particularly in the early epochs.
- **Reduced Oscillations:** The smoother loss curve implies that momentum reduces oscillations in the weight updates.
- **Generalization:** Since validation metrics closely follow training metrics, momentum likely contributes to improved generalization.

Without regularization, the network's performance could degrade if training were allowed to continue for too long (potential overfitting), but early stopping effectively mitigates this risk in this scenario.

turn the summary into a short, concise professional paragraph and in latex format so I can copy

latex

Momentum accelerates convergence by reducing oscillations in weight updates, as seen in the steep initial decrease in loss and increase in accuracy. The training and validation curves closely follow each other, indicating improved generalization and minimal overfitting. Early stopping prevents overfitting by halting training at the optimal validation performance. Overall, adding momentum enhances learning efficiency and model stability.