CS6910 : Fundamentals of Deep Learning

Assignment # 1

Visveshwar Jay Shankar, EE22B154
Pranay Mathur, EE22B132
Sriram Makkena, CS22B014

March 2, 2025

Contents

List of Figures

1 Task 1 : Comparison of Optimization Methods for an Image Classification Task

1.1 Introduction

- In any deep learning model, the associated optimization algorithm is a significant bottleneck, and greatly influences the speed of convergence as well as the final accuracy of the model.
- With increasingly complex error surfaces, algorithms of higher complexity are required to reach global minima and result in the best possible model.
- With models becoming more complex, with wider range of applications and real world demand, it is imperative to put significant amount of effort into comparison of different optimization methods to compare convergence power.

1.2 Dataset Description

The given task involves analysis of different update mechanisms for the given dataset. The dataset is summarized as follows:

- The train dataset contains 2000 feature vectors, each of length 36.
- As seen below, these features are extracted from a pre-existing dataset and given to us, and the features are not humanely interpretable.

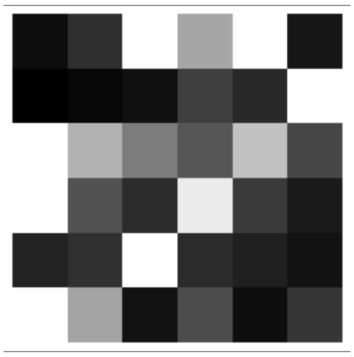


Figure 1: Input image

- There are 5 class labels, labeled as 0, 1, 2, 3, 4, with 400 examples of each. Thus, the dataset is well balanced.
- The test dataset is also well balanced, with 100 examples of each class.
- Each pixel is assigned a `float64` value between 0 and 1.

1.3 Approach

- The `pytorch` library was used for this assignment.
- A neural network was created using the given parameters, with input layer of size 36, hidden layer 1 of size 20, hidden layer 2 of size 10 and output layer of size 5.
- Learning rate was set to `0.01`, and the stopping criterion was taken to be the absolute change in average error being less than `0.0001`.

- Additionally, a max epochs cap of `1000` was kept in case the model was taking too long to converge.
- Momentum factor was kept at `0.1`, and tanh activation was used.
- Since pattern mode depends on order of inputs being fed in, they were shuffled randomly.
- Otherwise, if trained the way presented in the dataset, the model would first overfit on class 1, then on class 2, and so on, as can be seen below, where error from sample to sample within an epoch is shown:

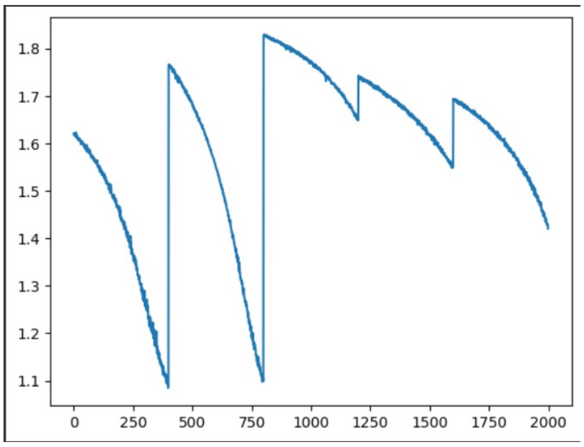


Figure 2: Loss variation within epoch when inputs not shuffled

- A neural net was initialized, it's weights were saved, and the same initial values were loaded for all the 5 cases.

1.4 Results

1.4.1 Delta Rule

- The delta rule took `55` epochs to converge, and resulted in a loss of `1.1702`.
- The loss varied as follows:

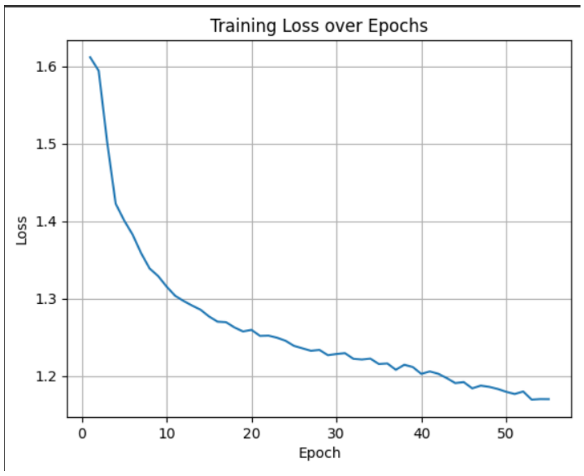
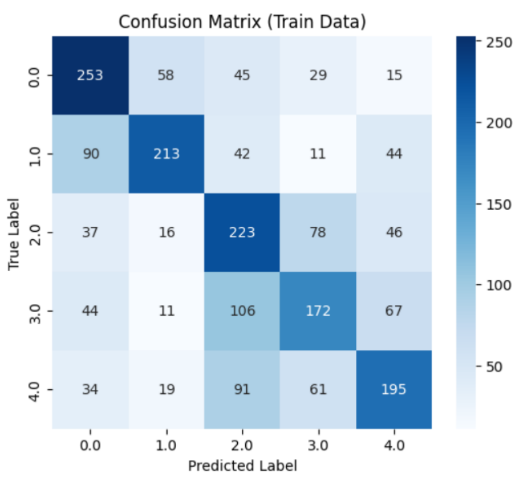


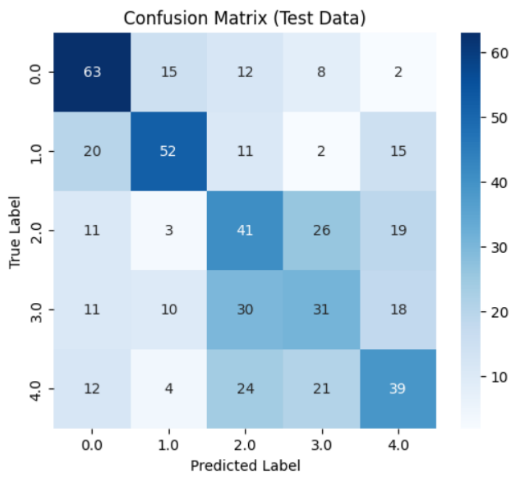
Figure 3: Delta rule loss

- The accuracy on training data was `52.8%`, and on testing data was `45.2%`.
- The confusion matrices are attached below for train and test datasets:



	0.0	1.0	2.0	3.0	4.0
0.0	253	58	45	29	15
1.0	90	213	42	11	44
2.0	37	16	223	78	46
3.0	44	11	106	172	67
4.0	34	19	91	61	195

(a) Train Data Confusion Matrix



	0.0	1.0	2.0	3.0	4.0
0.0	63	15	12	8	2
1.0	20	52	11	2	15
2.0	11	3	41	26	19
3.0	11	10	30	31	18
4.0	12	4	24	21	39

(b) Test Data Confusion Matrix

Figure 4: Confusion Matrices

1.4.2 Generalized Delta Rule

- The generalized delta rule took `146` epochs to converge, and resulted in a loss of `1.0425`.
- The loss varied as follows:

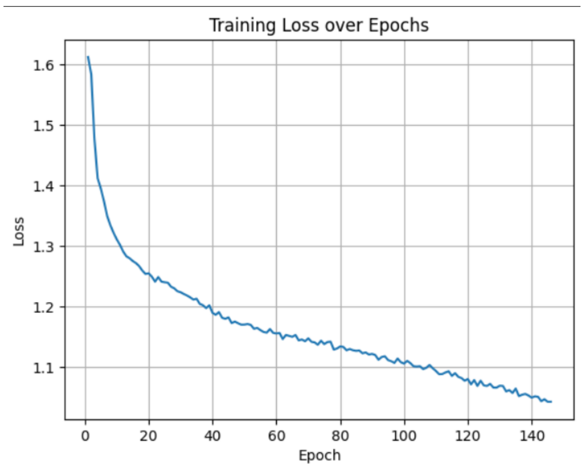
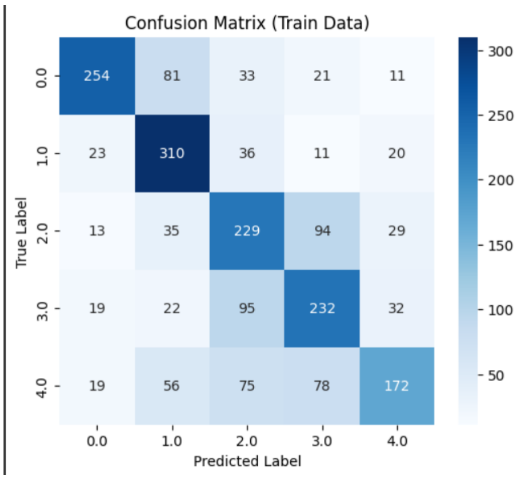
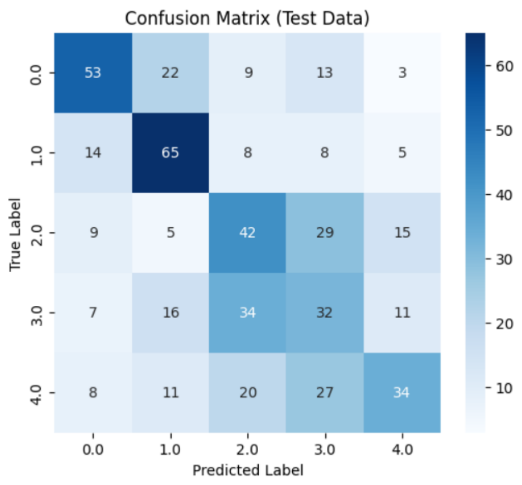


Figure 5: Generalized delta rule loss

- The accuracy on training data was `59.85%`, and on testing data was `45.2%`.
- The confusion matrices are attached below for train and test datasets:



(a) Train Data Confusion Matrix



(b) Test Data Confusion Matrix

Figure 6: Confusion Matrices

1.4.3 AdaGrad

- The AdaGrad rule took `79` epochs to converge, and resulted in a loss of `1.2763`.
- The loss varied as follows:

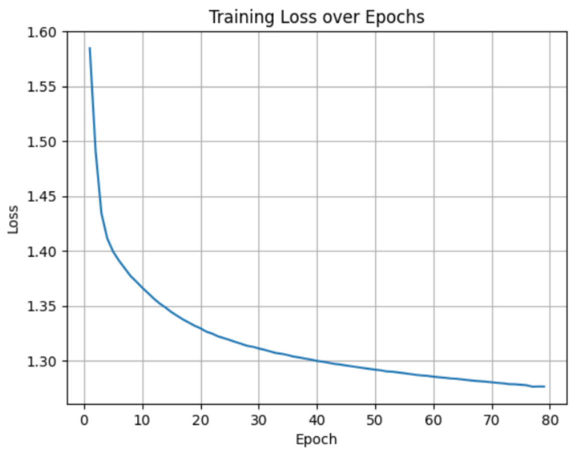
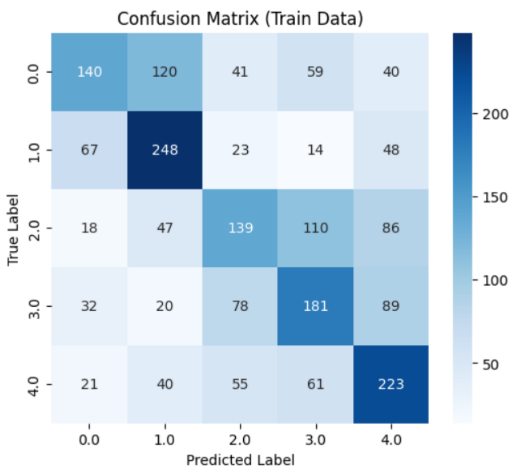
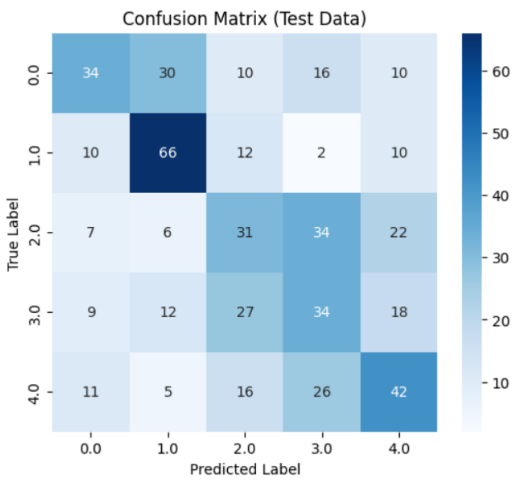


Figure 7: AdaGrad Loss

- The accuracy on training data was `46.55%`, and on testing data was `41.40%`.
- The confusion matrices are attached below for train and test datasets:



(a) Train Data Confusion Matrix



(b) Test Data Confusion Matrix

Figure 8: Confusion Matrices

1.4.4 RMSProp

- The RMSProp rule took `176` epochs to converge, and resulted in a loss of `0.9620`.
- The loss varied as follows:

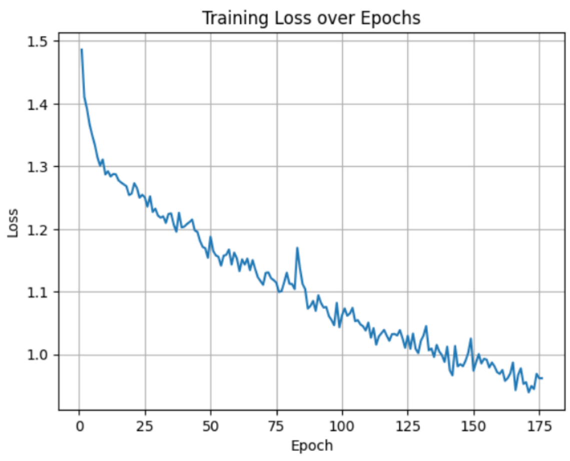
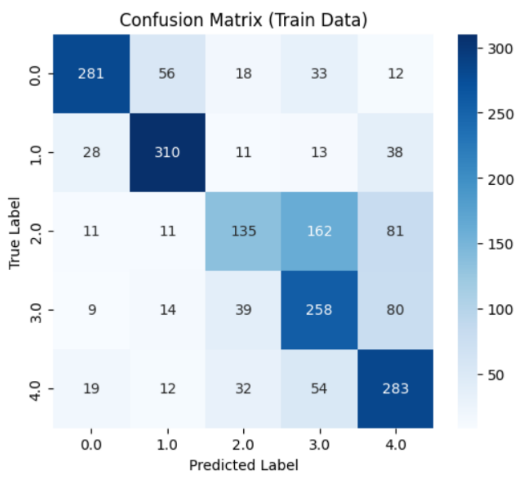


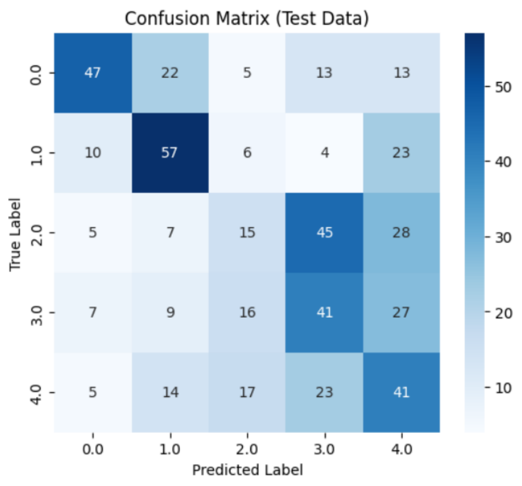
Figure 9: RMSProp Loss

- The accuracy on training data was `63.35%`, and on testing data was `40.20%`.
- The confusion matrices are attached below for train and test datasets:



	0.0	1.0	2.0	3.0	4.0
0.0	281	56	18	33	12
1.0	28	310	11	13	38
2.0	11	11	135	162	81
3.0	9	14	39	258	80
4.0	19	12	32	54	283

(a) Train Data Confusion Matrix



	0.0	1.0	2.0	3.0	4.0
0.0	47	22	5	13	13
1.0	10	57	6	4	23
2.0	5	7	15	45	28
3.0	7	9	16	41	27
4.0	5	14	17	23	41

(b) Test Data Confusion Matrix

Figure 10: Confusion Matrices

1.4.5 AdaM

- The AdaM rule took `64` epochs to converge, and resulted in a loss of `1.2098`.
- The loss varied as follows:

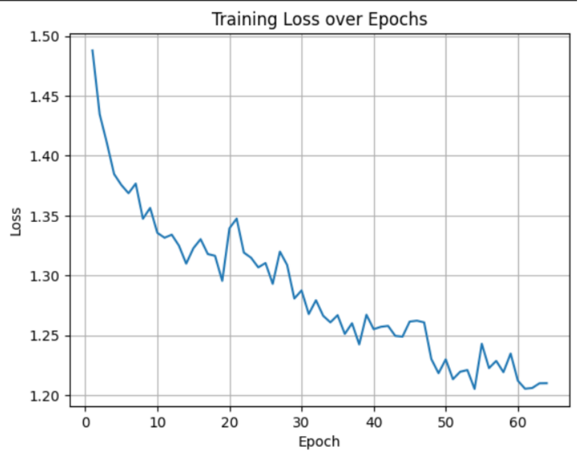
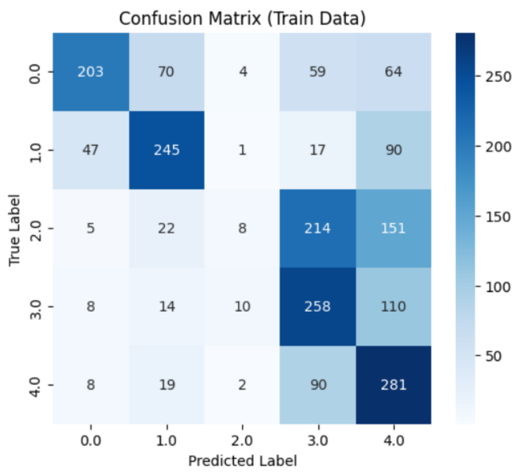
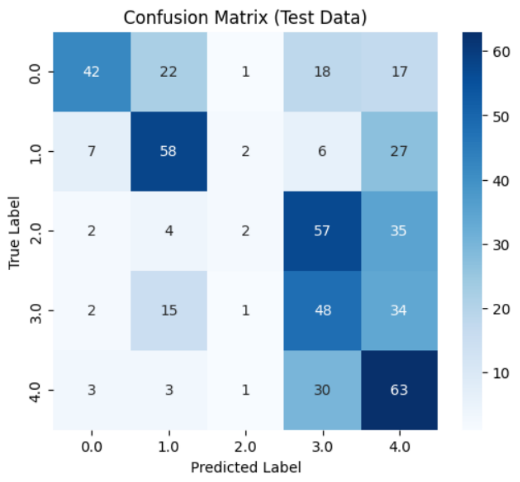


Figure 11: AdaM loss

- The accuracy on training data was `49.75`, and on testing data was `42.60`.
- The confusion matrices are attached below for train and test datasets:



(a) Train Data Confusion Matrix

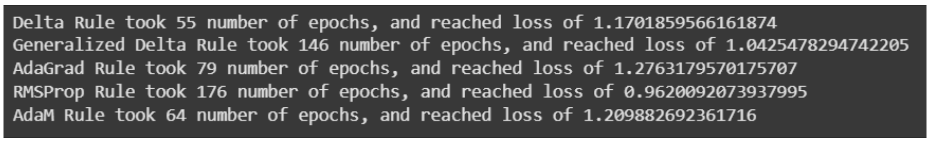


(b) Test Data Confusion Matrix

Figure 12: Confusion Matrices

1.5 Inferences

- The loss and epochs are summarized as follows:



```
Delta Rule took 55 number of epochs, and reached loss of 1.1701859566161874
Generalized Delta Rule took 146 number of epochs, and reached loss of 1.0425478294742205
AdaGrad Rule took 79 number of epochs, and reached loss of 1.2763179570175707
RMSProp Rule took 176 number of epochs, and reached loss of 0.9620092073937995
AdaM Rule took 64 number of epochs, and reached loss of 1.209882692361716
```

Figure 13: Summary

- All the losses can be plotted together as follows:

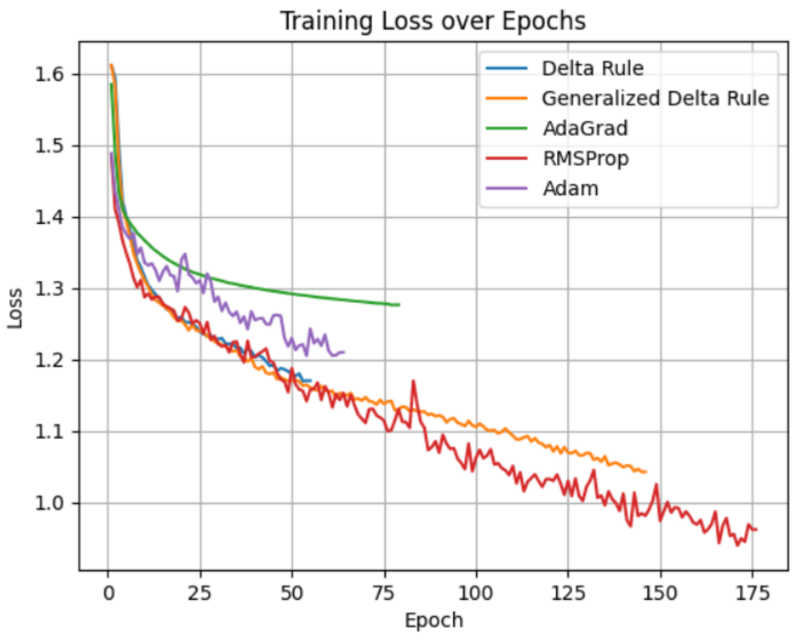


Figure 14: All losses

- We can see that here RMSProp has converged to a lower value, but has taken more epochs and has overfit, as seen from the testing accuracy values.
- No optimization method is a clear winner, as each one is very sensitive to hyperparameters such as stopping criteria, and vary in convergence, time of convergence, and test accuracy.
- Although AdaM is widely known to be one of the best optimization methods out there, it has not significantly outperformed the others in this experiment. High sensitivity to hyperparameters could be an issue.
- It was also noticed that running the experiment a number of times resulted in highly different accuracies, leading us to believe that the models were very sensitive to initial weights.
- To conclude, a comparative study of various optimizers is a very involved one, as there is huge reliance on a variety of hyperparameters.

2 Task 2 : Comparison of Normalization Methods for an Image Classification Task

2.1 Introduction

Deep learning models as great as they are they do come with their own set of challenges. Some of these include:

- **Disparity in the scales of features :** When you have many input features, across different ranges the features
- **Internal Covariant Shift :** The input layer distribution is constantly changing due to weight updates, the following layer always needs to adapt to the new distribution. It causes slower convergence and unstable training.
- **Faster Convergence:** When input features are not properly scaled, the error surface becomes distorted, making it harder to take steps along the gradient and converge efficiently.

Normalization of layers helps address these issues while also providing a degree of regularization, leading to more stable and efficient training.
Our task here was to compare how different normalization layers affect the performance of an image classifying model. We have been given an image data set with certain features already extracted. We were provided with three datasets:

- Training dataset : Containing 2000 image feature vectors, each of dimension = 36
- Test dataset : Containing 500 image feature vectors, each of dimension = 36
- Validation dataset : Containing 500 image feature vectors, each of dimension = 36

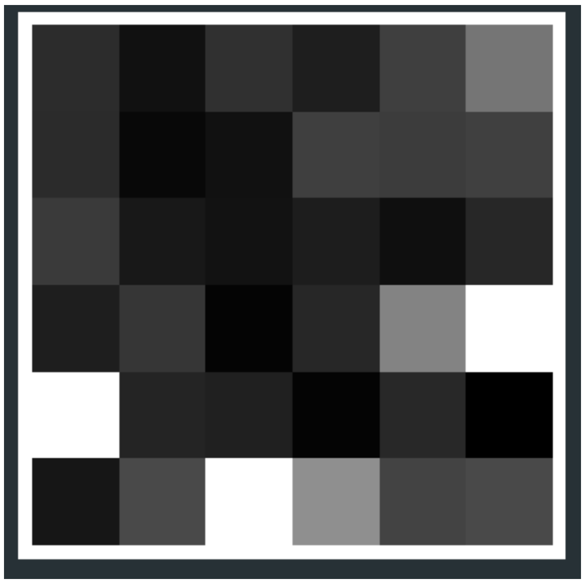


Figure 15: Example Image from the dataset

We had to compare the performances of a model trained with:

- No normalization
- Batch Normalization
- Layer Normalization

2.2 Approach

- The pytorch library was used for performing this task as well.
- A neural network was created using the given parameters, with input layer of size 36, hidden layer 1 of size 20, hidden layer 2 of size 10 and output layer of size 5.
- The model was trained with different hyperparameters (learning rate, error threshold) to evaluate which set of them gets you the most optimum performance.
- Additionally, a max epochs cap of 1000 was kept in case the model was taking too long to converge.

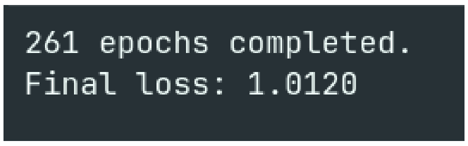
2.3 Results

Attached below are the results after training the model with different values of hyperparameters taken, and its performance on the training, test and validation datasets.

2.3.1 No Normalization

Learning Rate $=$ **0.001 , Stopping Criteria** $=$ **Δerror** $< 10^{-5}$

- Without any normalization, the training process took 261 epochs to converge.



```
261 epochs completed.
Final loss: 1.0120
```

- The loss varied as follows:

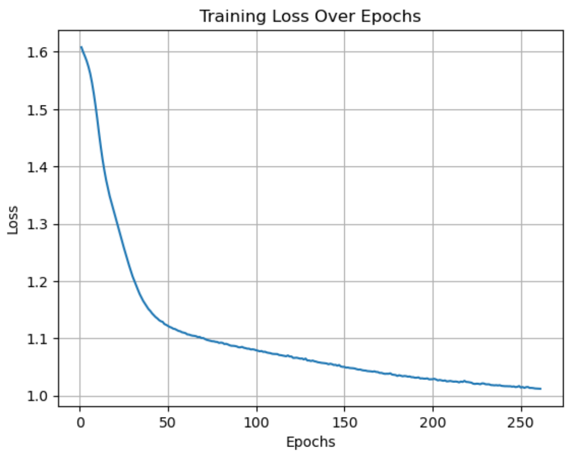
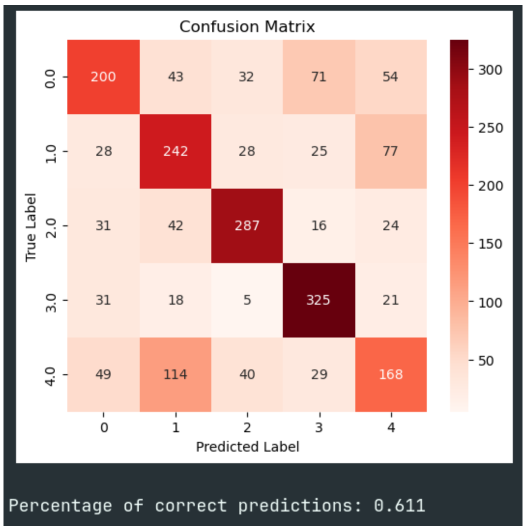


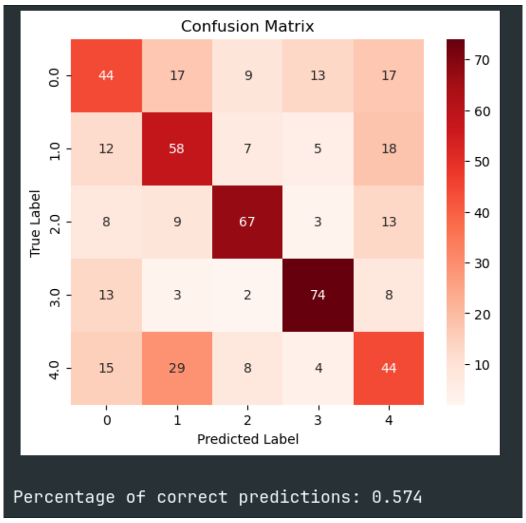
Figure 16: Loss curve without normalization

- The model had a **61.1%** accuracy on the training data and a **57.4%** accuracy on the training data.
- The confusion matrices for the test and training data are attached below



```
Percentage of correct predictions: 0.611
```

(a) Train Data Confusion Matrix



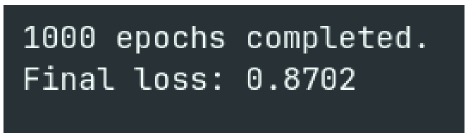
```
Percentage of correct predictions: 0.574
```

(b) Test Data Confusion Matrix

Figure 17: Confusion Matrices

Learning Rate = 0.001 , Stopping Criteria = Δerror $< 10^{-6}$, max_epochs = 1000

- With no normalization, after making the change threshold smaller, the training process automatically stopped after hitting the maximum number of epochs limit.



```
1000 epochs completed.
Final loss: 0.8702
```

- The loss varied as follows:

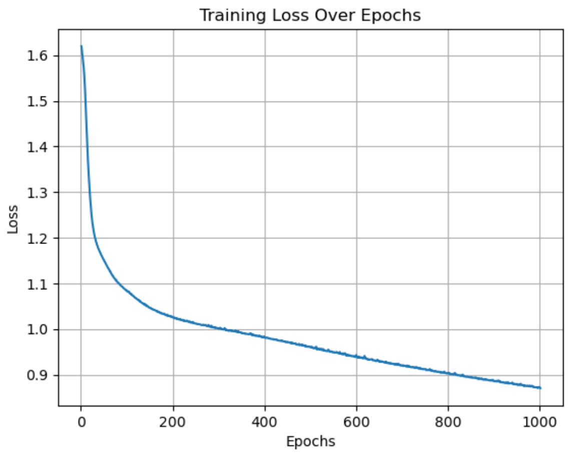
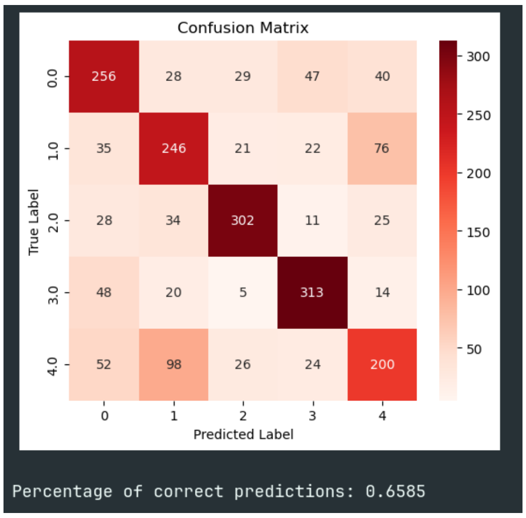


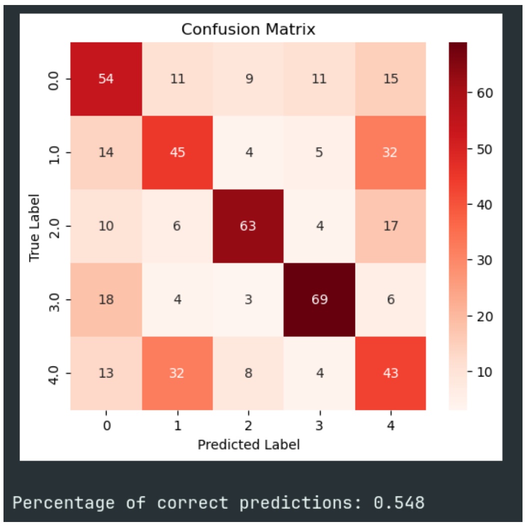
Figure 18: Loss curve without normalization

- The model had a **65.85%** accuracy on the training data and a **54.8%** accuracy on the training data.
- The confusion matrices for the test and training data are attached below



```
Percentage of correct predictions: 0.6585
```

(a) Train Data Confusion Matrix



```
Percentage of correct predictions: 0.548
```

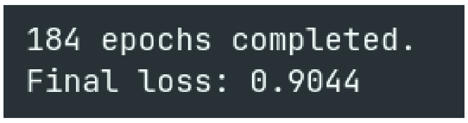
(b) Test Data Confusion Matrix

Figure 19: Confusion Matrices

2.3.2 Batch Normalization

Learning Rate $= 0.001$ **, Stopping Criteria** $= \Delta$**error** $< 10^{-5}$

- With Batch normalization, the training process took 184 epochs to converge.



```
184 epochs completed.
Final loss: 0.9044
```

- The loss varied as follows:

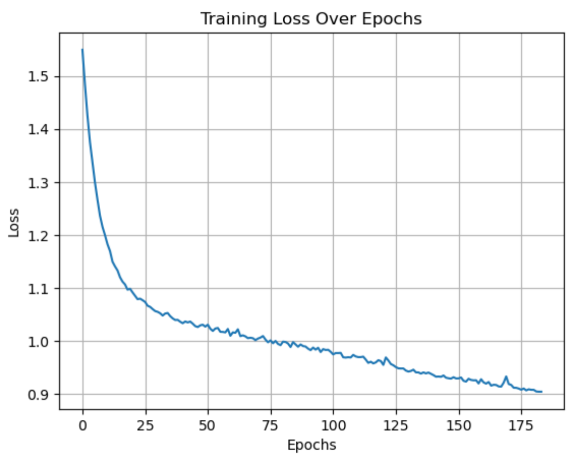
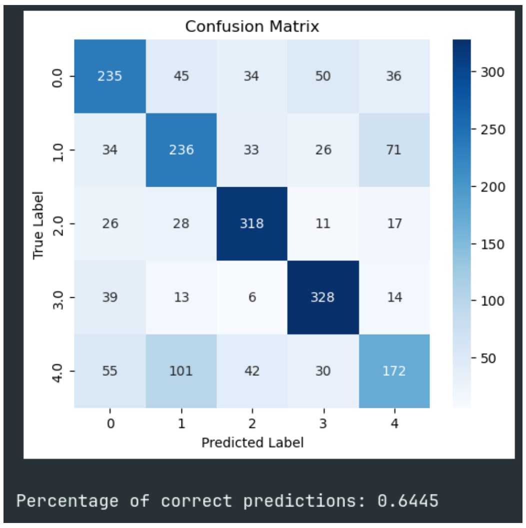


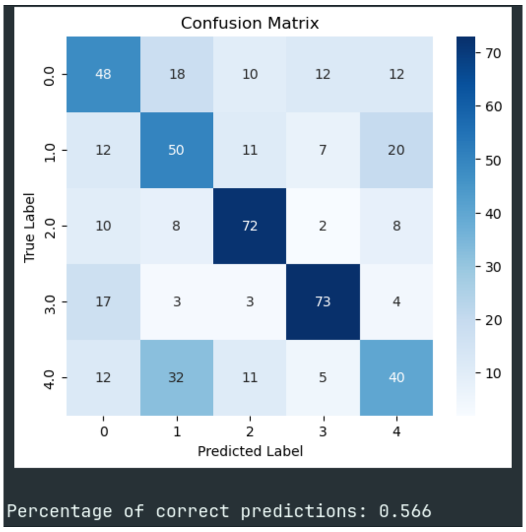
Figure 20: Loss curve with batch normalization

- The model had a **64.45%** accuracy on the training data and a **56.6%** accuracy on the training data.
- The confusion matrices for the test and training data are attached below



```
Percentage of correct predictions: 0.6445
```

(a) Train Data Confusion Matrix



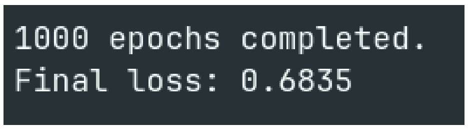
```
Percentage of correct predictions: 0.566
```

(b) Test Data Confusion Matrix

Figure 21: Confusion Matrices

Learning Rate = **0.001** , **Stopping Criteria** = Δ**error** $< 10^{-6}$, **max epochs** =**1000**

- With batch normalization as well, after making the change threshold smaller, the training process automatically stopped after hitting the maximum number of epochs limit.



```
1000 epochs completed.
Final loss: 0.6835
```

- The loss varied as follows:

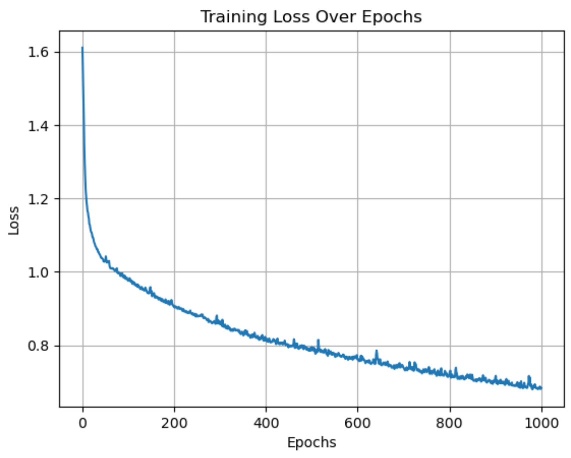
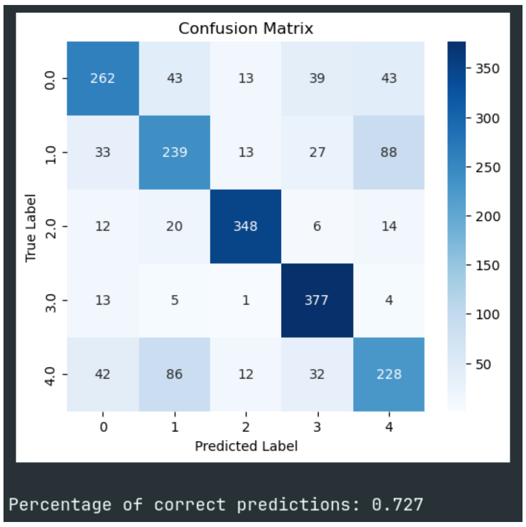


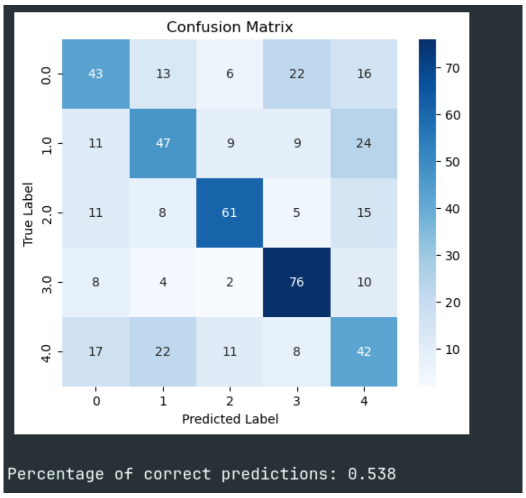
Figure 22: Loss curve with batch normalization

- The model had a **72.7%** accuracy on the training data and a **53.8%** accuracy on the training data.
- The confusion matrices for the test and training data are attached below



```
Percentage of correct predictions: 0.727
```

(a) Train Data Confusion Matrix



```
Percentage of correct predictions: 0.538
```

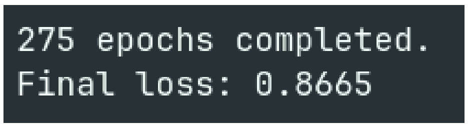
(b) Test Data Confusion Matrix

Figure 23: Confusion Matrices

2.3.3 Layer Normalization

Learning Rate $=$ **0.001 , Stopping Criteria** Δ**error** $< 10^{-5}$

- With Layer normalization, the training process took 275 epochs to converge.



```
275 epochs completed.
Final loss: 0.8665
```

- The loss varied as follows:

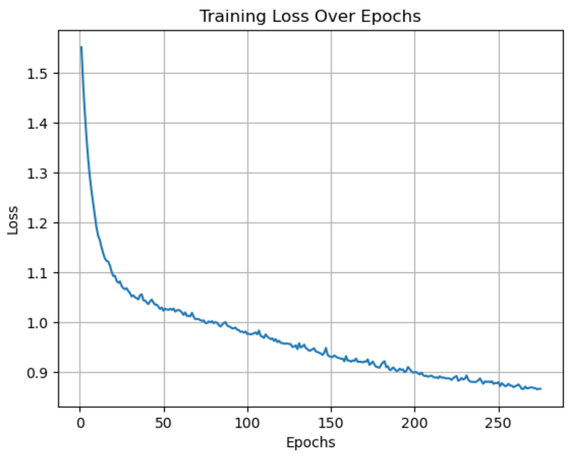
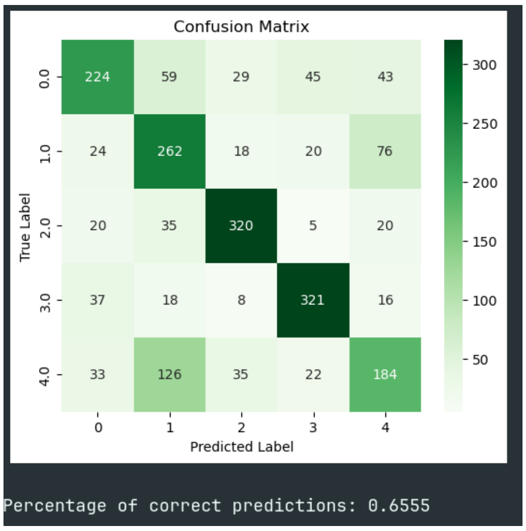


Figure 24: Loss curve with normalization

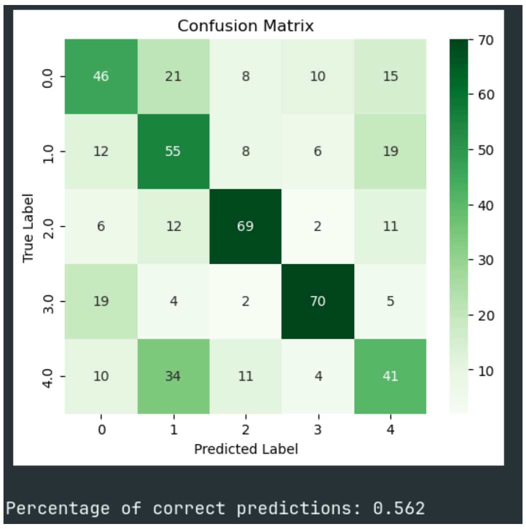
- The model had a **65.55%** accuracy on the training data and a **56.2%** accuracy on the training data.
- The confusion matrices for the test and training data are attached below



	0	1	2	3	4
0.0	224	59	29	45	43
1.0	24	262	18	20	76
2.0	20	35	320	5	20
3.0	37	18	8	321	16
4.0	33	126	35	22	184

```
Percentage of correct predictions: 0.6555
```

(a) Train Data Confusion Matrix



	0	1	2	3	4
0.0	46	21	8	10	15
1.0	12	55	8	6	19
2.0	6	12	69	2	11
3.0	19	4	2	70	5
4.0	10	34	11	4	41

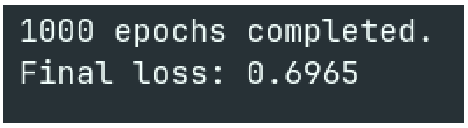
```
Percentage of correct predictions: 0.562
```

(b) Test Data Confusion Matrix

Figure 25: Confusion Matrices

Learning Rate $=$ **0.001** , **Stopping Criteria** $=$ **Δerror** $< 10^{-6}$,**max epochs** $=$ **1000**

- Just like the other two methods, with layer normalization as well,upon making the change threshold smaller, the training process automatically stopped after hitting the maximum number of epochs limit.



```
1000 epochs completed.
Final loss: 0.6965
```

- The loss varied as follows:

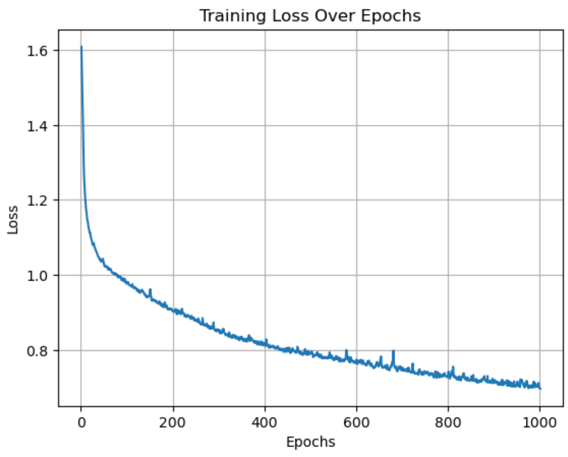
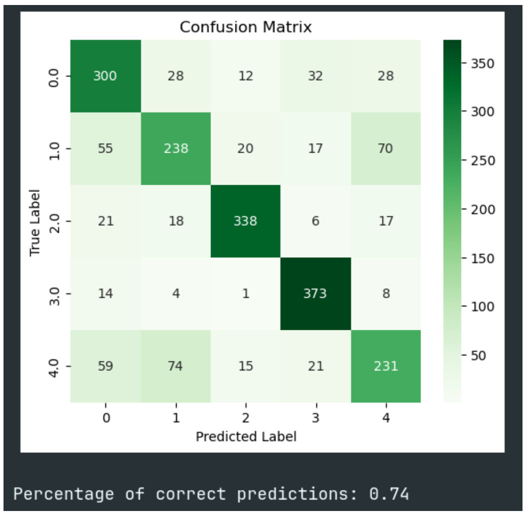


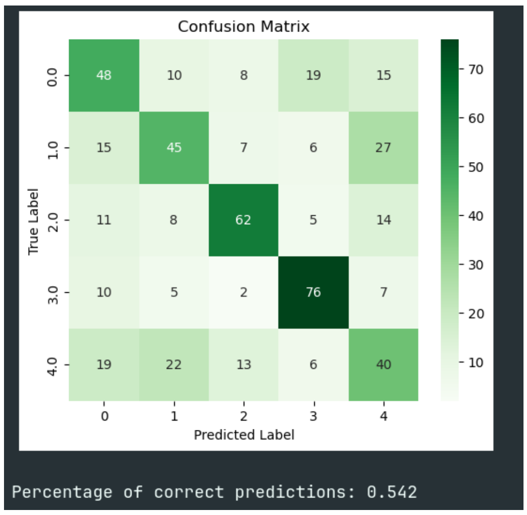
Figure 26: Loss curve with normalization

- The model had a **74%** accuracy on the training data and a **54.2%** accuracy on the training data.
- The confusion matrices for the test and training data are attached below



```
Percentage of correct predictions: 0.74
```

(a) Train Data Confusion Matrix



```
Percentage of correct predictions: 0.542
```

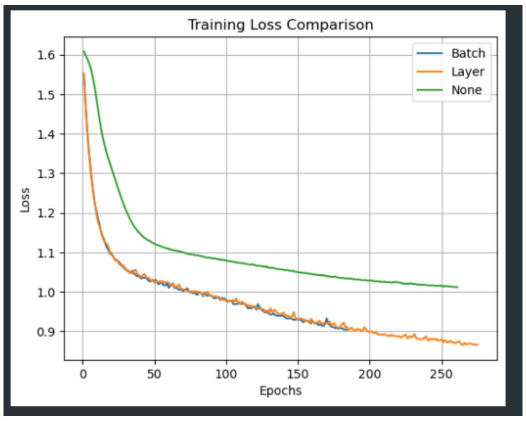
(b) Test Data Confusion Matrix

Figure 27: Confusion Matrices

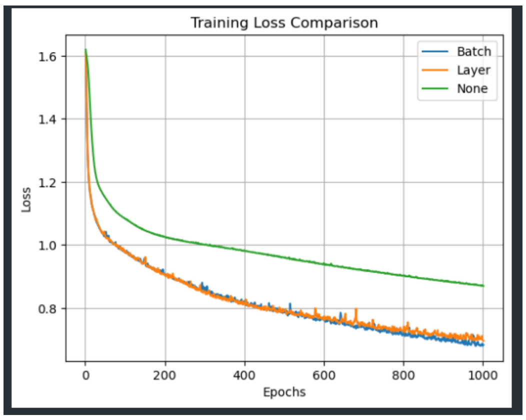
2.4 Inferences

- One thing we observed during the training of the models is that there was a decent amount of variation in the results of the model. The results appended above are the ones we were the most satisfied with, the one where the model preformed decently well on both the training and test data.
- For the same learning rate and error threshold there were times when the training accuracy went as high as 75% for batch normalized models, but the test accuracy was very poor (around 45%) percent leading us to believe the model was over-fitting the data . The same was observed with the layer normalized as well un-normalized models.
- It also seems to us that the performance of the current models depends very much on the values of the initial weights, if the initial point is in a region where the gradients are big enough i.e the surface is not very flat we get convergence to much lower values of loss, if not the model trains for a much longer time while giving higher values of loss as well.
- A comparative plot of the loss curves for all three Normalization methods are shown here: *Please move to the next page to view them* .
  - Without any normalization the model is supposed to take more time to converge, but in the fist case, we see that no normalization does take more time than batch to converge but converges quicker compared to layer, but both normalization methods converge to a much lower value of loss.
  - When using normalization the model loss goes lower than the un-normalized model within a few epochs itself, thus aiding it to converge quicker and to a lower loss.

- Both batch and layer normalization have pretty much identical performances in terms of the final loss of the model on the training data as well as accuracy in test and training data.
- When trained for the same number of epochs we see that botch normalization methods have a much better performance.



(a) Error threshold=10^{-5},1000 max epochs



(b) Error threshold=10^{-6},1000 max epochs

Figure 28: All Losses

- There were cases when the normalized models incredibly well on the training data but were disappointing on the test and validation datasets, as they were outperformed significantly by the un-normalized models, so it seems that there is a chance of over-fitting your data with normalization methods. Would like to get this clarified please :) .

3 Task 3: Stacked autoencoder based pre-training of a DFNN based classifier

3.1 Introduction

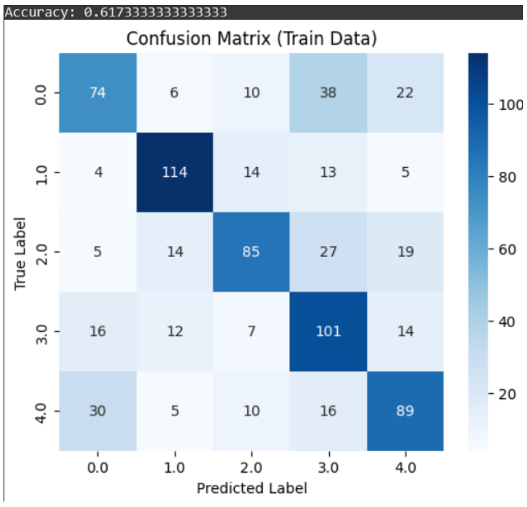
In the absence of sufficient labeled data, it is still feasible to extract the core features of the dataset using autoencoders. We use encoders to compress the input into a set of features and try to approximate it from these features. The idea is that there is an underlying pattern among all the inputs which captures most of the meaningful characteristics of the input in much smaller dimensions. We attempt to train the encoder to recognize this pattern.

3.2 Implementation

- We have three AANNs, with layer sizes as follows:
  - $AANN_1$: 36,28,20,28,36
  - $AANN_2$: 20,17,13,17,20
  - $AANN_3$: 13,10,7,10,13
- As we stack these encoders, we eventually reach a dimension of size 7 in our DFNN, which then has an output layer of size 5
- Since the AANNs have linear layers which can be merged using matrix multiplication, we effectively have a 5 layer DFNN.
- We train the AANNs using the loss function as MSE as it works best for regression tasks
- We use the AdaM update rule and mini batch mode of training as specified to train on the unlabeled dataset
- Once we train the AANNs we then copy these weights onto our DFNN as part of the pretraining
- As specified, we also have another DFNN which has all of its weights randomised as our control
- We then train/fine tune our models on the labeled dataset with a learning rate of 0.005 and stopping criteria threshold 0.0001

3.3 Results

- On an average, the pretrained model took lesser number of epochs to train
- The control took anywhere between 2x to 1.5x more epochs on average
- The accuracy on training data was 61.73% for the pretrained model and 58.93% for the control

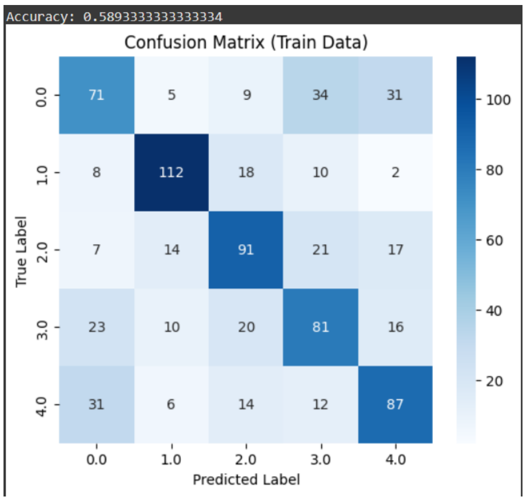


```
Accuracy: 0.6173333333333333
```

Confusion Matrix (Train Data)

True Label \ Predicted Label	0.0	1.0	2.0	3.0	4.0
0.0	74	6	10	38	22
1.0	4	114	14	13	5
2.0	5	14	85	27	19
3.0	16	12	7	101	14
4.0	30	5	10	16	89

(a) Pre-trained



```
Accuracy: 0.5893333333333334
```

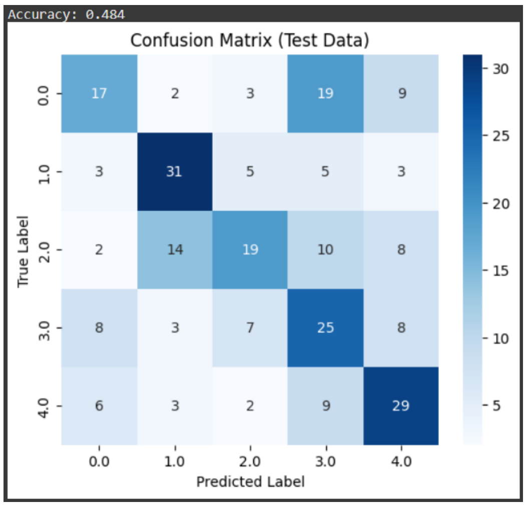
Confusion Matrix (Train Data)

True Label \ Predicted Label	0.0	1.0	2.0	3.0	4.0
0.0	71	5	9	34	31
1.0	8	112	18	10	2
2.0	7	14	91	21	17
3.0	23	10	20	81	16
4.0	31	6	14	12	87

(b) Control

Figure 29: Training Data Confusion Matrices

- The accuracy on the test data was 48.4% for the pretrained model and 47.6% for the control

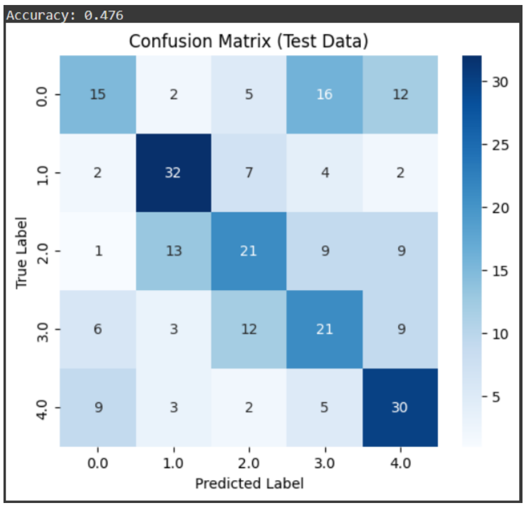


```
Accuracy: 0.484
```

Confusion Matrix (Test Data)

True Label \ Predicted Label	0.0	1.0	2.0	3.0	4.0
0.0	17	2	3	19	9
1.0	3	31	5	5	3
2.0	2	14	19	10	8
3.0	8	3	7	25	8
4.0	6	3	2	9	29

(a) Pre-trained



```
Accuracy: 0.476
```

Confusion Matrix (Test Data)

True Label \ Predicted Label	0.0	1.0	2.0	3.0	4.0
0.0	15	2	5	16	12
1.0	2	32	7	4	2
2.0	1	13	21	9	9
3.0	6	3	12	21	9
4.0	9	3	2	5	30

(b) Control

Figure 30: Test Data Confusion Matrices

- So, we can conclude that pre-training the model definitely gives better results consistently, the difference is not too substantial in terms of prediction.