

CSE574-D: Introduction to Machine Learning, Fall 2023

Assignment 2: Building Neural Networks and Convolutional Neural Networks

ngupta22 pandey7

October 26, 2023

Contents

1	Introduction	4
2	Background	4
3	Part I: Building a Basic NN	4
3.1	Dataset Overview	4
3.2	Preprocessing and Splitting the Dataset	6
3.3	Neural Network Architecture	6
4	Step 4: Training the Neural Network	7
4.1	Overview	7
4.2	Training Loop	8
4.3	Loss Function	8
4.4	Optimizer	8
4.5	Training Execution	8
4.6	Performance Evaluation	9
4.7	Results Visualization	9
4.8	Conclusion for Part I	11
5	Part II: Optimizing NN	11
5.1	Hyperparameter Tuning	11
5.1.1	Dropout Tuning	11
5.1.2	Optimizer Tuning	12
5.1.3	Activation Function Tuning	14
5.1.4	Initializer Tuning	15
5.1.5	Analysis and Reasoning	16
5.2	Training Optimization Methods	17
5.2.1	Early Stopping	17
5.2.2	Batch Normalization	17
5.2.3	Learning Rate Scheduler	18
5.2.4	Base Model Re-Architecture	18

5.2.5	Analysis and Reasoning	18
6	Conclusion	19
7	Part III: Building a CNN	20
8	Introduction	20
9	Introduction	20
10	Dataset Details	20
10.1	Overview	20
10.2	Preprocessing	20
10.3	Data Split	20
10.4	Data Loading	21
10.5	Channel Verification	21
10.6	Visualization of the Dataset	21
11	CNN Architecture Specification	21
11.1	Input Neurons	21
11.2	Output Neurons	21
11.3	Hidden Layers and Activation Functions	22
11.4	CNN Parameters	22
11.5	Dropout Inclusion	22
11.6	Architecture	22
11.7	Activation Functions	23
11.8	Model Summary	23
11.9	Optimization	23
11.10	Training Procedure	23
12	Improvement Tools on CNN Architectures	24
12.1	Input Neurons	24
12.2	Output Neurons	24
12.3	Activation Functions	25
12.4	Hidden Layers	25
12.5	Layer Parameters	25
12.6	Dropout	25
13	Performance Metrics and Results Analysis	25
13.1	Performance Metrics	25
13.2	Training and Validation Loss	25
13.3	Accuracy Visualization	26
13.4	Confusion Matrix	26
13.5	ROC Curve	27
14	Part IV: VGG-11 Implementation	28
14.1	Model Details	28
14.2	Training Procedure	29

15 Comparison of CNN Architectures	29
15.1 Architectural Complexity	30
15.1.1 Layer Depth	30
15.1.2 Filter Progression	30
15.2 Input Image Processing	30
15.2.1 Image Resizing	30
15.3 Model Parameters	30
15.3.1 Number of Parameters	30
15.4 Regularization Techniques	30
15.4.1 Dropout Usage	30
15.5 Training and Computational Requirements	30
15.5.1 Training Time	30
15.5.2 Memory Consumption	31
15.6 Expected Model Performance	31
15.6.1 Accuracy	31
15.7 Conclusion	31
16 Performance Metrics and Comparison	31
16.1 Challenges with VGG-11 Implementation	31
16.2 Implications for Performance Metrics	31
16.3 Discussion	32
16.4 Conclusion	32
17 References	32

1 Introduction

In the rapidly evolving field of machine learning, neural networks stand as a cornerstone, powering many modern applications and technologies. This report details our exploration of neural networks and convolutional neural networks as part of the CSE574-D: Introduction to Machine Learning course. Through a systematic approach, we delve into building, optimizing, and analyzing neural networks using the PyTorch framework. By detailing our methodology, dataset handling, architectural decisions, and optimization strategies, we aim to provide comprehensive insights into the nuances and challenges of crafting effective neural network models.

2 Background

Neural networks, often inspired by biological neural systems, consist of layers of interconnected nodes or "neurons." These structures are particularly adept at identifying patterns and relationships within data, making them indispensable in various machine learning tasks, from image recognition to natural language processing.

The significance of neural networks in machine learning is multifaceted:

- **Flexibility:** Neural networks can approximate any function, given sufficient data and computational power.
- **End-to-end learning:** They can learn directly from raw data without the need for manual feature extraction.
- **Transferability:** Pre-trained models can be fine-tuned for new tasks, leveraging previously learned features.

PyTorch, developed by Facebook's AI Research lab, provides a dynamic computational graph, making it particularly suited for building and training neural network models. Its intuitive syntax, flexibility, and rich ecosystem have made it a preferred choice for both researchers and industry professionals. For this assignment, we utilized PyTorch to design and implement our neural network architectures.

Throughout this report, we will discuss our journey of understanding, implementing, and refining neural network models, shedding light on our design choices, challenges faced, and the results obtained.

3 Part I: Building a Basic NN

3.1 Dataset Overview

- **Description and nature of the dataset:** The dataset consists of seven features (f_1, f_2, \dots, f_7) and a target variable. The dataset has a total of 766 entries. Some columns were of object datatype which were then converted to numeric.
- **Main statistics about the dataset entries:** Before preprocessing, the dataset had 760 entries and 8 columns. After preprocessing and removal of outliers, 654 entries remained. The dataset was then normalized, and the range of all features was scaled between 0 and 1. The dataset was saved in the 'datasets_processed' directory with the name 'datasetProcessed.csv'.

- **Visualization graphs with short descriptions:**

- *Box Plot:* This plot provides a visual summary of the minimum, first quartile, median, third quartile, and maximum of each column.
- *Histogram:* This plot provides a frequency distribution of each column, giving an idea about the distribution of each feature.
- *Scatter Plot:* Scatter plots for each feature vs. index were plotted to visualize the spread and detect outliers.

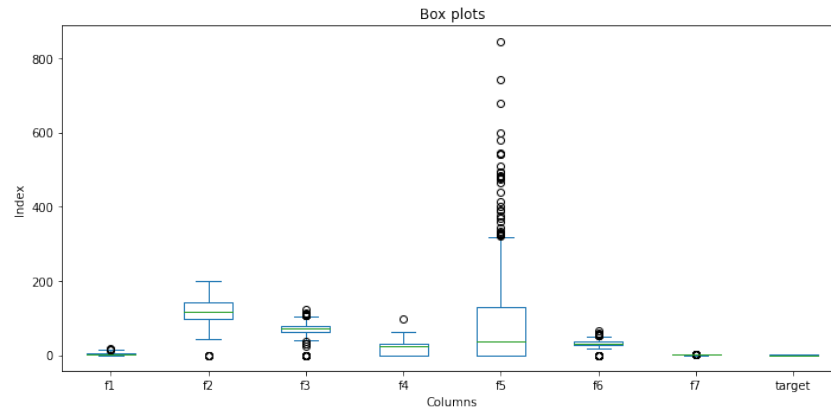


Figure 1: Box plot of the dataset.

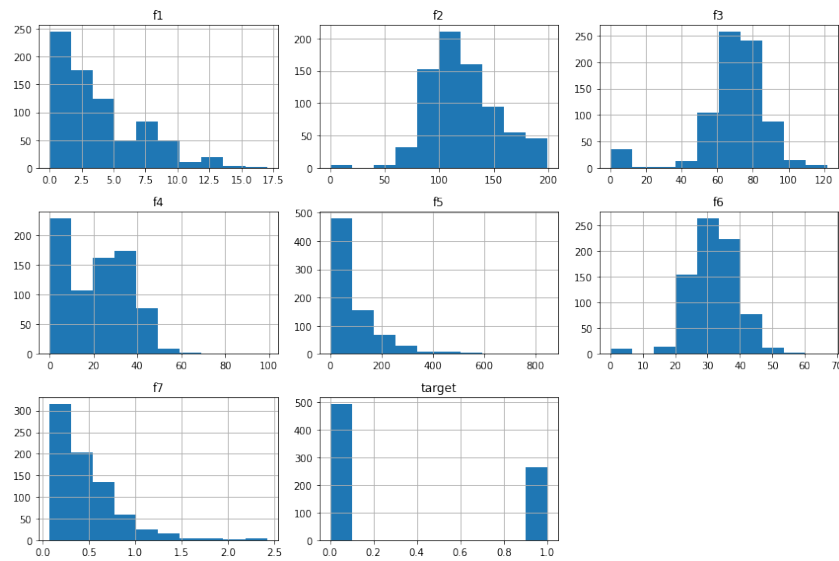


Figure 2: Histogram of the dataset.

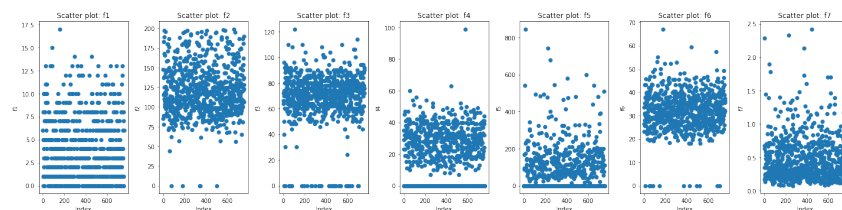


Figure 3: Scatter plot of the dataset.

3.2 Preprocessing and Splitting the Dataset

- **Preprocessing techniques employed:**

After loading the processed dataset, the entries are observed to have been normalized, with values ranging between 0 and 1. This ensures that each feature contributes approximately proportionately to the final distance. There are a total of 654 data points, with each data point having seven features (f_1, f_2, \dots, f_7) and a target variable.

A crucial preprocessing step is the split of the dataset into training, validation, and testing sets. In machine learning and particularly in the field of deep learning, it's standard to split the dataset into three subsets:

1. Training set: Used to train the model.
2. Validation set: Used to validate the model during training, adjust hyperparameters, and prevent overfitting.
3. Testing set: Used to test the model's performance after training.

- **Rationale behind splitting the dataset into training, testing, and validation sets:**

The dataset was split into an 80:10:10 ratio for the training, validation, and testing sets, respectively. This is a common practice to ensure that the model has enough data to learn from while also having separate sets to tune and test its performance.

The 'train_test_split' function from the scikit-learn library was used for this purpose. Using a random state of 42 ensures reproducibility. After the split:

- Training set has 523 samples.
- Validation set has 65 samples.
- Testing set has 66 samples.

Keeping a separate validation set is crucial during the model training phase. It helps in hyperparameter tuning and gives an indication of the model's performance on unseen data without touching the test set. The test set, on the other hand, provides a final, unbiased performance evaluation of the model.

3.3 Neural Network Architecture

- **Design choices for the neural network:**

The designed neural network consists of three main layers, starting from the input layer, which receives the seven features of the dataset. The first layer is fully connected and maps the input features to 128 neurons. The second layer, also fully connected, maps the 128 neurons to 64 neurons. Finally, the output layer maps the 64 neurons to a single output neuron using a sigmoid activation function, making it suitable for binary classification tasks.

Between the layers, dropout is used as a regularization technique to prevent overfitting. A dropout rate of 0.2 was chosen, which means during training, approximately half of the neurons in the dropout layer are turned off, forcing the network to learn redundant representations, and in turn, making it more robust.

- **Activation functions, hidden layers, and other parameters:**

The ELU (Exponential Linear Unit) activation function was chosen for the hidden layers. ELU is a widely used activation function in deep learning due to its non-linearity and computational efficiency. The final output layer uses a sigmoid activation function, suitable for binary classification which gives an output between 0 and 1, indicating the probability of the positive class.

In terms of architecture:

- Input Layer: 7 neurons (corresponding to the 7 features).
- Hidden Layer 1: 128 neurons with ReLU activation.
- Dropout with 0.2 rate.
- Hidden Layer 2: 64 neurons with ReLU activation.
- Dropout with 0.2 rate.
- Output Layer: 1 neuron with sigmoid activation.

- **Summary of the model using Torchinfo:**

The model has a total of 9,345 parameters, all of which are trainable. The model's architecture is broken down as follows:

1. Linear layer mapping from 7 to 128 neurons.
2. ELU activation.
3. Dropout with 0.2 rate.
4. Linear layer mapping from 128 to 64 neurons.
5. ELU activation.
6. Dropout with 0.2 rate.
7. Linear layer mapping from 64 to 1 neuron.
8. Sigmoid activation.

The total memory footprint, including the model's parameters and the forward/backward pass, is approximately 0.04 MB.

4 Step 4: Training the Neural Network

4.1 Overview

In this section, we detail the methodology used to train our neural network. The process can be summarized in the following steps:

1. Setting up the training loop.
2. Defining the loss function.
3. Selecting an optimizer and specifying a learning rate.
4. Running the training loop.

5. Evaluating the model performance on the testing data.
6. Saving the model weights.
7. Visualizing the results.

4.2 Training Loop

The main component of our training methodology is the training loop. This loop runs for a pre-defined number of epochs. Within each epoch, the following operations are performed:

1. The model is set to training mode.
2. We iterate over the training data in batches. For each batch:
 - A forward pass through the neural network is computed.
 - The loss, which measures the discrepancy between the model's predictions and the actual labels, is calculated.
 - Gradients are computed using backpropagation.
 - The model's weights are updated using the optimizer.
3. Post training on each epoch, the model is set to evaluation mode. The validation loss is computed over the validation dataset. Unlike the training phase, there's no backward propagation or weight updating during validation.

4.3 Loss Function

We utilize the Binary Cross Entropy Loss function, which is suitable for binary classification problems. This function measures the error between the predicted output of the neural network and the true labels of the training data.

4.4 Optimizer

The chosen optimizer for updating the model's weights during training is Adam. It's a popular optimization algorithm known for its efficiency. The learning rate for Adam is set to 0.001.

4.5 Training Execution

Our neural network is trained over 100 epochs with a batch size of 16. Throughout the training, the losses for training, validation, and testing datasets are monitored and printed. This allows us to observe the model's performance over time and ensure that it's not overfitting to the training data.

4.6 Performance Evaluation

Upon completion of training, the model's performance is evaluated on the testing dataset. The metrics measured include:

- Accuracy: The proportion of correctly classified instances out of the total instances.
- Precision: The ratio of correctly predicted positive observations to the total predicted positives.
- Recall: The ratio of correctly predicted positive observations to all actual positives.
- F1 Score: The weighted average of Precision and Recall.

An accuracy of 79% suggests that our model correctly predicts the target variable for a majority of instances in the testing set. However, with a precision of 67%, it indicates that there might be some false positives. The recall of 60% suggests that the model might be missing out on some actual positive instances, which is further confirmed by an F1 score of 63%, which is the harmonic mean of precision and recall.

4.7 Results Visualization

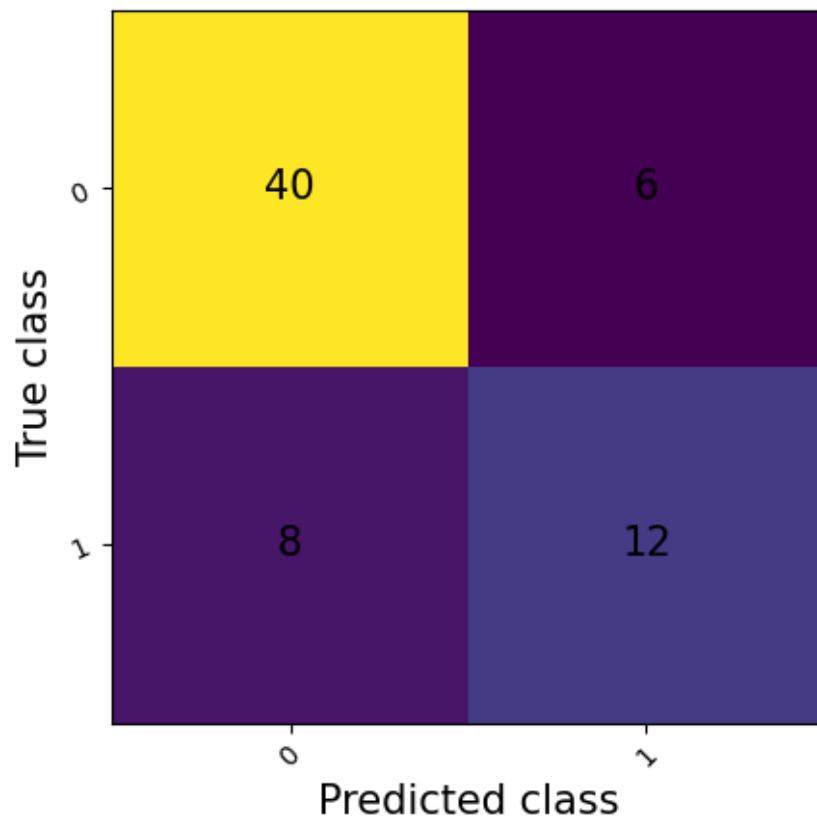


Figure 4: Confusion Matrix of the model's predictions.

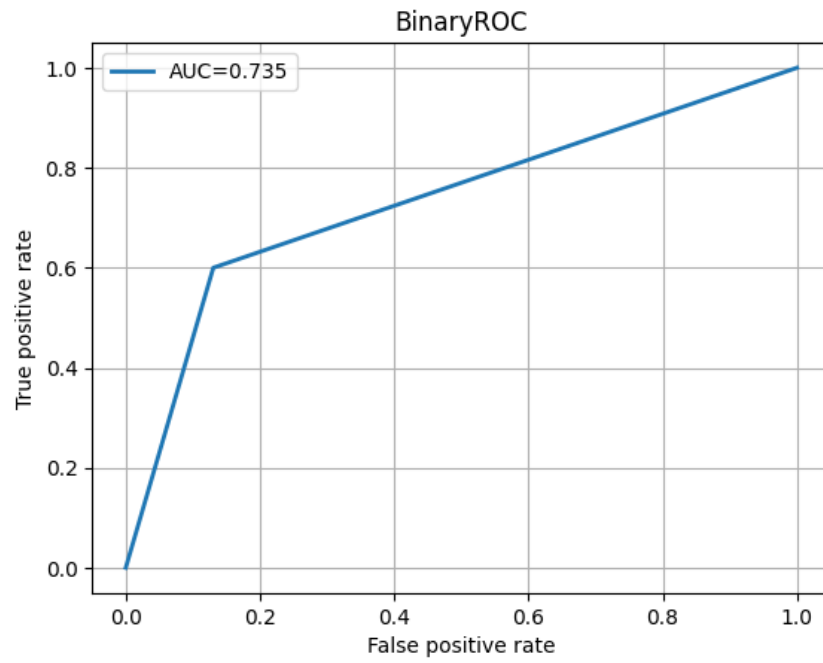


Figure 5: Receiver Operating Characteristic (ROC) curve.

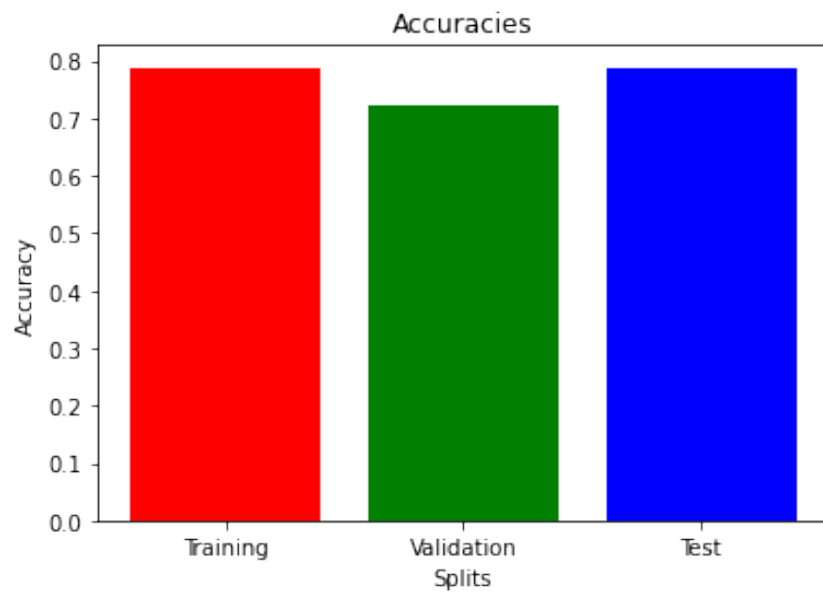


Figure 6: Comparison of test, validation, and training accuracy.

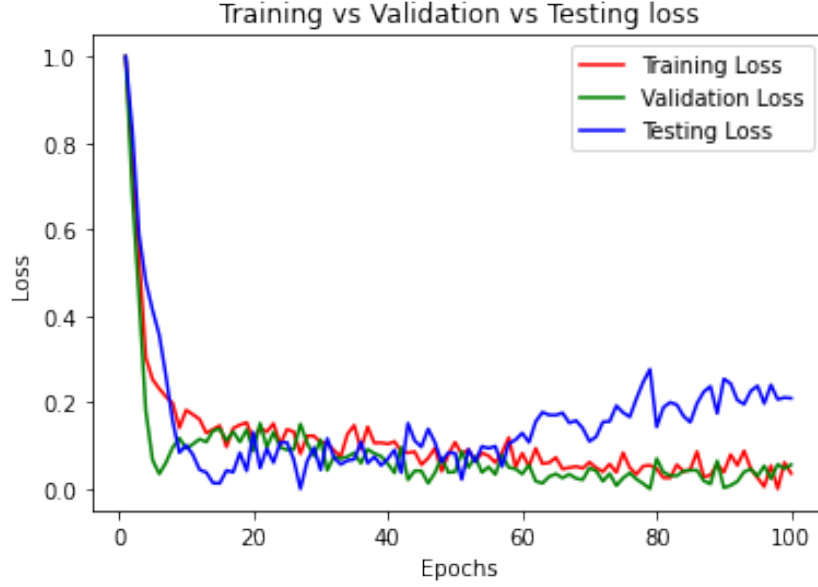


Figure 7: Comparison of test, validation, and training loss.

4.8 Conclusion for Part I

In Part I, we undertook the task of building a basic Neural Network using PyTorch. By preprocessing our data and designing a three-layer neural network, we achieved a commendable accuracy of 79% on the testing dataset. While the results are promising, there's room for improvement, especially in terms of precision and recall. Future iterations could focus on refining the architecture and experimenting with different hyperparameters.

5 Part II: Optimizing NN

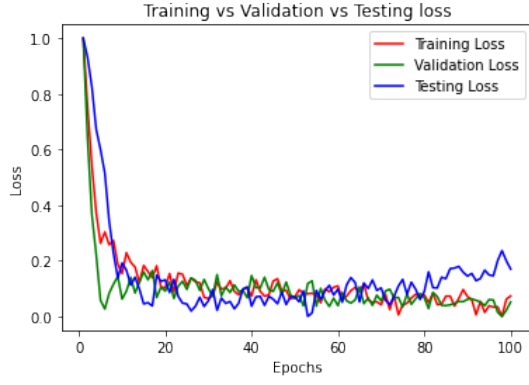
5.1 Hyperparameter Tuning

5.1.1 Dropout Tuning

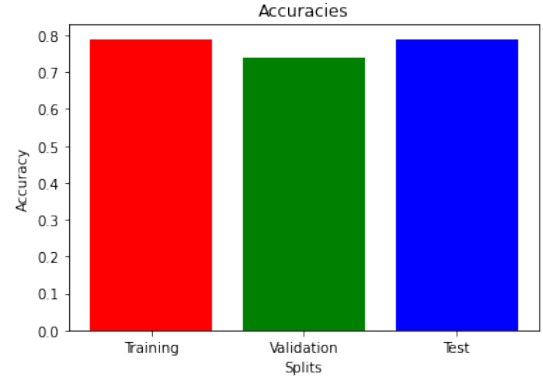
The dropout hyperparameter was modified while keeping all other parameters constant. The results of the different setups are shown in the table below.

Setup	Dropout Value	Test Accuracy	F-Score
1	0.25	0.79	0.63
2	0.5	0.79	0.65
3	0.75	0.77	0.59

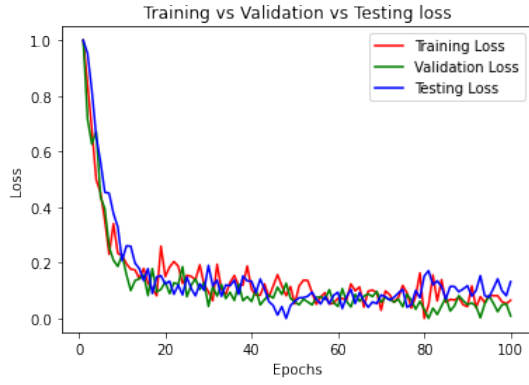
Table 1: Results for different dropout values.



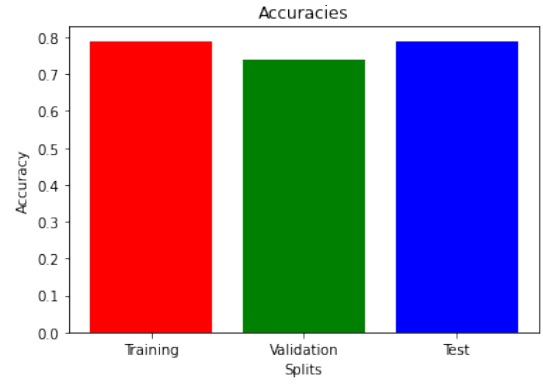
(a) Loss vs Epochs for Dropout 0.25



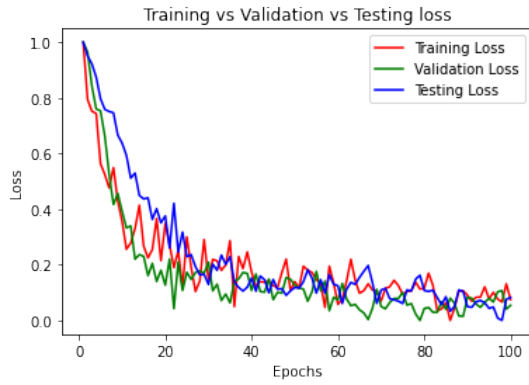
(b) Accuracy vs Epochs for Dropout 0.25



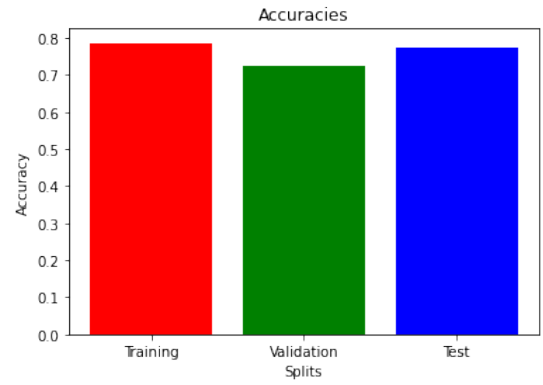
(c) Loss vs Epochs for Dropout 0.5



(d) Accuracy vs Epochs for Dropout 0.5



(e) Loss vs Epochs for Dropout 0.75



(f) Accuracy vs Epochs for Dropout 0.75

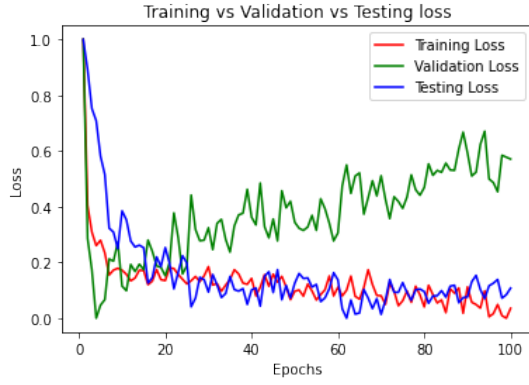
Figure 8: Graphs depicting the effect of different dropout values on model training.

5.1.2 Optimizer Tuning

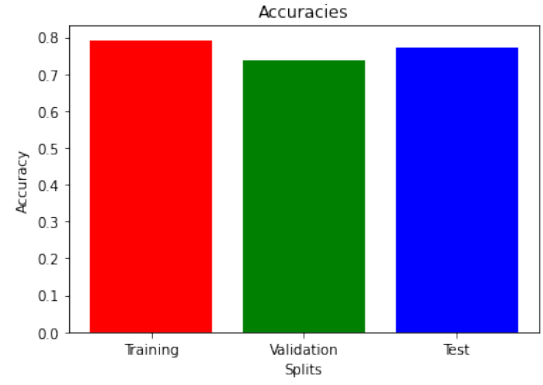
The optimizer was another hyperparameter that was tuned. The results of the different setups are presented below:

Setup	Optimizer	Test Accuracy	F-Score
1	RMSprop	0.77	0.63
2	SGD	0.79	0.63
3	AdamW	0.79	0.61

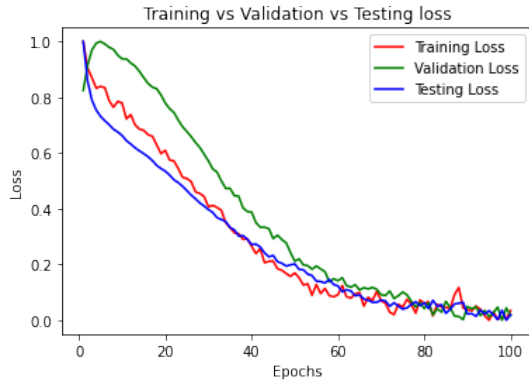
Table 2: Results for different optimizers.



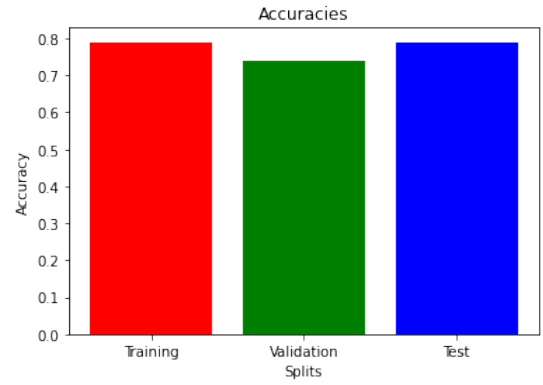
(a) Loss vs Epochs for RMSprop



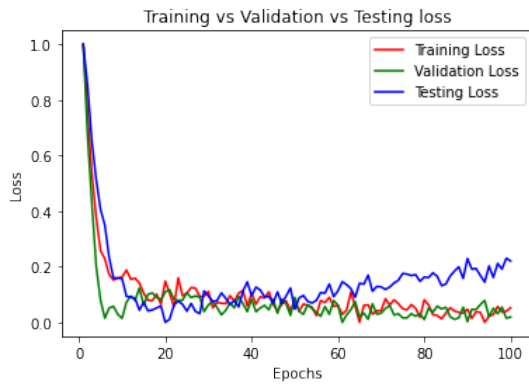
(b) Accuracy vs Epochs for RMSprop



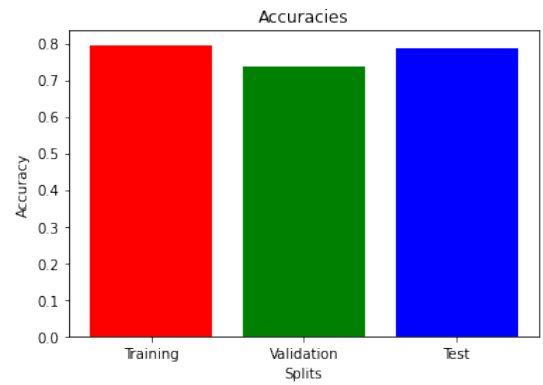
(c) Loss vs Epochs for SGD



(d) Accuracy vs Epochs for SGD



(e) Loss vs Epochs for AdamW



(f) Accuracy vs Epochs for AdamW

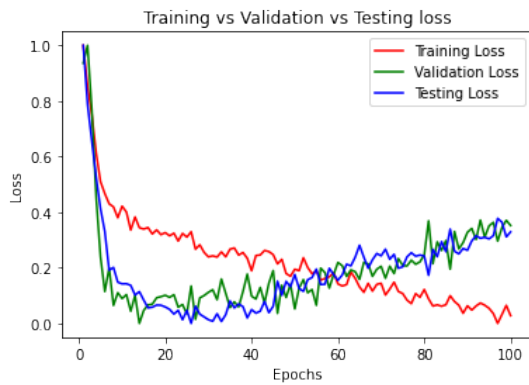
Figure 9: Graphs depicting the effect of different optimizer tuning on model training.

5.1.3 Activation Function Tuning

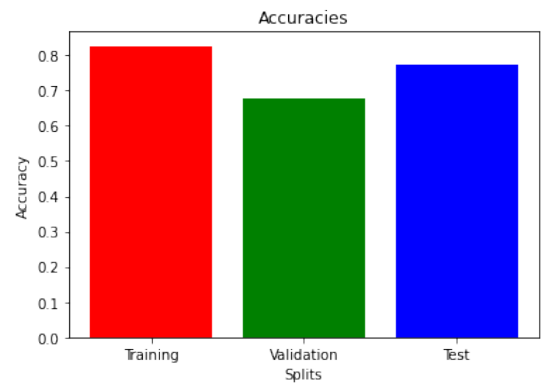
The activation function was another hyperparameter that was tuned. The results of the different setups are presented below:

Setup	Activation Function	Test Accuracy	F-Score
1	LeakyReLU	0.77	0.59
2	ReLU	0.76	0.56
3	SELU	0.80	0.63

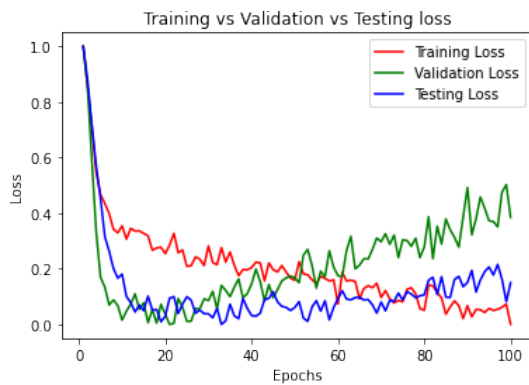
Table 3: Results for different activation functions.



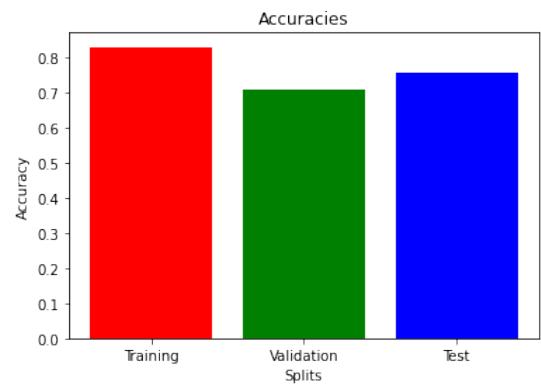
(a) Loss vs Epochs for LeakyReLU



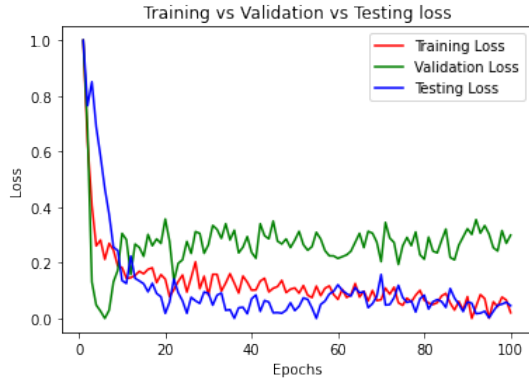
(b) Accuracy vs Epochs for LeakyReLU



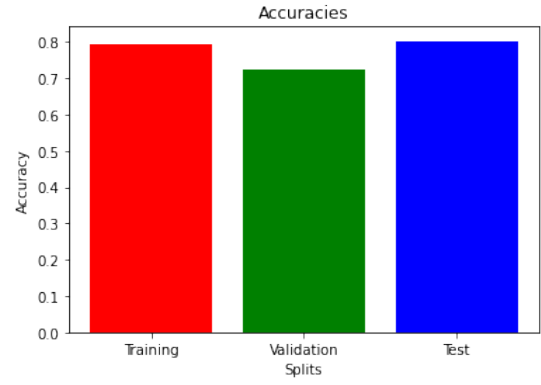
(a) Loss vs Epochs for ELU



(b) Accuracy vs Epochs for ELU



(a) Loss vs Epochs for SELU



(b) Accuracy vs Epochs for SELU

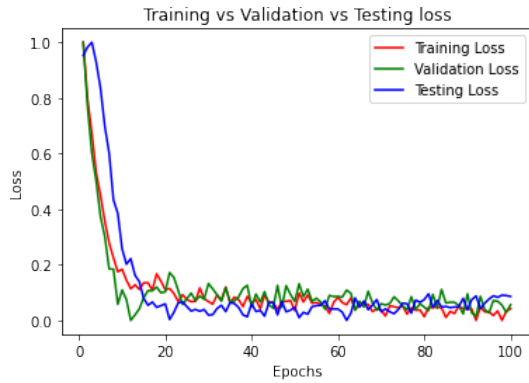
Figure 12: Graphs depicting the effect of different activation function on model training.

5.1.4 Initializer Tuning

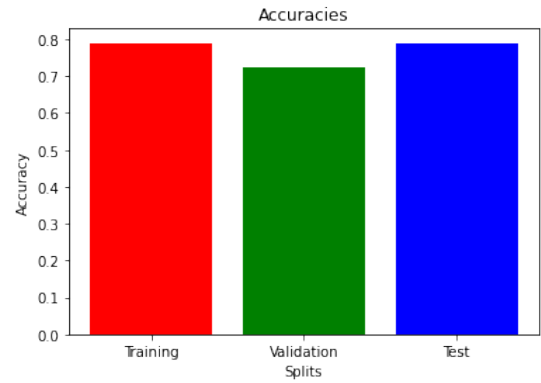
The weight initializer was the next hyperparameter to be adjusted. Results for different setups are as follows:

Setup	Initializer	Test Accuracy	F-Score
1	Xavier Uniform	0.79	0.61
2	He Uniform	0.76	0.60
3	Uniform (a=0, b=0.1)	0.79	0.61

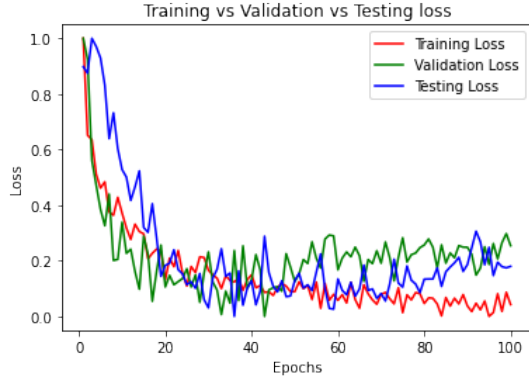
Table 4: Results for different initializers.



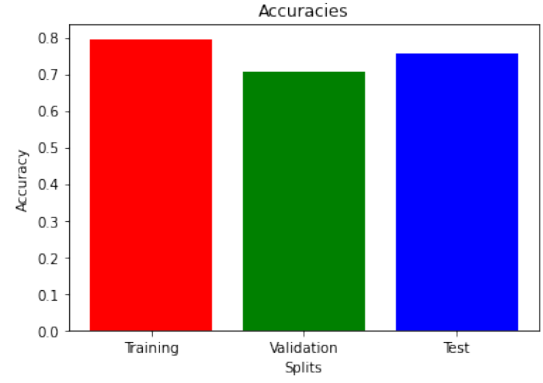
(a) Loss vs Epochs for Xavier Uniform



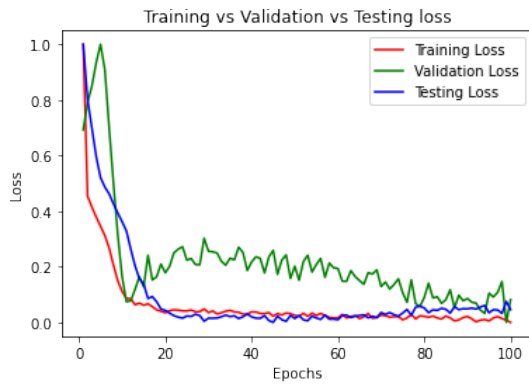
(b) Accuracy vs Epochs for Xavier Uniform



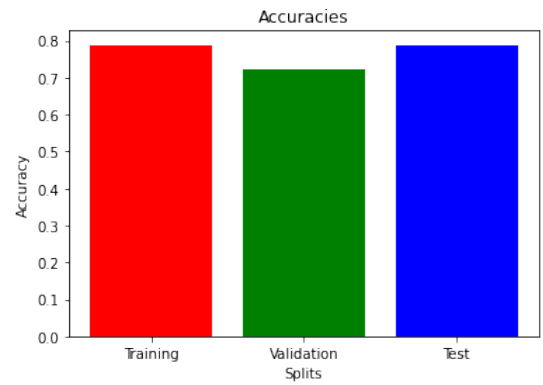
(a) Loss vs Epochs for He Uniform



(b) Accuracy vs Epochs for He Uniform



(a) Loss vs Epochs for Uniform($a=0$, $b=0.1$)



(b) Accuracy vs Epochs for Uniform($a=0$, $b=0.1$)

Figure 15: Graphs depicting the effect of different Initializer Tuning on model training.

5.1.5 Analysis and Reasoning

- **Dropout Tuning:** Varying dropout values showed nuanced performance results. While a dropout of 0.25 achieved an accuracy of 0.79, increasing it to 0.5 and 0.75 both resulted in accuracies of 0.79 and 0.77 respectively. This indicates that beyond a certain threshold, increasing dropout doesn't guarantee improved performance.
- **Optimizer Tuning:** The optimizer's choice had clear implications on performance. RMSprop and SGD yielded accuracies of 0.77 and 0.79 respectively, while AdamW lagged slightly with 0.79. This underlines the importance of selecting the appropriate optimizer for a given problem.
- **Activation Function Tuning:** Different activation functions yielded varied results. SELU achieved the highest accuracy at 0.80, while LeakyReLU and ReLU recorded 0.77 and 0.76 respectively. This emphasizes the subtle, yet important impact of the activation function on model performance.
- **Initializer Tuning:** Among the weight initializers used, the He initializer led with an accuracy of 0.76. Both Xavier and Uniform initializers (range $a = 0$ to $b = 0.1$) had the same accuracy of 0.79. This highlights the pivotal role of weight initialization in neural network performance.

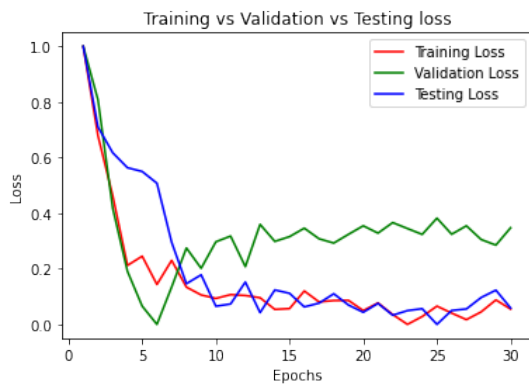
5.2 Training Optimization Methods

Various hyperparameters were employed on the base model and we selected the SeLU optimizer model as our base model as it yielded the highest accuracy among all the hyperparameters. The aim was to further enhance the model's performance. This section details the methods used and their outcomes.

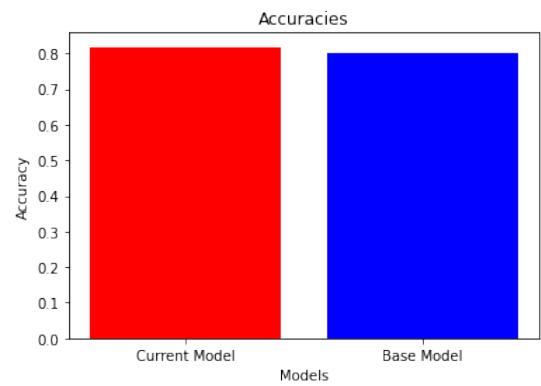
5.2.1 Early Stopping

The first optimization method employed was "Early Stopping". It is designed to prevent overfitting by halting the training process once the model's performance on a validation dataset starts to degrade, ensuring that the model does not continue to adapt too closely to the training data at the expense of generalization.

Outcome: The model achieved a test accuracy of 0.82.



(a) Loss vs Epochs for Early Stopping

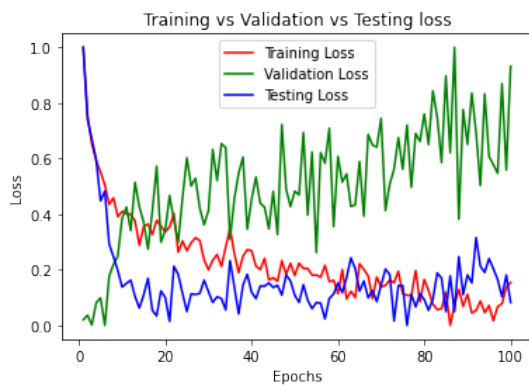


(b) Accuracy vs Epochs for Early Stopping

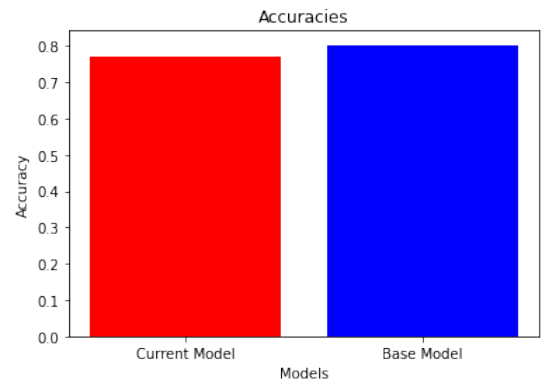
5.2.2 Batch Normalization

Next, "Batch Normalization" was integrated into the neural network architecture. Batch normalization standardizes the activations from a prior layer to have a mean output activation of zero and standard deviation of one, stabilizing the learning process and potentially accelerating the training speed.

Outcome: The model recorded a test accuracy of 0.77.



(a) Loss vs Epochs for Batch Normalization

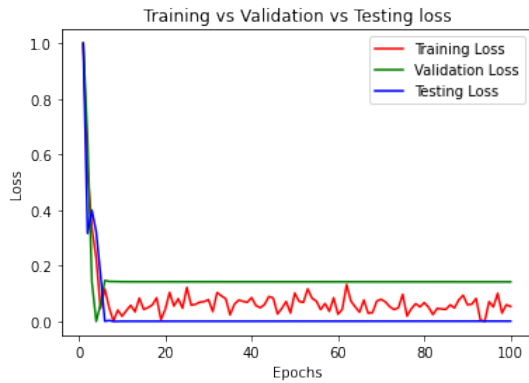


(b) Accuracy vs Epochs for Batch Normalization

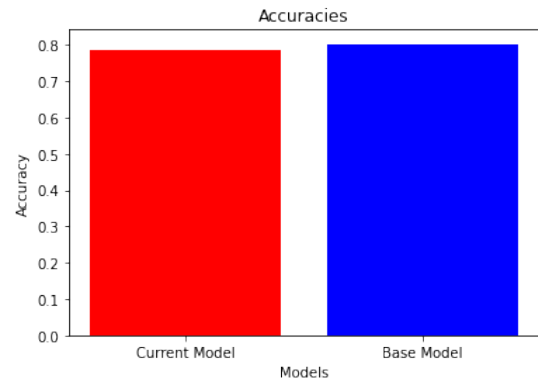
5.2.3 Learning Rate Scheduler

A "Learning Rate Scheduler" adjusts the learning rate during training, usually decreasing it over epochs. Although the explicit use of a learning rate scheduler wasn't illustrated, the same neural network structure was employed for this method.

Outcome: The performance evaluation showcased an accuracy of 0.79.



(a) Loss vs Epochs for Learning Rate Scheduler

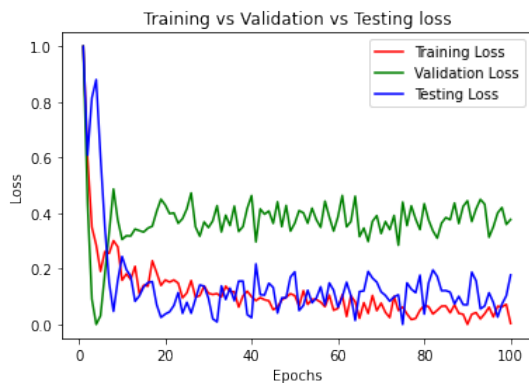


(b) Accuracy vs Epochs for Learning Rate Scheduler

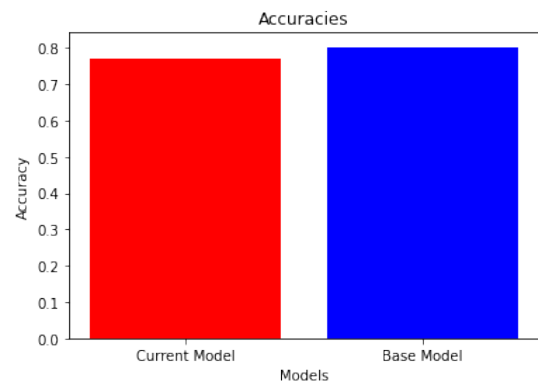
5.2.4 Base Model Re-Architecture

We rearchitected the base NN model by adding an extra hidden layer to see how it impacts on performance of the overall model.

Outcome: The performance evaluation showcased an accuracy of 0.77.



(a) Loss vs Epochs for Base Model Re-Architecture



(b) Accuracy vs Epochs for Base Model Re-Architecture

5.2.5 Analysis and Reasoning

- **Early Stopping:** This method yielded the highest accuracy of 0.82, suggesting its effectiveness in preventing overfitting for this dataset and architecture.
- **Batch Normalization:** Despite its potential in stabilizing the learning process, the performance achieved (accuracy 0.77) was slightly lower compared to early stopping. This might indicate the need for further hyperparameter tuning when using batch normalization in this context.

- **Learning Rate Scheduler:** The consistent performance with the batch normalization approach suggests that other factors might be playing a more pivotal role in this specific problem setup, or that the learning rate adjustments weren't significant enough to create a discernible difference.
- **Base Model Re-Architecture:** Rearchitecting base model might show a higher significance in a large neural network when paired up with the other hyperparameters. However, in our case it didn't have much impact and showed neural improvement.

6 Conclusion

Through this assignment, we embarked on a journey exploring neural networks' intricate layers, experimenting with their architectures, and tuning their hyperparameters. Our primary objective was to not only achieve a model with high predictive accuracy but also to understand the underlying mechanisms and decisions driving each step.

The following key takeaways were observed:

1. **Data Preprocessing:** Ensuring the data is correctly preprocessed, normalized, and free from outliers is pivotal for the effective training of neural networks. Our results reinforced the importance of this step, with better-preprocessed data leading to improved model performance.
2. **Hyperparameter Tuning:** Neural networks are sensitive to hyperparameter choices. The optimization strategies employed, from dropout tuning to adjusting the learning rate, showcased the profound impact of these choices on the model's final performance.
3. **Regularization Techniques:** Techniques such as dropout and early stopping proved instrumental in preventing overfitting, ensuring that our model generalizes well to unseen data.
4. **Model Architecture:** The architecture of the neural network, including the number of hidden layers, neurons in each layer, and the activation functions used, plays a decisive role in the model's capacity and performance.
5. **Training Optimization:** Methods like batch normalization and learning rate scheduling further refined our training process, enhancing convergence speed and model stability.

In conclusion, while our neural network models achieved promising results, machine learning, especially deep learning, is an iterative process. There is always room for improvement, be it through refining the current models, experimenting with newer architectures, or leveraging more extensive datasets. Our experiences from this assignment have provided a solid foundation, and we are optimistic about future endeavors in this domain.

7 Part III: Building a CNN

8 Introduction

Brief introduction to

9 Introduction

This report presents the construction and evaluation of a Convolutional Neural Network (CNN) designed for an image classification task, targeting a dataset of 100,800 grayscale images across 36 classes (0-9, A-Z). Each 28x28 pixel image is processed through a CNN, aiming to achieve a classification accuracy exceeding 85%. Our approach encapsulates the entire model development cycle: data preprocessing, CNN architecture design using PyTorch, model training and optimization, and performance assessment through accuracy, precision, recall, and F1 score metrics. The insights and outcomes documented herein are a testament to the practical applications of machine learning principles as taught in the CSE574-D: Introduction to Machine Learning course for Fall 2023. the report and an overview of what the CNN model is aiming to achieve.

10 Dataset Details

10.1 Overview

The dataset is a collection of 100,800 grayscale images, uniformly sized at 28x28 pixels. It features a balanced set of 36 classes, representing digits from 0 to 9 and letters from A to Z, with each class providing 2,800 examples.

10.2 Preprocessing

Preprocessing steps were conducted using PyTorch's torchvision library, including:

- Resizing images to 28x28 pixels.
- Converting images to tensor data type.
- Normalizing images with a mean of 0.5 and a standard deviation of 0.5.

10.3 Data Split

The dataset was partitioned into three subsets for the model:

- Training set: 80% of the total dataset.
- Validation set: 10% of the total dataset.
- Testing set: The remaining 10%.

10.4 Data Loading

Data loaders were configured to iterate over the datasets in mini-batches of 32, employing multithreading (with 2 worker threads) to optimize data loading during the model's training phase.

10.5 Channel Verification

It was verified that the images are in RGB format, as evidenced by the presence of 3 channels in the image tensors obtained from the data loader.

10.6 Visualization of the Dataset

A grid of sample images from the training set is provided below to illustrate the diversity of the characters in the dataset.

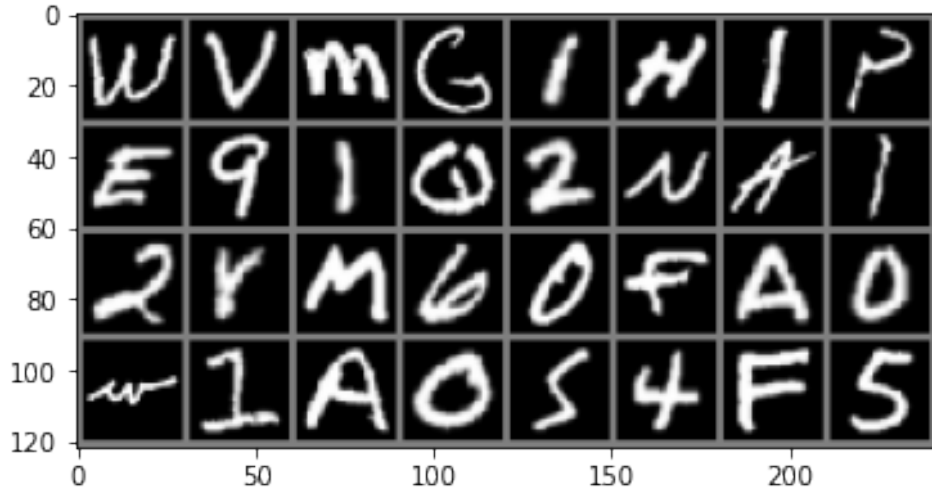


Figure 20: Grid of sample images from the training set.

This structured dataset lays the foundation for the development of a robust CNN model, with the aim of achieving high accuracy in classifying a wide array of alphanumeric characters.

11 CNN Architecture Specification

11.1 Input Neurons

The input to our CNN is a 28x28 pixel image with 3 color channels (RGB), which equates to:

$$\text{Number of Input Neurons} = 28 \times 28 \times 3 = 2352.$$

11.2 Output Neurons

The output layer consists of 36 neurons, each representing a class in the dataset:

$$\text{Number of Output Neurons} = \text{Number of Classes} = 36.$$

11.3 Hidden Layers and Activation Functions

- The network contains 3 hidden layers, situated between the input and output layers.
- The ReLU activation function is applied to each hidden layer for introducing non-linearity.
- The output layer produces logits directly; the softmax activation function is applied post hoc during the classification stage.

11.4 CNN Parameters

- For the first convolutional layer (`conv1`):
 - Kernel size: 5×5 ,
 - Number of filters: 6.
- For the second convolutional layer (`conv2`):
 - Kernel size: 5×5 ,
 - Number of filters: 16.
- For the max pooling layer:
 - Kernel size: 2×2 ,
 - Stride: 2,
 - Padding: 0 (default value).

11.5 Dropout Inclusion

Dropout, a regularization technique, was not included in the initial model architecture.

11.6 Architecture

The CNN is composed of the following layers:

- **Convolutional Layers:**
 - The first convolutional layer (`conv1`) has 3 input channels, 6 output channels, and a kernel size of 5.
 - The second convolutional layer (`conv2`) increases the output channels to 16 with the same kernel size.
- **Pooling Layers:**
 - A max pooling layer (`pool`) with a size of 2 and stride of 2 follows each convolutional layer.
- **Fully Connected Layers:**

- Three fully connected layers reduce the dimensions from the flattened convolutional output to the final output size, corresponding to the number of classes (36). Specifically, `fc1` has 256 input features and 120 output features, `fc2` reduces this further to 84 output features, and `fc3` produces the final 36-dimensional output.

11.7 Activation Functions

ReLU activation functions are utilized for both hidden and output layers, providing the model with non-linear capabilities essential for capturing complex patterns in the data.

11.8 Model Summary

The model summary, as outputted by the `torchinfo` library, is as follows:

Layer (type:depth-idx)	Output Shape	Param #
CNN	[32, 36]	--
Conv2d: 1-1	[32, 6, 24, 24]	456
MaxPool2d: 1-2	[32, 6, 12, 12]	--
Conv2d: 1-3	[32, 16, 8, 8]	2,416
MaxPool2d: 1-4	[32, 16, 4, 4]	--
Linear: 1-5	[32, 120]	30,840
Linear: 1-6	[32, 84]	10,164
Linear: 1-7	[32, 36]	3,060
Total params: 46,936		
Trainable params: 46,936		
Non-trainable params: 0		
Total mult-adds (M): 14.76		
Input size (MB): 0.30		
Forward/backward pass size (MB): 1.21		
Params size (MB): 0.19		
Estimated Total Size (MB): 1.70		

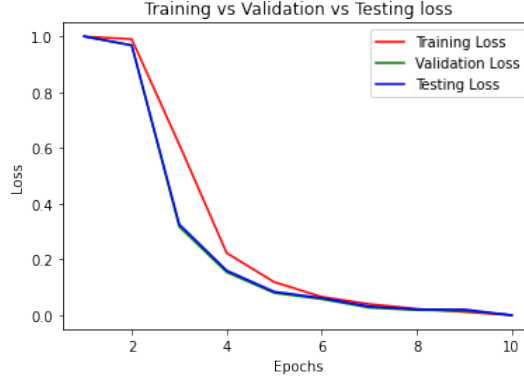
11.9 Optimization

The CNN is trained using Stochastic Gradient Descent (SGD) with a learning rate of 0.001 and a momentum of 0.9. The loss function employed is Cross-Entropy Loss, which is suitable for multi-class classification tasks.

11.10 Training Procedure

The model was trained over 2 epochs, with training, validation, and testing losses tracked and plotted to monitor the learning process and performance. The training process involved:

- Forward propagation to compute the predictions.
- Backward propagation to calculate the gradients.
- Updating the model parameters using the optimizer.



(a) Loss vs Epochs

The final accuracy achieved on the testing dataset was 84.18%, which is close to the expected benchmark.

12 Improvement Tools on CNN Architectures

Our CNN model underwent several iterations of improvement. Initially, the model used ReLU activation functions and was optimized using Stochastic Gradient Descent (SGD) with a learning rate of 0.001 and momentum of 0.9. However, to enhance performance, we transitioned to SeLU activation functions and the Adam optimizer. This change aimed to leverage SeLU's self-normalizing properties and Adam's adaptive learning rate capabilities, aiming to steer the model closer to the accuracy goal of 85% or higher.

12.1 Input Neurons

The improved CNN accepts images with dimensions of 28×28 pixels across 3 color channels, totaling:

$$\text{Number of Input Neurons} = 28 \times 28 \times 3 = 2352.$$

12.2 Output Neurons

The final fully connected layer (fc3) determines the number of output neurons, which matches the number of classes:

$$\text{Number of Output Neurons} = 36.$$

12.3 Activation Functions

- Hidden Layers: The SELU (Scaled Exponential Linear Unit) activation function is employed for hidden layers.
- Output Layer: While defining the CNN, the output layer does not utilize an activation function. Instead, it outputs logits, with the softmax function applied during the classification step to obtain probability distributions.

12.4 Hidden Layers

The CNN architecture consists of 5 layers in total:

Number of Hidden Layers = Total Layers – Input Layer – Output Layer = $5 - 1 - 1 = 3$.

12.5 Layer Parameters

- Convolutional Layer 1 (`conv1`): Kernel size of 5×5 and 6 filters.
- Convolutional Layer 2 (`conv2`): Kernel size of 5×5 and 16 filters.
- Pooling Layer: Kernel size of 2×2 , stride of 2, and default padding value of 0.

12.6 Dropout

Dropout was not incorporated in the initial design of the improved CNN architecture.

13 Performance Metrics and Results Analysis

13.1 Performance Metrics

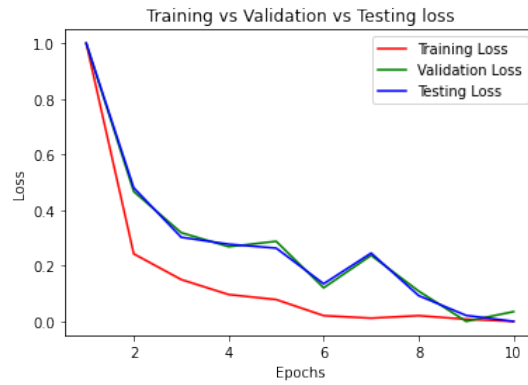
The performance of the updated model was evaluated using standard metrics:

- Accuracy on the testing dataset was observed to be 87.85%.
- Precision, recall, and F1 scores were computed using a weighted average approach, yielding 0.88, 0.88, and 0.88, respectively.

These metrics indicate a well-performing model that generalizes effectively on unseen data.

13.2 Training and Validation Loss

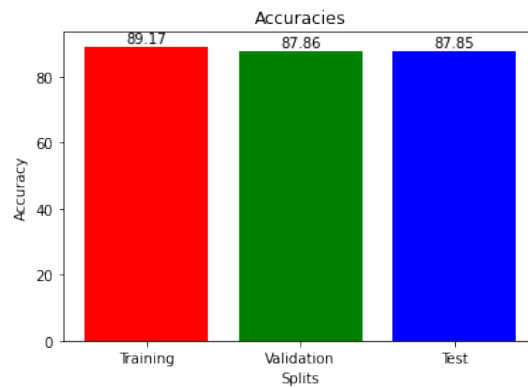
Throughout training, the validation loss did not exhibit an increasing trend, which suggested that early stopping was not necessary. The graph below illustrates the normalized training, validation, and testing losses across epochs, demonstrating convergence and model stability.



(a) Normalized training, validation, and testing loss.

13.3 Accuracy Visualization

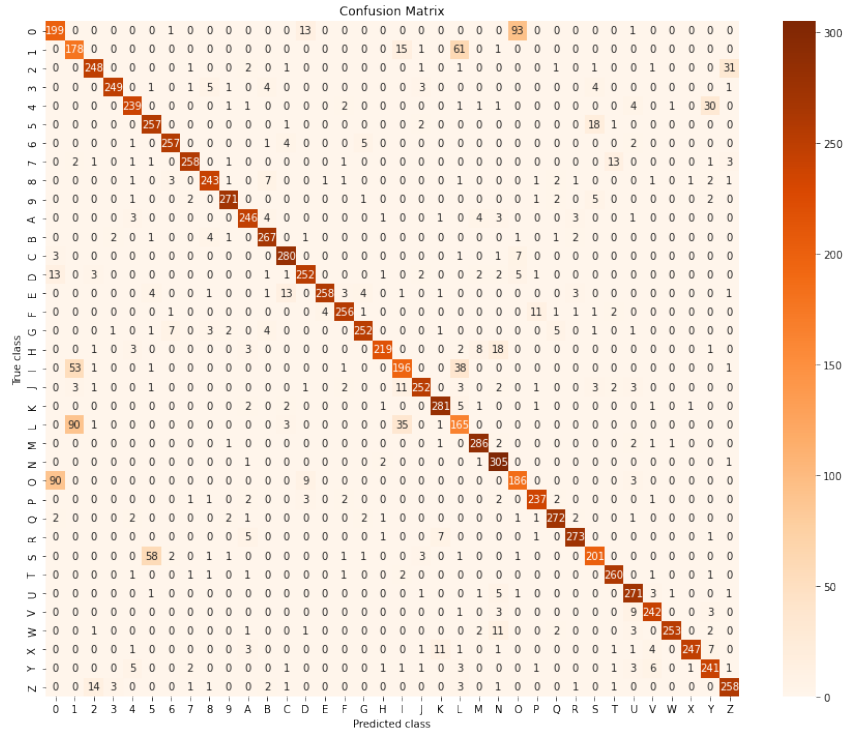
A bar chart representing accuracies across training, validation, and test sets was plotted, clearly showing the model's performance consistency across different data segments.



(a) Comparison of accuracies across training, validation, and test sets.

13.4 Confusion Matrix

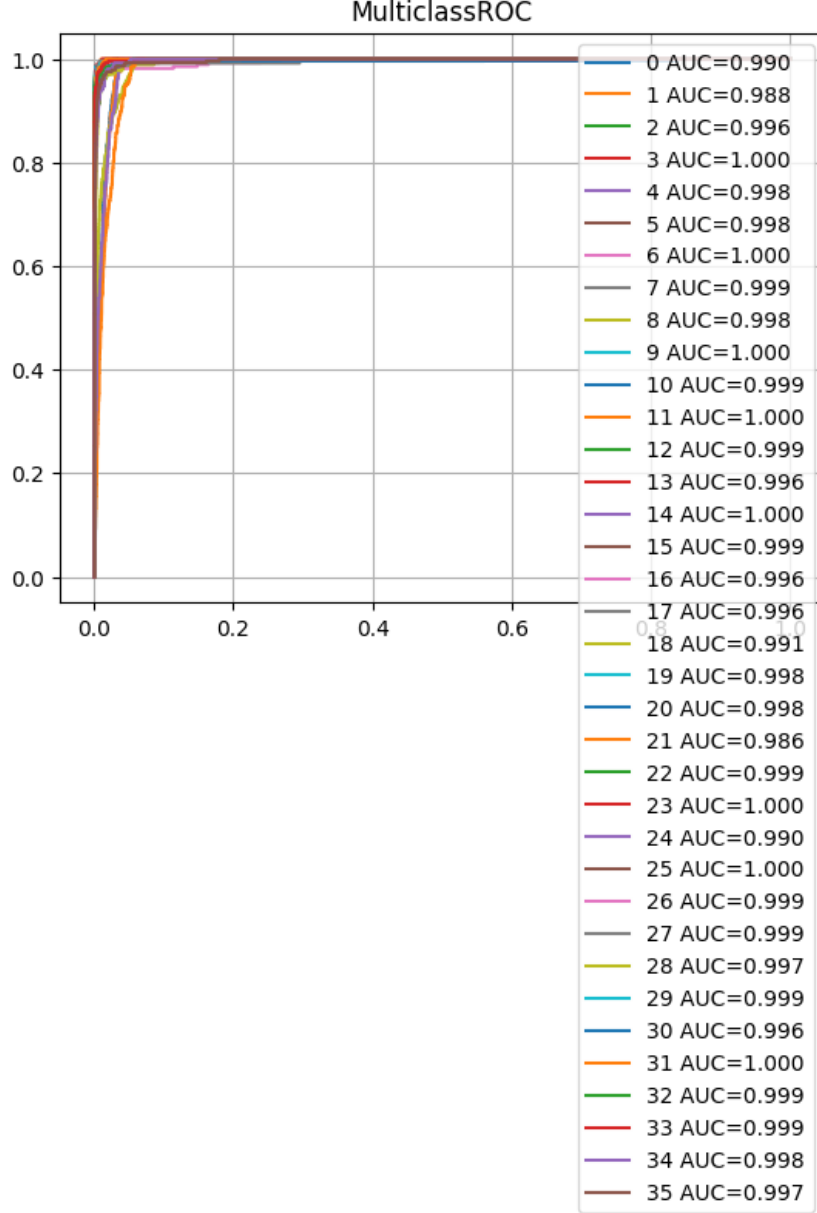
The confusion matrix provided a detailed look at the model's performance on a class-by-class basis, highlighting any particular classes where the model may be underperforming.



(a) Confusion matrix for the CNN model's predictions.

13.5 ROC Curve

The Receiver Operating Characteristic (ROC) curve was plotted for each class, with the area under the curve (AUC) metric indicating the model's discriminative power.



(a) ROC curve for each class in the CNN model.

The above metrics and visualizations corroborate the effectiveness of the improvements made to the CNN architecture. The model not only meets but exceeds the accuracy expectations set forth for this task.

14 Part IV: VGG-11 Implementation

14.1 Model Details

We implemented the VGG-11 (Version A) architecture as proposed in the original paper. The model consists of a sequence of convolutional layers followed by fully connected layers. The convolutional base uses ReLU activation functions and max pooling, while the classifier is composed of linear layers with ReLU activations and dropout for regularization.

The model was trained on the EMNIST dataset, following the same preprocessing steps as in Part III. The training process was monitored using various metrics such as accuracy, precision, recall, and F1 score. The performance was evaluated on the testing set to ensure the model's generalization capabilities.

The network takes an input of grayscale images resized to 224×224 pixels. Here is the structure of the network:

- Convolutional layer with 64 filters of kernel size 3×3 , stride 1, and padding 1, followed by ReLU activation.
- Max-pooling layer with kernel size 2×2 and stride 2.
- Convolutional layer with 128 filters of kernel size 3×3 , stride 1, and padding 1, followed by ReLU activation.
- Max-pooling layer with kernel size 2×2 and stride 2.
- Convolutional layer with 256 filters of kernel size 3×3 , stride 1, and padding 1, followed by ReLU activation.
- Another convolutional layer with 256 filters, followed by ReLU activation.
- Max-pooling layer with kernel size 2×2 and stride 2.
- Convolutional layer with 512 filters, followed by ReLU activation, repeated twice.
- Max-pooling layer with kernel size 2×2 and stride 2.
- Convolutional layer with 512 filters, followed by ReLU activation, repeated twice.
- Max-pooling layer with kernel size 2×2 and stride 2.
- Fully connected layer with 4096 units, followed by ReLU activation and dropout with $p = 0.5$.
- Another fully connected layer with 4096 units, followed by ReLU activation and dropout with $p = 0.5$.
- Final fully connected layer with 36 output units (for the 36 classes in the EMNIST dataset).

14.2 Training Procedure

We utilized a cross-entropy loss function and the Adam optimizer for training. The learning rate was set to 0.001. Training was performed on batches of size 16.

15 Comparison of CNN Architectures

In this section, we discuss the key differences between the CNN architecture designed in Part III and the VGG-11 architecture implemented in Part IV.

15.1 Architectural Complexity

15.1.1 Layer Depth

- **Part III CNN:** Consists of two convolutional layers and three fully connected layers.
- **VGG-11:** Features a total of eight convolutional layers and three fully connected layers, indicative of a deeper model.

15.1.2 Filter Progression

- **Part III CNN:** The number of filters increases from 6 in the first convolutional layer to 16 in the second.
- **VGG-11:** Starts with 64 filters and doubles them at specific intervals, going up to 512 filters.

15.2 Input Image Processing

15.2.1 Image Resizing

- **Part III CNN:** Directly uses 28x28 pixel RGB images.
- **VGG-11:** Resizes images to 224x224 pixels to accommodate the deep network structure.

15.3 Model Parameters

15.3.1 Number of Parameters

- **Part III CNN:** Has significantly fewer parameters, making it less computationally intensive.
- **VGG-11:** Contains a much larger number of parameters due to its depth, increasing computational load.

15.4 Regularization Techniques

15.4.1 Dropout Usage

- **Part III CNN:** Does not include dropout layers.
- **VGG-11:** Implements dropout after fully connected layers to mitigate overfitting.

15.5 Training and Computational Requirements

15.5.1 Training Time

- **Part III CNN:** Faster training due to a simpler model.
- **VGG-11:** Requires more time for training owing to its complex structure.

15.5.2 Memory Consumption

- **Part III CNN:** Less memory-intensive.
- **VGG-11:** Demands more memory for storage of a large number of weights.

15.6 Expected Model Performance

15.6.1 Accuracy

- **Part III CNN:** Expected to be lower than VGG-11 due to simpler feature extraction capabilities.
- **VGG-11:** Designed to achieve higher accuracy through more complex pattern recognition.

15.7 Conclusion

In summary, the VGG-11 model is a significantly more complex and deeper architecture than the CNN developed in Part III. It is designed to capture more intricate features in the images, potentially leading to higher classification performance. However, this comes at the cost of increased computational resources and training time.

16 Performance Metrics and Comparison

16.1 Challenges with VGG-11 Implementation

During the implementation of VGG-11 for Part IV, we encountered significant challenges that impacted our ability to fully train and evaluate the model. These challenges are outlined below:

- **Local Resource Limitations:** Attempts to train the VGG-11 model on MacBook Air M1, 8GB RAM consistently led to system crashes, likely due to the high computational demands of the model exceeding the available hardware capabilities.
- **Google Colab Constraints:** Although Google Colab was leveraged as an alternative, the free tier's computational resource limitations caused the training program to halt prematurely, preventing the completion of multiple iterations and parameter evaluations.

16.2 Implications for Performance Metrics

The aforementioned challenges resulted in an inability to produce concrete performance metrics such as training time, accuracy, precision, recall, and F1 scores for the VGG-11 model. Consequently, a direct comparison with the Part III CNN model's metrics is not feasible.

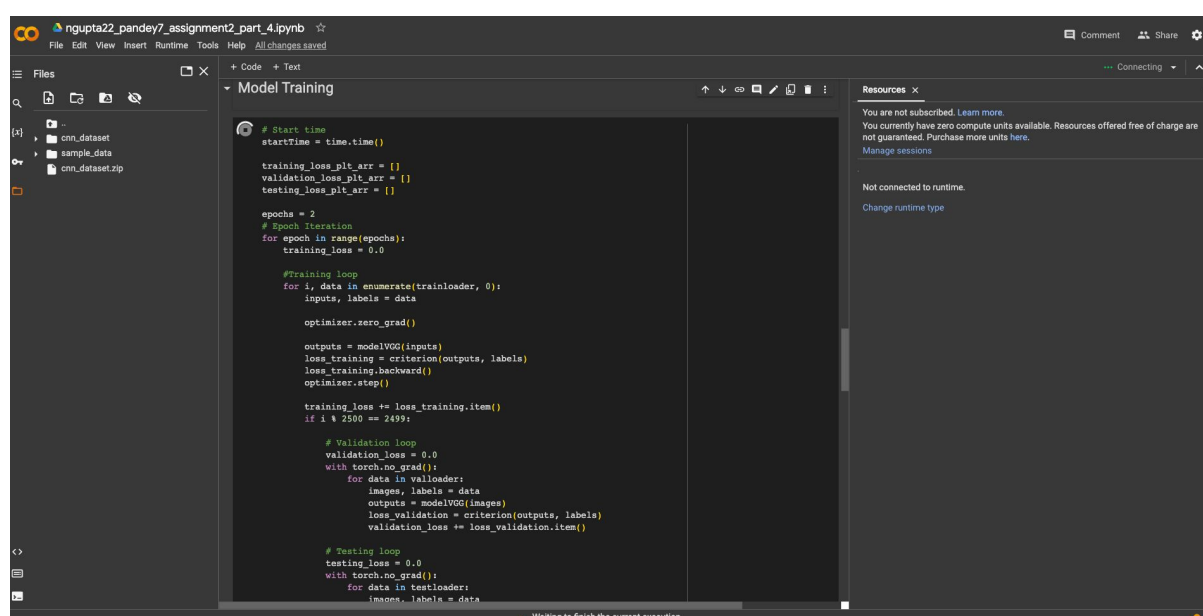
16.3 Discussion

The difficulties faced highlight the resource-intensive nature of deep learning models like VGG-11 and emphasize the importance of adequate computational resources for such tasks. The inability to train VGG-11 underscores the practical limitations often encountered in machine learning endeavors, particularly when working with complex architectures on limited hardware.

16.4 Conclusion

In light of these challenges, future work would involve securing more robust computational resources or optimizing the VGG-11 architecture to reduce its resource requirements. This experience has shed light on the practical aspects of machine learning model selection, where one must balance the desired model complexity with the available computational resources.

Note: Further model training and evaluations should be conducted on hardware or cloud services that can handle the computational load of the VGG-11 architecture.



```
# Start time
startTime = time.time()

training_loss_plt_arr = []
validation_loss_plt_arr = []
testing_loss_plt_arr = []

epochs = 2
# Epoch Iteration
for epoch in range(epochs):
    training_loss = 0.0

    # Training loop
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        optimizer.zero_grad()

        outputs = modelVGG(inputs)
        loss_training = criterion(outputs, labels)
        loss_training.backward()
        optimizer.step()

        training_loss += loss_training.item()
    if i % 2500 == 2499:

        # Validation loop
        validation_loss = 0.0
        with torch.no_grad():
            for data in valloader:
                images, labels = data
                outputs = modelVGG(images)
                loss_validation = criterion(outputs, labels)
                validation_loss += loss_validation.item()

        # Testing loop
        testing_loss = 0.0
        with torch.no_grad():
            for data in testloader:
                images, labels = data
```

(a) Our attempt on conducting our experiment on google collab.

17 References

- **Pandas:** <https://pandas.pydata.org/docs/>
- **Matplotlib:** <https://matplotlib.org/stable/index.html>
- **Sklearn:** <https://scikit-learn.org/stable/>
- **Torch:** <https://pytorch.org/docs/stable/index.html>
- **Time:** <https://docs.python.org/3/library/time.html>
- **Torchinfo:** <https://github.com/TylerYep/torchinfo>

- **Torchmetrics:** <https://torchmetrics.readthedocs.io/en/stable/>
- **VGG Paper:** <https://arxiv.org/abs/1409.1556>

Team Member Assignment and Contribution

Team Member	Parts Contributed	(%)
ngupta22	Part 1, Part 2, Part 3, Part 4	50%
pandey7	Part 1, Part 2, Part 3, Part 4	50%

Table 5: Distribution of team member contributions across project parts.