EE5179:Deep Learning for Imaging Programming Assignment 1: MNIST Report

Saranath P
ns24z458

Contents

1 Implementation of Multi-Layered Perceptron

Before we implement the actual Feed Forward Neural Networks, there are some helper classes which is used to implement them.

1.1 Loss Functions

The class `LossFunctions` includes two main types of loss functions:

1.1.1 `SquaredErrorLoss` Class

This class is responsible for calculating the Mean Squared Error (MSE) loss with optional regularization (L1 or L2).

- **__init__**: Initializes the loss function with two parameters:
  - `alpha`: A regularization constant.
  - `regularization`: Specifies the type of regularization (either 'l1', 'l2', or none).
- **__call__**: Computes the MSE loss between the predicted values (`y_pred`) and the true values (`y_true`). It also includes a regularization term to penalize the model's weights, depending on the specified regularization type (L1 or L2).
- **gradient**: Computes the gradient of the MSE loss with respect to the predicted values and the model weights, including the gradient of the regularization term. The gradient is important for backpropagation during training to update the model's parameters.
- **__repr__**: Provides a string representation of the class with its current settings (i.e., `alpha` and the type of regularization).

1.1.2 `CrossEntropyLoss` Class

This class is designed to compute the Cross-Entropy loss, often used in classification problems.

- **__init__**: Similar to the `SquaredErrorLoss`, it initializes the class with an `alpha` value and an optional regularization parameter (`regularization`).
- **__call__**: Calculates the Cross-Entropy loss between the predicted probabilities (`y_pred`) and the true one-hot encoded labels (`y_true`). Like the Squared Error Loss, it also includes a regularization term to adjust the model weights based on the specified regularization method.
- **gradient**: Computes the gradient of the Cross-Entropy loss with respect to the predicted probabilities and the model weights. This function plays a critical role in optimizing the model during training by adjusting weights according to the loss minimization process.

- **__repr__**: Provides a string representation of the current settings of the Cross-Entropy Loss function, including `alpha` and the regularization type.

Both classes in the `LossFunctions` module allow for flexibility by including regularization terms, which help prevent overfitting during the training of neural networks. They also return gradients that are essential for updating the model's parameters using optimization algorithms like gradient descent.

1.2 Activation Functions

The class `Activations` includes several common activation functions used in neural networks:

1.2.1 `ReLU` Class

This class applies the Rectified Linear Unit (ReLU) activation function, which is a widely used non-linear activation function.

- **__call__**: Applies the ReLU function element-wise to the input array `x`. The function returns `max(0, x)` for each element in `x`, which means that negative values are replaced with 0, while positive values remain unchanged.
- **gradient**: Computes the gradient (derivative) of the ReLU function. The gradient is 1 for values of `x` greater than 0, and 0 otherwise. This is used during backpropagation to update weights.

1.2.2 `Sigmoid` Class

This class applies the Sigmoid activation function, which maps input values to a range between 0 and 1, making it useful for binary classification tasks.

- **__call__**: Applies the Sigmoid function to the input `x`, returning `1 / (1 + exp(-x))`. This function squashes input values to the range (0, 1).
- **gradient**: Computes the gradient of the Sigmoid function. The gradient is calculated as `sigmoid(x) * (1 - sigmoid(x))`, which is used for backpropagation.

1.2.3 `Softmax` Class

The Softmax function is often used in the output layer of neural networks for multiclass classification problems.

- **__call__**: Applies the Softmax function to the input array `x`. The function returns probabilities by exponentiating the input values and normalizing them, ensuring that the sum of all output values equals 1.
- **gradient**: Computes the gradient of the Softmax function, returning the Jacobian matrix. This is essential for calculating the gradients when training models for multiclass classification tasks.

1.2.4 `Tanh` Class

The Tanh (hyperbolic tangent) function maps input values to a range between -1 and 1, making it useful for zero-centered activation.

- **__call__**: Applies the Tanh function to the input array `x`, returning `tanh(x)`. This function squashes values to the range (-1, 1).
- **gradient**: Computes the gradient of the Tanh function, returning `1 - tanh(x)`^{2}. This is used during backpropagation to adjust the model weights.

1.2.5 `Linear` Class

The Linear activation function is simply the identity function, often used in the output layer for regression tasks.

- **__call__**: Returns the input `x` as-is without any modifications.
- **gradient**: The gradient of the Linear function is always 1, as the derivative of the identity function is constant. This is used during backpropagation to update the model weights.

1.3 Parameter Initialization

The class `ParameterInitializer` is used to initialize the weights and biases of a neural network. It supports three different initialization methods: `random`, `gaussian`, and `glorot`.

1.3.1 `__init__` Method

This method initializes the `ParameterInitializer` class and takes an optional argument:

- `initialization`: Specifies the type of initialization to be used for the network parameters. The possible options are:
  - `random`: Uniform random initialization.
  - `gaussian`: Initialization based on the Gaussian (normal) distribution.
  - `glorot`: Glorot initialization, also known as Xavier initialization, which helps in stabilizing the variance of inputs across layers.

1.3.2 `initialize_parameters` Method

This method is responsible for initializing the weights and biases for all layers of the neural network.

- **Inputs**:
  - `inputs`: The number of input nodes in the network.

  - `hidden_layers`: A list of integers representing the number of neurons in each hidden layer.
  - `outputs`: The number of output nodes.

- **Outputs**:
  - A dictionary containing the initialized parameters:
    - `W1`, `b1`: Weights and biases for the first layer.
    - `W{i+1}`, `b{i+1}`: Weights and biases for the subsequent hidden layers, where `i` is the index of the layer.
    - `W{len(hidden_layers) + 1}`, `b{len(hidden_layers) + 1}`: Weights and biases for the output layer.

- **Initialization Methods**:
  - `random`: Weights and biases are initialized using a uniform random distribution.
  - `gaussian`: Weights and biases are initialized using a Gaussian (normal) distribution.
  - `glorot`: Weights are initialized using the Glorot uniform distribution, which ensures that the variance of inputs and outputs is preserved across layers. Biases are initialized to zeros.

- **Example Initialization Process**:
  - For each hidden layer, the weights `W` are initialized using the specified method and dimensions based on the number of neurons in the current and previous layers. Biases `b` are initialized accordingly.
  - For the output layer, the weights and biases are initialized similarly based on the number of neurons in the last hidden layer and the number of output nodes.

The initialization of parameters is critical for the convergence and performance of neural networks, as poor initialization can lead to vanishing or exploding gradients.

1.4 Feed Forward Neural Networks

The class `FeedForwardNeuralNets` implements a multi-layer feedforward neural network with options for different activation functions, loss functions, and optimizers.

1.4.1 `__init__` Method

The constructor method initializes the neural network by setting up the parameters and hyperparameters.

- **Inputs**:
  - `inputs`: The input data for the network.

  - `hidden_layers`: A list of integers representing the number of neurons in each hidden layer.
  - `outputs`: The target or output data of the network.
  - `g`: The activation function for the hidden layers, defaulting to Sigmoid.
  - `L`: The loss function, defaulting to CrossEntropyLoss.
  - `O`: The output layer activation function, defaulting to Softmax.
  - `eta`: The learning rate, defaulting to 0.01.
  - `optimizer`: The optimization method, defaulting to "gd" (Gradient Descent). Options include "adam" for Adam optimization.
  - `initialization_method`: The method to initialize the weights, defaulting to "glorot".
  - `batch_size`: The size of mini-batches for training, defaulting to 32.
  - `beta1`, `beta2`: Parameters for the Adam optimizer.
  - `epsilon`: A small value to prevent division by zero in the Adam optimizer.
  - `t`: The time step used for Adam optimization.

- **Outputs**:
  - Initializes network parameters (weights and biases) using the `ParameterInitializer` class.
  - Sets up optimization-specific variables like `v` and `s` for Adam optimization.

1.4.2 `forward_propogation` Method

This method performs forward propagation through the neural network layers.

- **Inputs**:
  - `x`: The input to the network.

- **Outputs**:
  - `y_pred`: The predicted output of the neural network after passing through the layers and applying the activation functions.

- **Process**:
  - Computes activations for each layer by performing matrix multiplication between weights and inputs, adding biases, and applying the activation function `g`.
  - The final layer applies the output activation function `O`, typically Softmax.

1.4.3 `backPropogation` Method

This method performs backpropagation to compute the gradients of the loss with respect to the network's parameters.

- **Inputs**:
  - `y_pred`: The predicted output.
  - `y`: The actual target values.
  - `x`: The input to the network.
- **Process**:
  - Starts with the error at the output layer and propagates it backward through the network, computing gradients for weights and biases at each layer.
  - Uses the derivative of the activation function for each layer during backpropagation.

1.4.4 `gradient_descent` Method

This method updates the network's parameters using gradient descent.

- **Process**:
  - For each weight and bias, subtracts the product of the learning rate `eta` and the computed gradient.

1.4.5 `adam` Method

This method implements the Adam optimization algorithm to update the network's parameters.

- **Process**:
  - Uses running averages of both the gradient and the squared gradient to update the parameters.
  - Corrects for bias in the estimates of these running averages.

1.4.6 `train` Method

This method trains the neural network over a specified number of epochs.

- **Inputs**:
  - `epochs`: The number of epochs to train the model.
  - `validation_data` (optional): Data for validation during training.

- **Process**:
  - Shuffles the data and divides it into batches.
  - For each batch, computes the loss and gradients, updates the parameters using the optimizer, and stores the training loss.
  - If validation data is provided, computes validation loss at regular intervals.

1.4.7 `evaluate` Method

This method evaluates the network for given input data.

- **Inputs**:
  - `X`: The input data.
- **Outputs**:
  - The predicted output of the network.

1.4.8 `predict` Method

This method predicts the class label for given input data using the output of the `evaluate` method.

- **Inputs**:
  - `X`: The input data.
- **Outputs**:
  - The class label with the highest probability.

2 Data Preparation and Preprocessing

The dataset is prepared by dividing it into training and validation sets, and subsequently preprocessing the data to be fed into the neural network.

2.1 Dataset Splitting

The original dataset is split into a training set and a validation set. A fixed number of 10,000 samples are used for validation, and the rest are assigned to the training set. This allows the model to be evaluated on a separate validation set during the training process, helping to monitor the model's performance and avoid overfitting.

2.2 Data Loading

The training and validation datasets are loaded using PyTorch's `DataLoader`. The training data is shuffled during loading to ensure that the model does not learn any sequence or order from the input data. Shuffling enhances generalization by making sure the model is exposed to different distributions of the data across different training iterations. The validation data is not shuffled as it is only used for evaluation purposes.

2.3 Extracting Data

Once the data is loaded into memory, the features (input data) and labels (target values) are extracted from the data loaders. This involves converting the data from tensors into NumPy arrays, which are required for further manipulation and preprocessing.

2.4 Data Preprocessing

The data preprocessing stage involves normalizing, standardizing, and reshaping the input data, as well as transforming the target labels into a format suitable for classification tasks.

- **Normalization**: The pixel values of the input images, which originally range from 0 to 255, are normalized to fall within the range [0, 1]. This ensures that the input values are scaled uniformly, which is important for efficient learning by the neural network.
- **Standardization**: To further ensure that the inputs have a consistent distribution, the data is standardized by subtracting the mean and dividing by the standard deviation of the input values. This process helps the network converge faster by providing inputs with a zero mean and unit variance.
- **Flattening**: The images, which are initially represented as multi-dimensional arrays (e.g., 28x28 for grayscale images), are reshaped into one-dimensional arrays. This flattening step is necessary because neural networks expect input data to be in a vectorized format.
- **One-hot Encoding**: For classification tasks, the labels are transformed into a one-hot encoded format. This represents each class label as a binary vector, where the correct class is represented by a 1, and all other classes are represented by 0. This format is essential for using classification loss functions like cross-entropy.

2.5 Preprocessed Data

The training and validation datasets are now preprocessed into normalized, standardized, and one-hot encoded formats. The input data is ready to be fed into the neural network for training, and the labels are appropriately formatted for classification.

3 Baseline Model

The baseline model used in this project is a fully connected feedforward neural network. It has three hidden layers and is trained using gradient descent (GD) optimization with a batch size of 64. The model's architecture, training process, and performance are detailed below.

3.1 Model Architecture

The feedforward neural network architecture for the baseline model consists of:

- **Input Layer**: The input layer receives the training data `X_train`, where each input is a flattened image consisting of 784 features (28x28 pixels).
- **First Hidden Layer**: The first hidden layer has 500 neurons. This layer applies a non-linear activation function (in this case, Sigmoid) to the input data.
- **Second Hidden Layer**: The second hidden layer has 250 neurons and applies the same activation function to its inputs.
- **Third Hidden Layer**: The third hidden layer consists of 100 neurons. This layer further refines the data features.
- **Output Layer**: The output layer uses a softmax activation function to produce the class probabilities for the dataset. The number of neurons in the output layer corresponds to the number of classes in the dataset (for example, 10 classes for digits in MNIST).

3.2 Model Training

The model is trained using the following parameters:

- **Optimizer**: The gradient descent (GD) optimizer is used to update the model's weights based on the loss function. In this case, Cross-Entropy loss is used to measure the model's performance.
- **Batch Size**: The model is trained with a batch size of 64, meaning that 64 samples are processed at a time before updating the weights.
- **Epochs**: The model is trained for 15 epochs. During each epoch, the entire dataset is passed through the network, and the weights are updated based on the gradients calculated from the loss.
- **Validation**: A validation set (`X_val` and `y_val_one_hot`) is used to evaluate the model's performance at each epoch, helping to track the model's generalization.

The total time taken for training the model was calculated by recording the time at the start and end of the training process. On an 8-core CPU, the training process took approximately 44 minutes.

3.3 Training Time

The total time for training the model was measured as the difference between the start and end times of the training process. This time depends on the computational resources and complexity of the model. For this baseline model, the training took approximately 44 minutes on an 8-core CPU.

4 Results for Baseline model

4.1 Training and Test Results

4.1.1 Graph

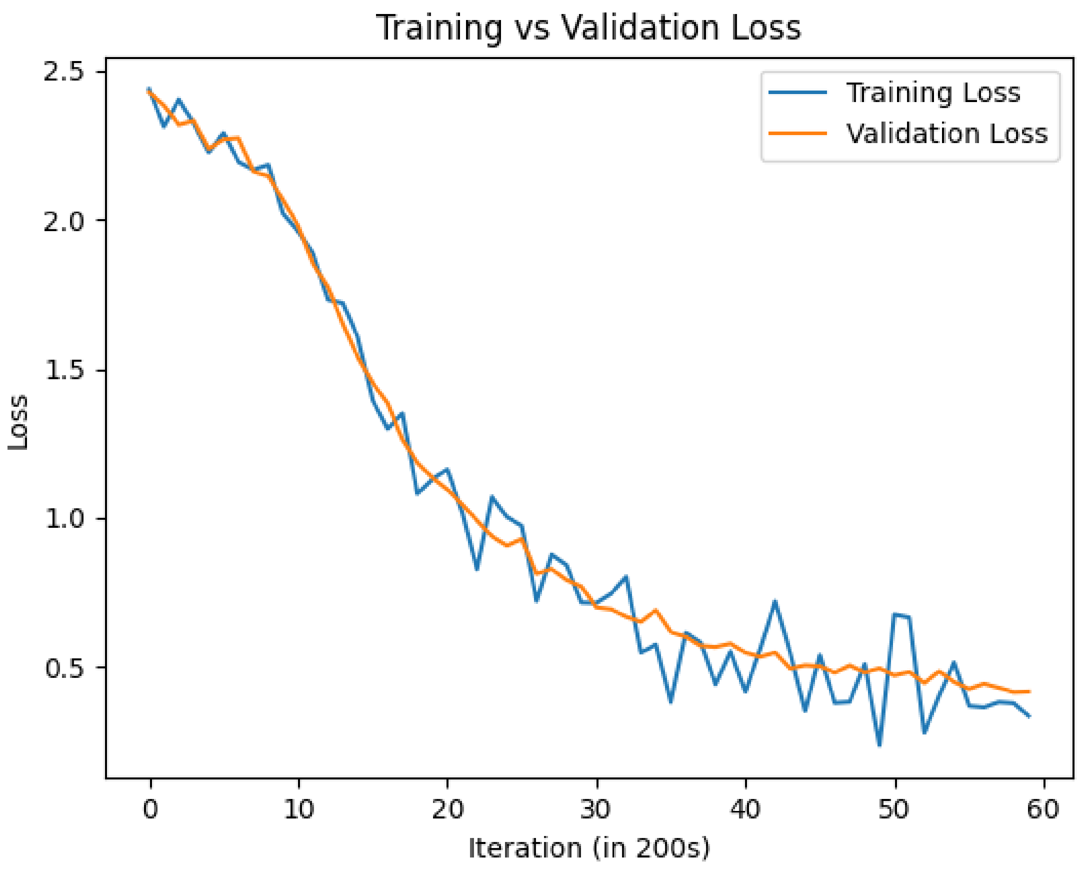


Figure 1: Final Training loss : 0.4487, Final Validation Loss: 0.4499

4.1.2 Training Classification Report

After training the model, the following classification report was generated based on the model's predictions on the validation set:

```
              precision    recall  f1-score   support

           0       0.90      0.97      0.93      4968
           1       0.87      0.98      0.92      5689
           2       0.86      0.86      0.86      4967
           3       0.84      0.86      0.85      5141
           4       0.92      0.78      0.84      4872
           5       0.85      0.79      0.82      4524
           6       0.97      0.86      0.91      4932
           7       0.72      0.96      0.83      5192
           8       0.90      0.77      0.83      4832
           9       0.77      0.68      0.72      4883

    accuracy                           0.85     50000
   macro avg       0.86      0.85      0.85     50000
weighted avg       0.86      0.85      0.85     50000
```

4.1.3 Test Classification Report

```
              precision    recall  f1-score   support

           0       0.89      0.98      0.93       980
           1       0.90      0.99      0.94      1135
           2       0.87      0.86      0.86      1032
           3       0.83      0.88      0.86      1010
           4       0.90      0.78      0.84       982
           5       0.85      0.77      0.81       892
           6       0.97      0.84      0.90       958
           7       0.73      0.95      0.83      1028
           8       0.91      0.77      0.84       974
           9       0.79      0.72      0.76      1009

    accuracy                           0.86     10000
   macro avg       0.86      0.86      0.86     10000
weighted avg       0.86      0.86      0.86     10000
```

It seems that the model performs equivalently well in both seen (train) and unseen (test) data. The model has a good accuracy of **0.85**

5 Training with different activation functions

The default activation function used is sigmoid. There was a trial with 2 other activation functions **ReLU** and **Tanh**

5.1 Results with Tanh activation

5.1.1 Training loss graph

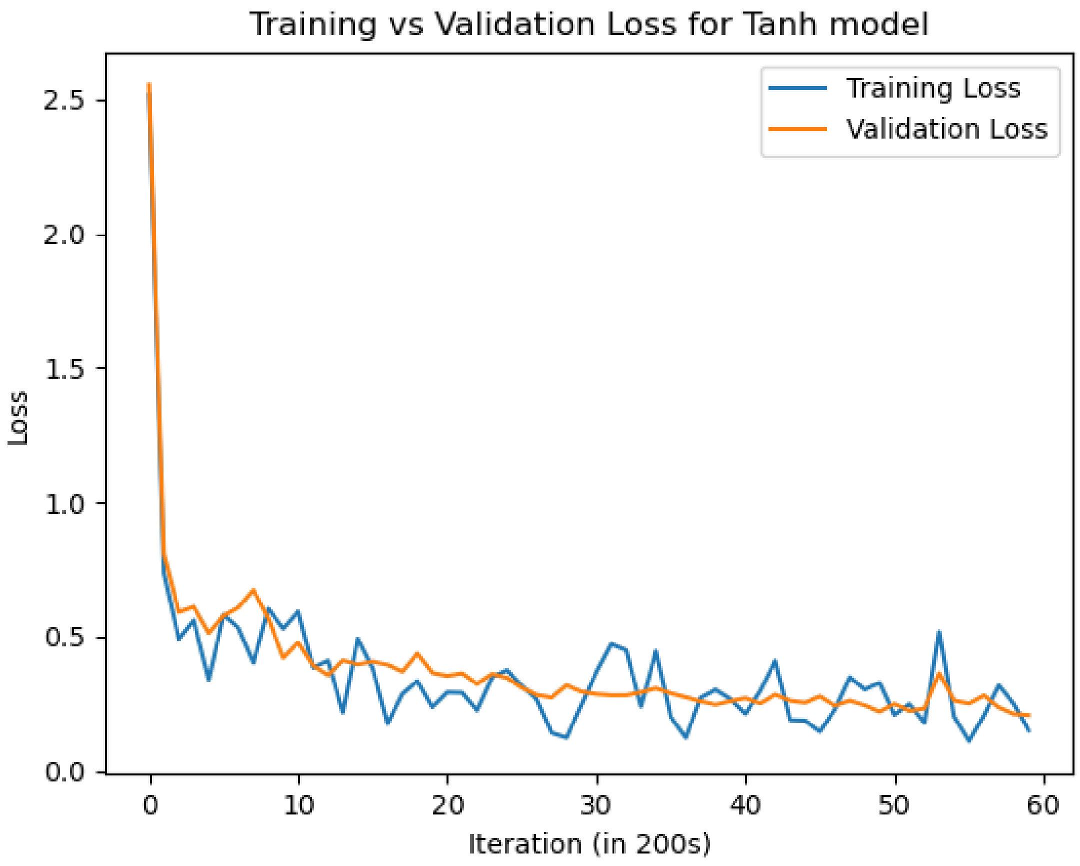


Figure 2: Final Training loss : 0.2275, Final Validation Loss: 0.2076

5.1.2 Training Classification report

```
      precision    recall  f1-score   support

   0       0.96      0.97      0.96      4968
   1       0.98      0.98      0.98      5689
   2       0.93      0.95      0.94      4967
   3       0.94      0.90      0.92      5141
   4       0.96      0.89      0.92      4872
   5       0.95      0.89      0.92      4524
   6       0.98      0.93      0.95      4932
   7       0.97      0.90      0.93      5192
   8       0.85      0.94      0.89      4832
```

```
           9       0.82      0.96      0.88      4883

    accuracy                           0.93     50000
   macro avg       0.93      0.93      0.93     50000
weighted avg       0.93      0.93      0.93     50000
```

5.1.3 Testing Classification report

```
              precision    recall  f1-score   support

           0       0.94      0.99      0.96       980
           1       0.98      0.98      0.98      1135
           2       0.92      0.94      0.93      1032
           3       0.93      0.91      0.92      1010
           4       0.96      0.89      0.93       982
           5       0.94      0.89      0.91       892
           6       0.98      0.91      0.94       958
           7       0.95      0.88      0.92      1028
           8       0.87      0.93      0.90       974
           9       0.83      0.95      0.89      1009

    accuracy                           0.93     10000
   macro avg       0.93      0.93      0.93     10000
weighted avg       0.93      0.93      0.93     10000
```

It seems that the model performs equivalently well in both seen (train) and unseen (test) data. The model has a good accuracy of **0.93**. This is better than the sigmoid activation.

5.2 Results with ReLU activation

5.2.1 Graph

5.2.2 Training Classification report

```
              precision    recall  f1-score   support

           0       0.93      0.99      0.96      4968
           1       0.99      0.97      0.98      5689
           2       0.91      0.96      0.94      4967
           3       0.96      0.92      0.94      5141
           4       0.94      0.95      0.94      4872
           5       0.88      0.95      0.91      4524
           6       0.96      0.96      0.96      4932
           7       0.96      0.93      0.95      5192
```

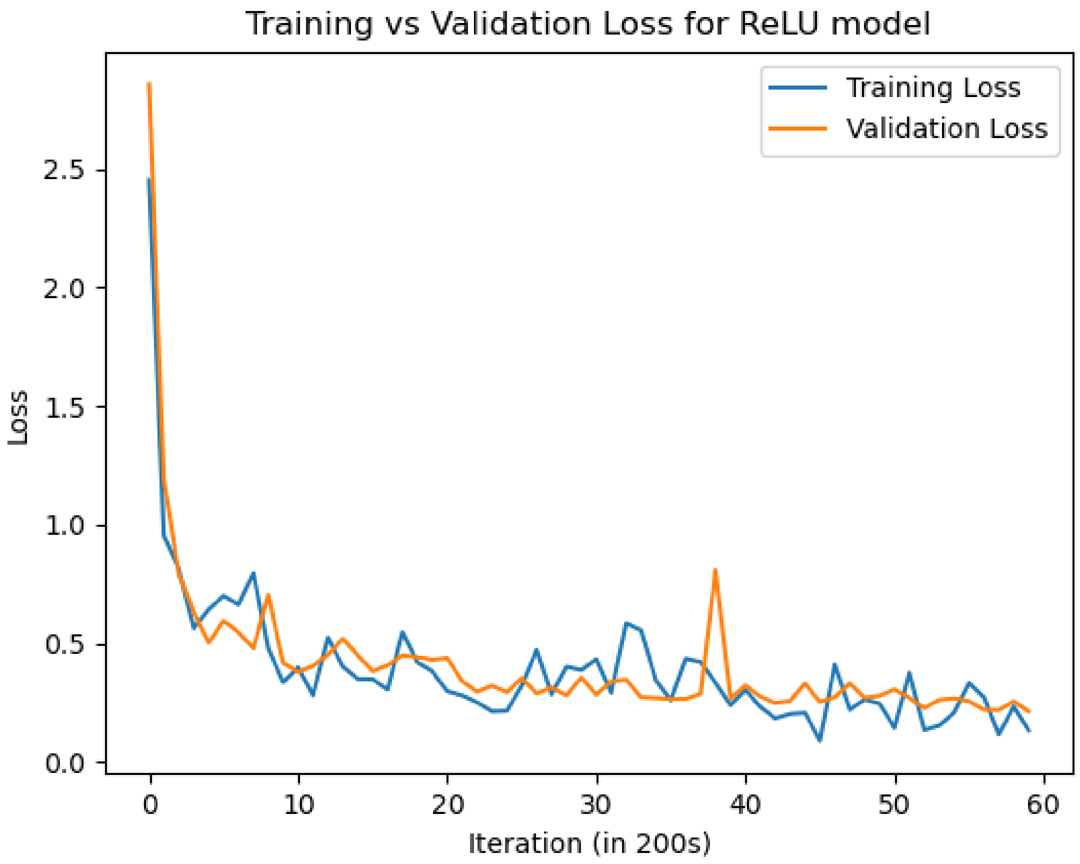


Figure 3: Final Training loss : 0.2233, Final Validation Loss: 0.2131

```
           8       0.95      0.88      0.91      4832
           9       0.92      0.91      0.91      4883

    accuracy                           0.94     50000
   macro avg       0.94      0.94      0.94     50000
weighted avg       0.94      0.94      0.94     50000
```

5.2.3 Testing Classification report

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       980
           1       0.96      0.99      0.98      1135
           2       0.97      0.90      0.93      1032
           3       0.91      0.95      0.93      1010
           4       0.94      0.94      0.94       982
           5       0.92      0.95      0.93       892
           6       0.97      0.94      0.95       958
           7       0.92      0.97      0.94      1028
           8       0.96      0.88      0.92       974
           9       0.91      0.92      0.92      1009

    accuracy                           0.94     10000
   macro avg       0.94      0.94      0.94     10000
weighted avg       0.94      0.94      0.94     10000
```

It seems that the model performs equivalently well in both seen (train) and unseen (test) data. The model has a good accuracy of **0.94**. This is slightly better than Tanh and lot better than sigmoid

6 Package

Used the PyTorch package to mimic the baseline model and here are the interesting results.

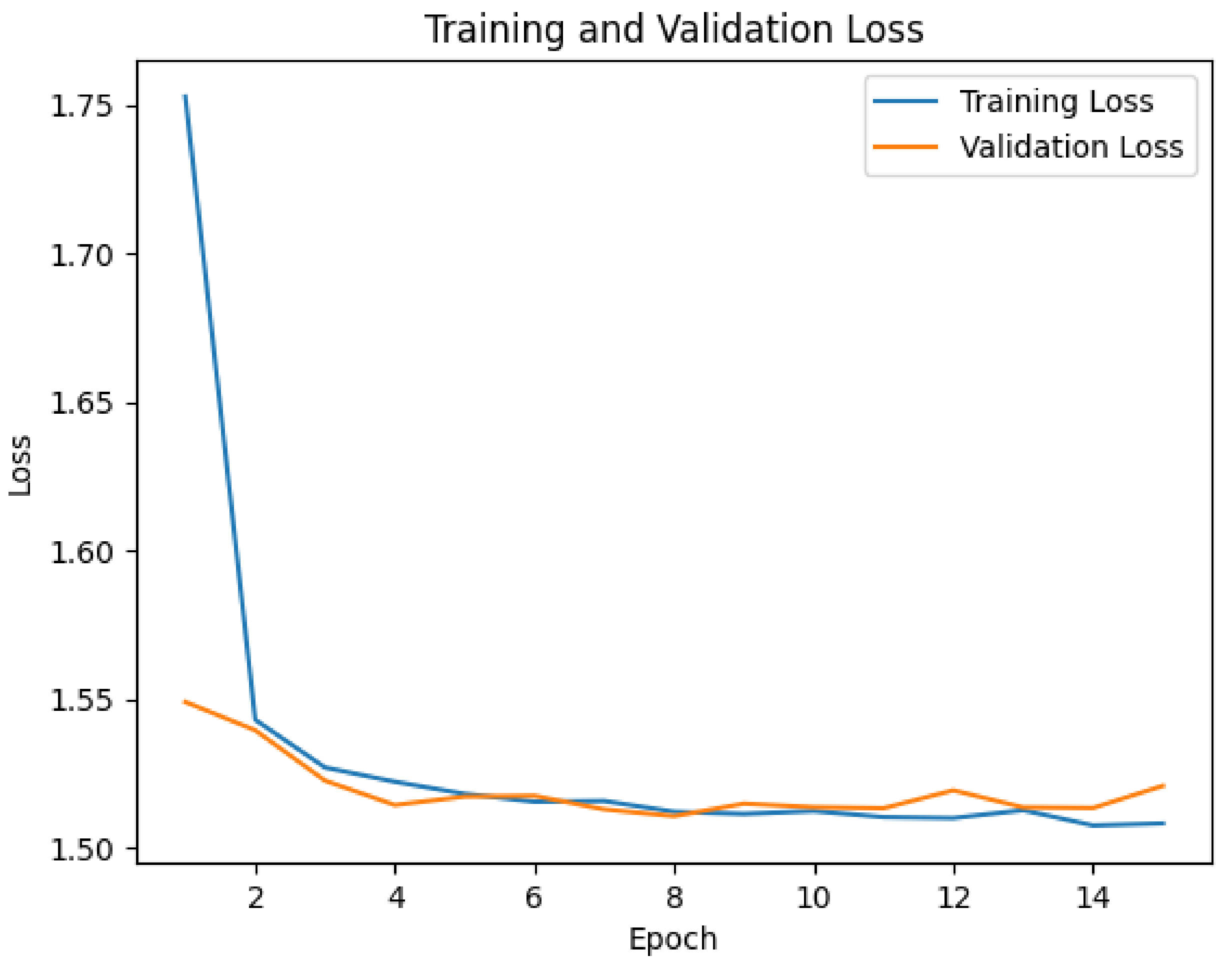
6.1 Loss Graph

6.2 Classification report on Test Data

```
              precision    recall  f1-score   support

           0       0.97      0.98      0.97       980
           1       0.97      0.99      0.98      1135
           2       0.97      0.94      0.96      1032
           3       0.98      0.88      0.92      1010
           4       0.92      0.97      0.95       982
           5       0.94      0.94      0.94       892
           6       0.90      0.97      0.93       958
           7       0.96      0.93      0.94      1028
           8       0.95      0.88      0.92       974
           9       0.86      0.93      0.89      1009

    accuracy                           0.94     10000
   macro avg       0.94      0.94      0.94     10000
weighted avg       0.94      0.94      0.94     10000
```



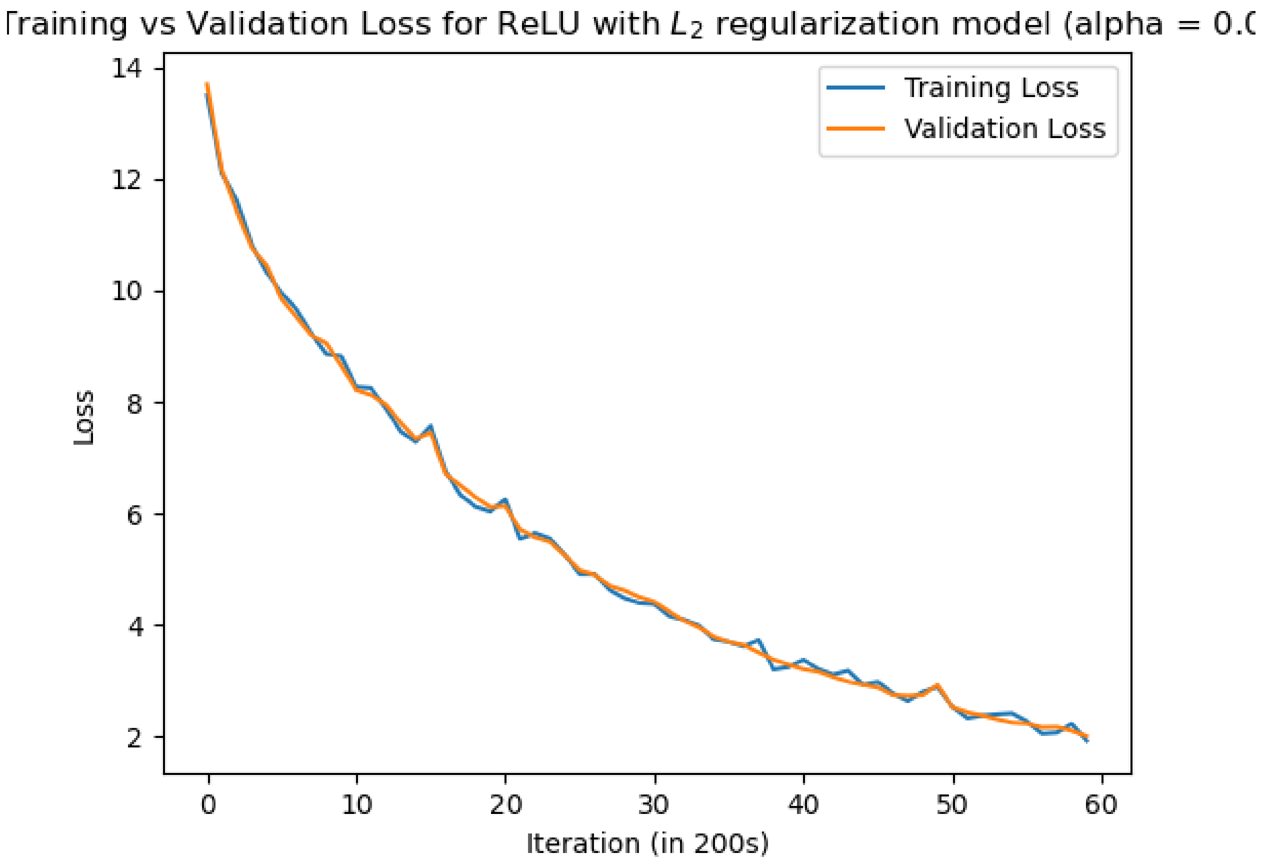
There is a good accuracy of 0.94.

7 Regularization

L_2 regularization is added to the loss function and trained with the best performing architecture. In our case we go with ReLU. Here are the results of that.

7.1 Loss graph

The plot is much smoother than the unregularized version, showing more stable learning. With additional epochs, there's a good chance the model will converge further, leading to a lower loss. This indicates the model is learning better patterns and could perform even more effectively with some fine-tuning.



7.2 Classification report

```
              precision    recall  f1-score   support

           0       0.89      0.94      0.92      4968
           1       0.65      0.98      0.79      5689
           2       0.87      0.72      0.79      4967
           3       0.89      0.62      0.73      5141
           4       0.77      0.85      0.81      4872
           5       0.73      0.78      0.76      4524
           6       0.82      0.92      0.87      4932
           7       0.72      0.89      0.80      5192
           8       0.78      0.58      0.66      4832
           9       0.84      0.47      0.60      4883

    accuracy                           0.78     50000
   macro avg       0.80      0.78      0.77     50000
weighted avg       0.79      0.78      0.77     50000
```

Here we see that there is a significant loss in accuracy even for the train dataset.

8 Other Experiments

There were other experiments which was performed by changing the number of epochs, hidden layers which are present in the `ipynb` notebook. They did not perform better than the baseline model.