CS6220
Big Data System and Analytics
Assignment 2

Sathvik Karatattu Padmanabha

September 2024

Student Session: cs6220-A
GT ID: xx4032361

Topic: Programming

Problem : 2.1

Title : Understanding Performance Tuning for
Training High Accuracy Deep or Shallow Models

Contents

1 Problem Chosen

Problem 2. Understanding Performance Tunning for Training High Accuracy Deep or Shallow Models

In this problem, you are given 2 options. In each option, you are asked to train multiple models using different hyperparameter configurations (option 1) or using different dataset complexity plus hyperparameter configurations (option 2).

Option 2.1 Performance Tunning for Deep or Shallow Learning

You are asked to train 5 models for the same learning task and dataset, each uses one configuration of the hyperparameters: the default configuration from the package you download, and the four different configurations you are asked to change and each of the four should have only one hyperparameter different to the baseline to help with the analysis on performance tuning results. If you use CPU without GPU card, then create a smaller dataset. Here are the steps:

(1) Choose your favorite ML framework, say TensorFlow, Scikit Learn, PyTorch, etc. or finetuning of your favorite pretrained LLM. (see Hugging face for models and datasets).

In the rest of the requirement, we will use TensorFlow as an example.

(2) After download TensorFlow at https://www.tensorflow.org. Following the instruction to install it on your laptop or desktop computer:

```
https://www.tensorflow.org/install/install_linux#ValidateYourInstallation.
There are several options to install TensorFlow on machines without GPU card(s).
For example, installing Tensorflow on virtualenv will isolate
your Tensorflow's python environment.
```

(3) Choose a dataset and a machine learning (ML) algorithm to train a baseline ML model using the default hyperparameters, say CNN3 MNIST, which use a 3 layer CNN to train a 10-class neural network classifier on MNIST dataset. Given MNIST dataset has 60K images. You may choose a subset of them, say 6000 total, evenly distributed for 10 classes, say 600 per class.

(4) You are asked to show the default settings of the hyperparameters of the TensorFlow CNN algorithm you used to train a k-class classification model, such as the CNN layers used, the #kernels (weight filters) used, the feature map sizes, the minibatch size, the #iterations (#epochs), the loss optimizer (e.g., SGD, Adam, etc.)

(5) Train your model using the algorithm and the dataset you have selected. Report the output of the trained K-class model, including

• the storage size of the model in MB, • the test accuracy (use a small portion of the training data as training validation), • the training time and • the test time. • In addition, you may want to show the confusion matrix for all K classes. For K¿10, you may choose to show a selected subset of classes. [Hint: see lecture 6 on confusion matrix]

(6) Outlier Test Scenario. When performing testing on a k-class classifier you have trained, in addition to use the test dataset from the same collection, you are asked to perform an outlier test by using 10 or more images that do not belong to the dataset of your learning task. For example, if you choose dog and cat as your binary classification task, then create 10 photos of your favorite actor/actress, or favorite cars which will be the out of distribution data. Perform outlier test on your classifiers and report your results in a table titled outlier test. Also include 5 examples of your outlier test set in your report.

(7) For each of the 10 outlier queries, and run each of the outlier tests at least three times and report your results, including • The results from the 3 times of the same outlier query: show if they return the same prediction result or not; • Make an attempt to elaborate on why the results from 3 times of test using the same test example may result in very different probability vectors. Hint: There are two ways to perform the above tests: (i) select 3 different pre-trained models on the same dataset, run each of the outlier tests against three models and analyze your results on the above two bullets. (ii) perform the same outlier query against the same model. Report your finding. Ideally compare with case (i). Elaborate your comparison results and your thought on the results.

(8) Now choose at least two different types of hyperparameters (say different learning rate functions and different #kernel filers) and for each type, select two settings (say fixed learning rate with

two different values, plus 10 kernel filers v.s. 5 kernel filers) and use the same training dataset (say MNIST) to train 4 different models for the same learning task (e.g., MNIST with 10 classes), each using one changed hyperparameter configuration, and compare the performance of these 4 newly trained models with the baseline model you have trained in step (1)-(7) in • training time, • test accuracy, • test time, • the storage size of the trained model

(9) Analyze the comparison results based on the confusion matrix and the different hyperparameter configurations you have used in the 5 models you have trained.

Hints: Hyperparameters include (i) different #kernels (weight filters) used, (ii) different learning rate function or settings, say different value for a constant learning rate; (iii) different convergence condition, e.g., #iterations (or #epoches), (iv) other hyperparameters of your own choice, including different training algorithm: say you used LeNet MNIST and now change to another algorithm, such as ResNet32, ResNet64, MobiNet, VGG16, ...

Deliverable: (1) provide URL of your open source code package and the dataset download. (2) Screen shots of your execution process/environments (3) Input Analysis: Use a table to report your training configuration parameters: a. the input dataset (size, resolution, storage size in KB or MB per image, storage size of dataset in MB or GB). b. choose and show 5 sample images per class for all K classes in the dataset. c. the training v.s. testing data split ratio and size used in your CNN training. d. You are asked to record the structure and default settings of the neural network (NN) algorithm you chose to train your k-class classifier, such as LeNet, or ResNet, or DenseNet, and the default NN structures (e.g., CNN with at least 2 5 convolutional layers), and the default hyper-parameters, such as neuron size, the number of weight filters (kernels) and the size of kernel (weight filter) for each layer of your CNN, the min-batch size, #epochs/#iterations (convergence), in a table for the configuration you use to train your four CNN classifiers. e. You are asked to report the default #iterations used for training convergence.

(4) Output Analysis: Report the performance comparison and analysis of each of your 5 CNN classifiers: a. You are asked to provide a table to compare the 5 models you have trained in terms of training time, training accuracy, testing time and testing accuracy. b. You are asked to record the trained model size in MB in the above table for all 5 models. c. You are asked to make at least three observations from your experimental comparison of the 5 models that you have trained on the same dataset under 5 different hyperparameter configurations (including the baseline model using the default hyperparameters).

2 Introduction and README

I have chosen MNIST classification. The MNIST is a database of handwritten digits. It is of size 21.00 MiB, but I am using a subset of this for this assignment.

I am using the tensorflow framework for experiment. I am also using numpy, pandas, matplotlib, seaborn and a few other libraries for data manipulation and visualization.

I am using the PACE Jupyter network to run the experiments. I used the NVIDA V100 32 GB GPU to accelerate training and I used a anaconda environment to install the necessary libraries.

The .ipynb notebook is included with the submission that has the entire code. Most of the code reference is from kaggle and other official websites such as tensorflow website.

Here is the list of conda commands that I used to setup the environment

```
conda create -n test_gpu python=3.9
conda activate test_gpu
#conda install -c conda-forge notebook #if jupyter is not accessed separately
conda install numpy=1.23.1 matplotlib seaborn scikit-learn
conda install -c conda-forge tensorflow-gpu
conda install ipykernel
python -m ipykernel install --user --name=bda --display-name "bda"
conda install Jinja2
conda install tabulate
conda install pydot
```

The github repo link containing the code and other files is provided below. However you need to request access from me as it is private. I have anyways provided the .ipynb book along with the submission.

`https://github.com/SathvikKP/CNN_Hyperparameter_MNIST`

The Figures are present in the images directory and The outliers input images is present in the outliers directory.

The directory "architectures" and the text file "Model_Architectures.txt" contain the model architectures images and model descriptions with weights obtained by tensorflow's model.summary(). I have not included every model image and structure in the report as it will be too large and cumbersome.

3 Notebook details

I have neatly added markup cells describing the actions taking place in the code cells. I have written several auxiliary functions to make code more understandable and reusable.

Step 1: Importing necessary libraries
Step 2: Check for GPU
Step 3: Function to Load the MNIST Dataset
Step 4: Function for exploratory data analysis
Step 5: CNN3 Model
Step 6: Auxillary functions for training models
Step 7: More Auxillary functions
Step 8: Outlier Testing Auxillary Functions
Step 9: Auxillary Functions to compare with Standard Models
Step 10: Load Data and perform EDA
Step 11: Define Configurations and train, evaluate the model
Step 12: Compare with baseline and visualize predictions
Step 13: Outlier Testing
Step 14: Train and Evaluate standard models, Compare with earlier results
Step 15: Printing MODELS (to get the model architecture details)

Both the .ipynb notebook file and a .py python file for the same code are provided in the zip file. They can be easily run provided the conda environment is setup with the instructions provided in the previous section.

The deliverable trained models (tensorflow .h models) are also included in the zip file. You can simply load the module and test using tensorflow functions.

4 Dataset details

Tensorflow conviniently has the MNIST dataset readily available. We need to use mnist.load_data() function to get a tuple of train and test images and labels. For purposes of the experiment, I am using a subset (one-tenth) of the dataset. Why only a subset? Because when I used the entire dataset, it resulted in high and equal accuracies for each model which made comparison difficult. Also, the problem asks to use a subset.

I am using a train:validation:test set ratio of 60:20:20 The details are shown below and visualizations through the Figures 1, 2 3, 4, 5, and 6.

```
Train Set Metrics:
Total Number of Images: 3600
Number of Images per Class: {0: 327.0, 1: 360.0,
2: 353.0, 3: 371.0, 4: 357.0, 5: 379.0, 6: 372.0,
7: 347.0, 8: 361.0, 9: 373.0}
Total Image Size (in pixels): 2822400
Average Image Size (in pixels): 784.00
Total Size of Images (in bytes): 10.77 MB
Image Resolution (Height x Width): 28 x 28

Validation Set Metrics:
Total Number of Images: 1200
Number of Images per Class: {0: 146.0, 1: 118.0,
2: 123.0, 3: 116.0, 4: 114.0, 5: 111.0, 6: 114.0,
7: 131.0, 8: 114.0, 9: 113.0}
Total Image Size (in pixels): 940800
Average Image Size (in pixels): 784.00
Total Size of Images (in bytes): 3.59 MB
Image Resolution (Height x Width): 28 x 28

Test Set Metrics:
Total Number of Images: 1200
Number of Images per Class: {0: 127.0, 1: 122.0,
2: 124.0, 3: 113.0, 4: 129.0, 5: 110.0, 6: 114.0,
7: 122.0, 8: 125.0, 9: 114.0}
Total Image Size (in pixels): 940800
Average Image Size (in pixels): 784.00
Total Size of Images (in bytes): 3.59 MB
Image Resolution (Height x Width): 28 x 28
```

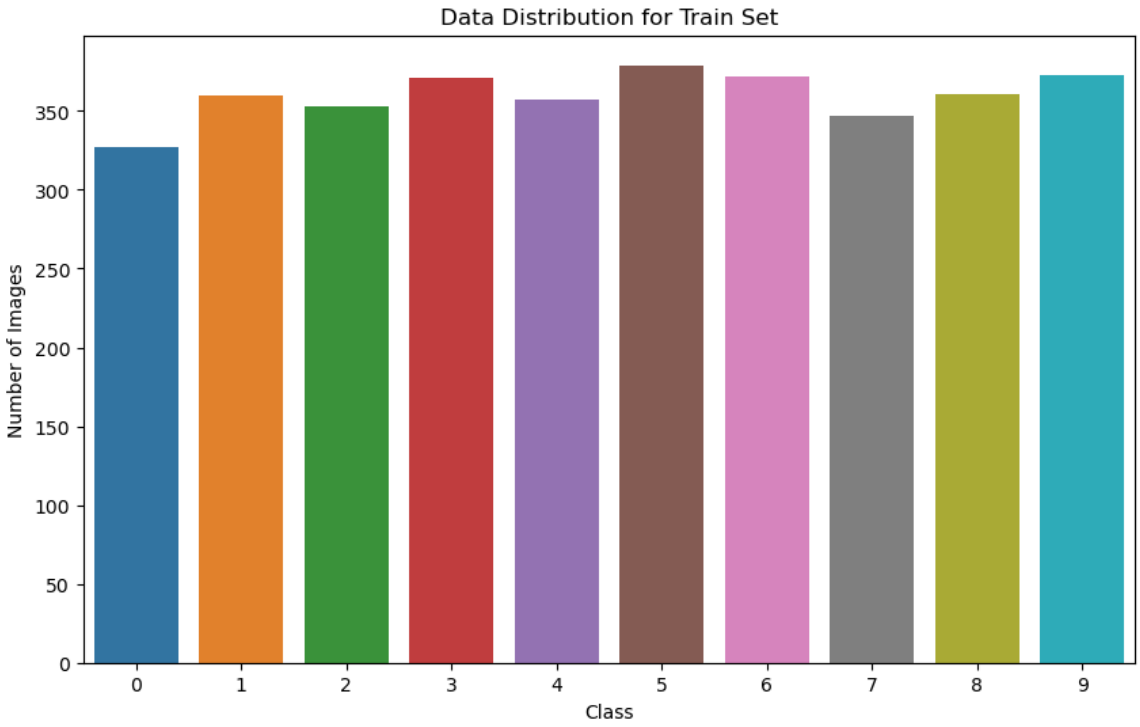


Figure 1: Data Distribution for Train Set.

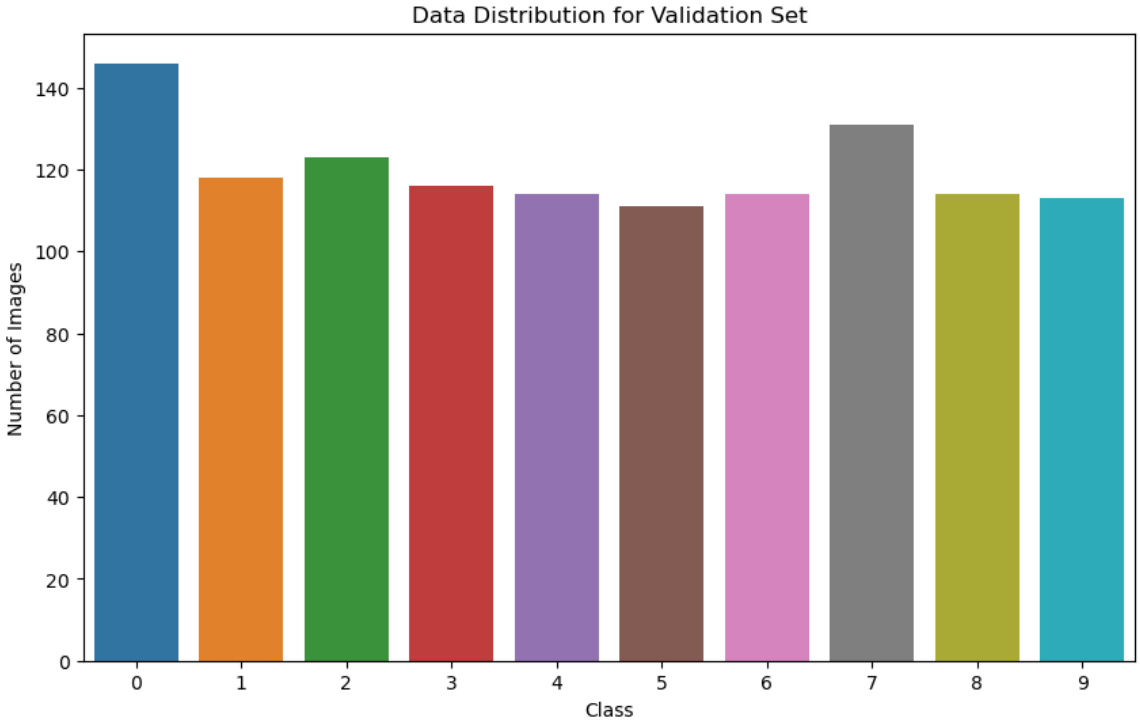


Figure 2: Data Distribution for Validation Set.

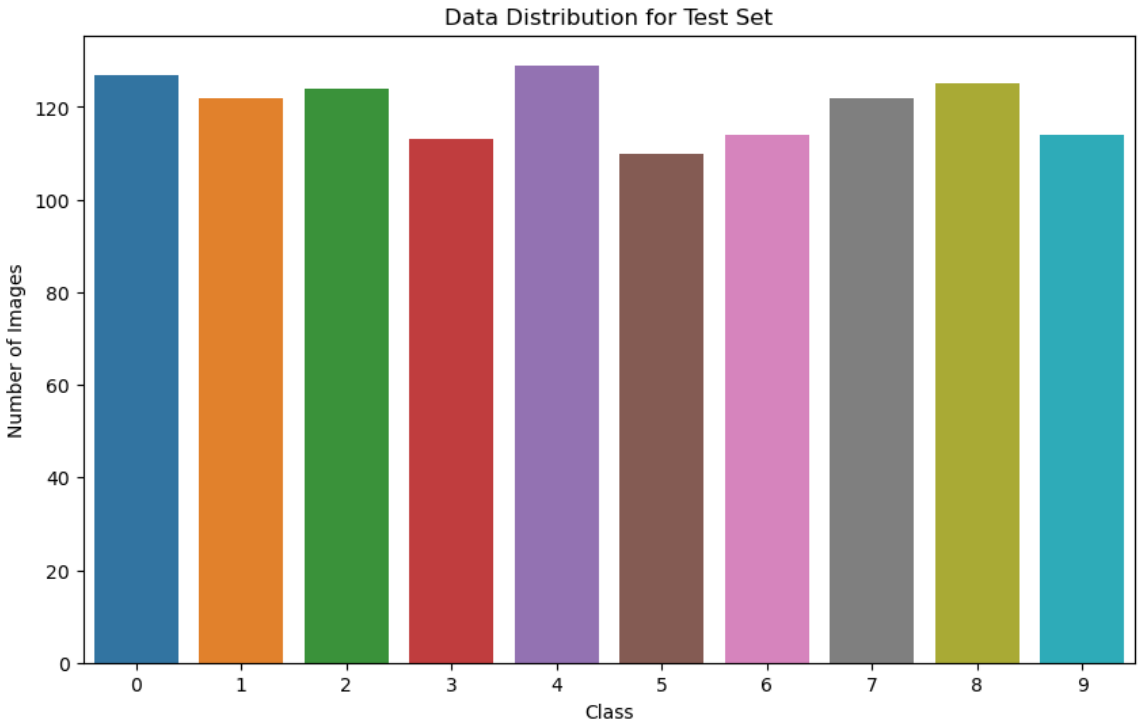


Figure 3: Data Distribution for Test Set.

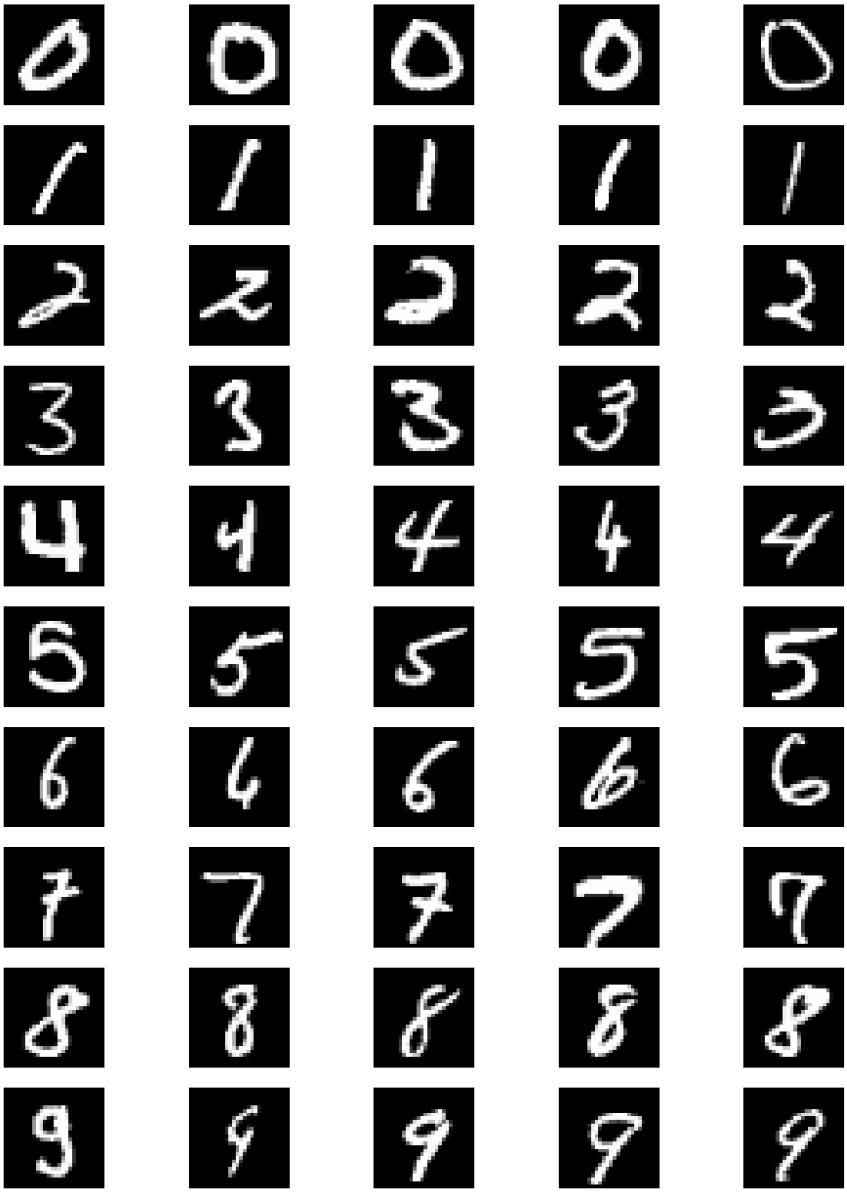


Figure 4: Train Set Sample images for all classes

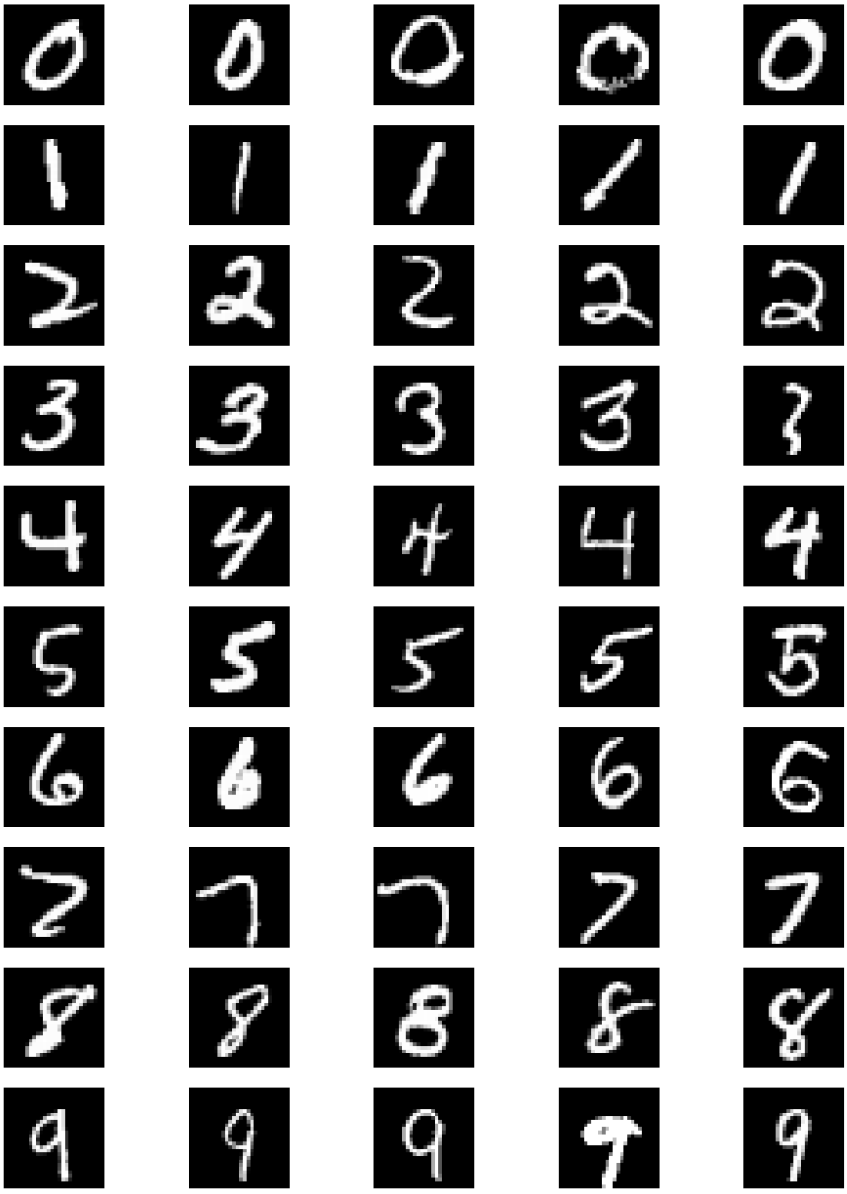


Figure 5: Validation Set Sample images for all classes

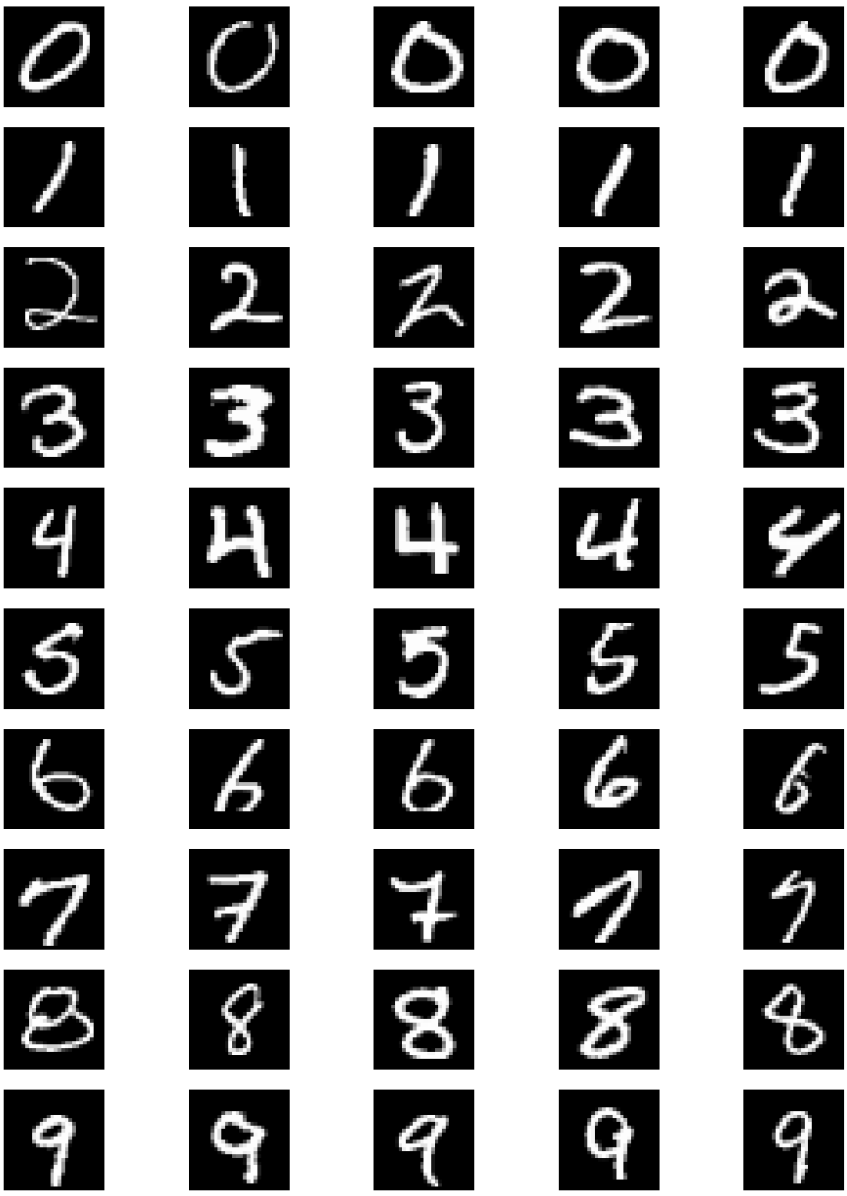


Figure 6: Test Set Sample images for all classes

5 Baseline model

I have chosen the baseline model as a simple 3 layer CNN model. The details of the model are provided in Figure 0.

5.1 Baseline model Architecture

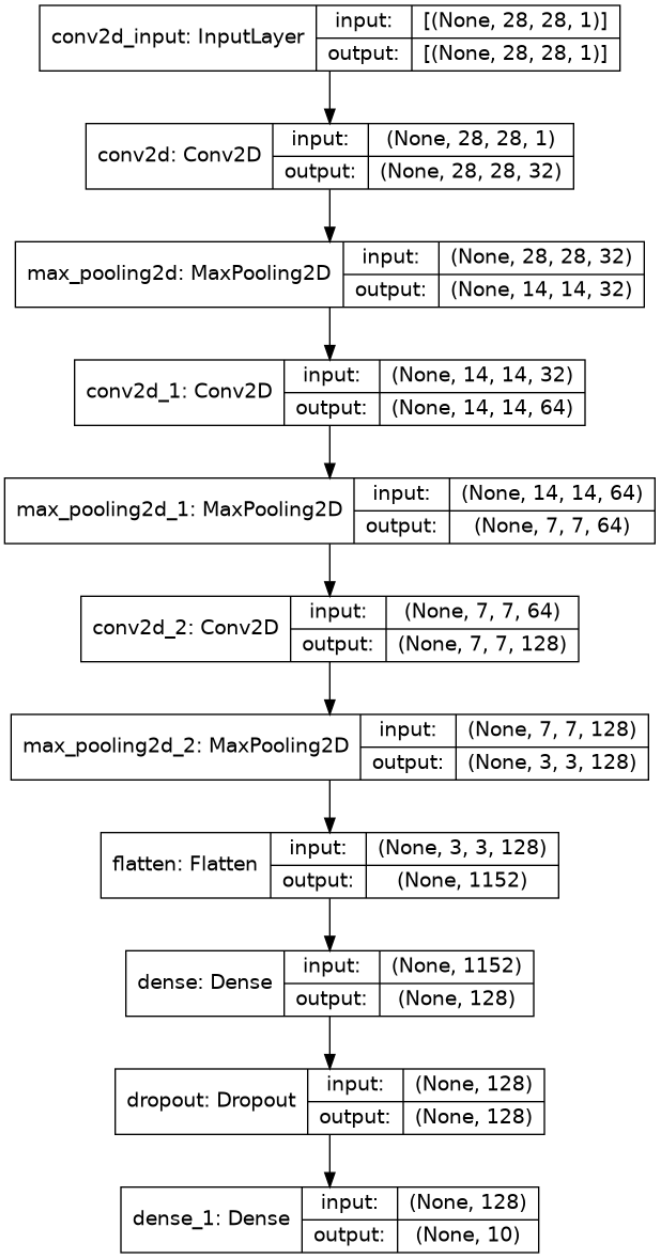


Figure 0: Baseline Model Architecture

The model.summary() of tensorflow gives the number of parameters, layers etc. For the baseline model, it shows the following output

```
Visualizing predictions for Model_0.h5...
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 128)         73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 128)               147584
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 241,546
Trainable params: 241,546
Non-trainable params: 0
_________________________________________________________________
```

5.2 Reason for choice of baseline

I chose this model based on rough experimentation as it has decent complexity and gives good accuracy for MNIST dataset. The best model architectures are determined by extensive experimentation and sometimes, it can be difficult to explain why a particular model performs better than others. If models are too complex, then, we can analyze by looking at results.

5.3 Baseline Parameters

I have tested by varying 7 different parameters. The parameters and their values are listed below.

- Optimizer: 'adam' - Adam is an adaptive learning rate optimization algorithm widely used for training deep learning models.
- Learning rate: 0.001 - The learning rate determines the step size during gradient descent, with 0.001 being a common default for Adam.
- Number of filters: [32, 64, 128] - The model has 3 convolutional layers with 32, 64, and 128 filters respectively, controlling the number of feature detectors.
- Feature map size: 3 - The feature map size (or kernel size) of 3 refers to the 3x3 filter dimensions used in each convolutional layer.
- Dropout rate: 0.25 - Dropout randomly drops 25% of the neurons during training to prevent overfitting.

- Batch size: 128 - The model processes 128 samples at a time before updating the weights during training.
- Epochs: 5 - The model is trained over the entire dataset for 5 complete passes (epochs).

5.4 Iterations per epoch calculation

$$\text{Iterations per epoch} = \frac{\text{Number of samples}}{\text{batch size}}$$

For this experiment with default number of epochs = 5, batch size = 128 and training set sample = 80 % of 6000 samples (train + validation), we get iterations per epoch as 4800 / 128. We multiply this by 5 to get total number of iterations = 187.5. However, we need to be aware about the dropout rate = 0.2. Hence we multiply by 0.8 and get approximate number of iterations for baseline as 150. We can verify this by looking at the tensorflow output logs and adding up each iteration count.

6 Training Baseline Models and varying hyper-parameters to get new models

In the .ipynb notebook, I have setup a function where it gets different configurations, compiles model for each, performs training, testing and saves the model. The environment execution screenshots will be provided later down the report in a separate section. I will only print the confusion matrix and training plots for each model, and the final summary comparing the models so that we can analyse the effect of hyperparameters.

6.1 Summary Table

This is the overall summary after varying 7 hyper-parameters. The model statistics and architectures are provided in separate directories and parameter details are present in a text file titled "Model_Architecures.txt".

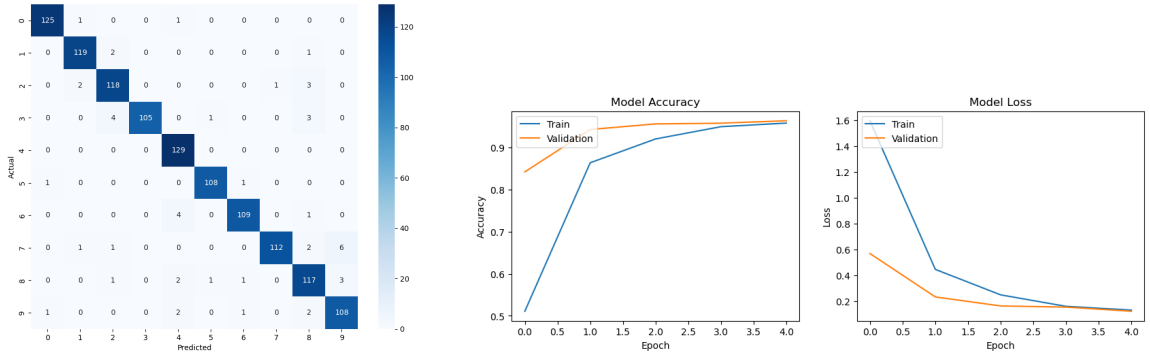
Comparison of Each Model with the Baseline Model

Model	Training Time (s)	Training Accuracy (%)	Validation Accuracy (%)	Testing Time (s)	Testing Accuracy (%)	Model Size (MB)
Baseline model	13.58	95.86	96.42	1.97	95.83	2.82
Different optimizer	27.01	88.19	92.00	0.11	90.08	0.95
Different learning rate	1.37	93.94	95.75	0.11	94.42	2.82
More filters	12.41	96.78	96.83	2.07	96.25	7.67
Larger feature map size	11.16	97.03	97.08	1.79	96.17	4.70
Higher dropout rate	1.56	94.06	96.67	0.10	95.92	2.82
Smaller batch size	1.71	96.69	96.50	0.10	96.17	2.82
More epochs	2.10	98.31	97.00	0.10	96.75	2.82

Figure 7: Summary Table

6.2 Baseline model

We will use the baseline model for comparing the effect of varying other hyper-parameters. Figure 7 shows the various details. Figure 7.0 shows the confusion matrix and the accuracy and loss plots.



(a) Confusion Matrix

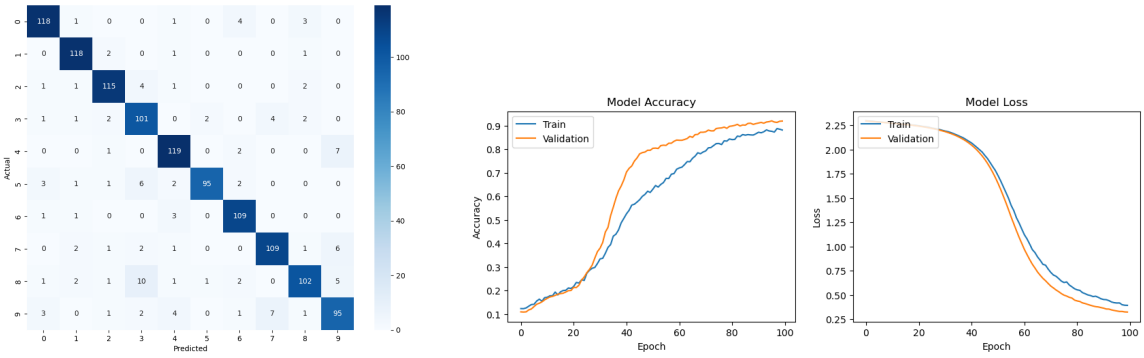
(b) Accuracy and loss plots

Figure 7.0: Baseline model statistics

Analysis: Final testing accuracy of 95.83 percent. Moderate execution times and moderate model size. We will use this to compare other hyper-parameter effects.

6.3 Model 1: Different Optimizer SGD

Let us use Stochastic Gradient Descent (SGD) as the optimizer keeping other baseline hyper-parameters constant. I will also increase number of epochs to 100 only for this model as I got terrible accuracies for just 5 epochs. Figures 7 and 7.1 shows the statistics.



(a) Confusion Matrix (b) Accuracy and loss plots

Figure 7.1: Model 1 - SGD Optimizer

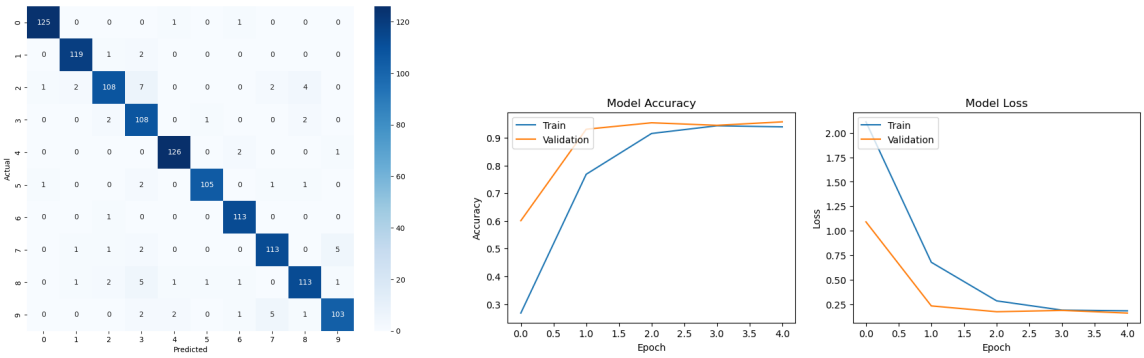
Analysis: We can see that from Figure 7 that it took a longer time to train. But this was because we have more number of epochs. If we keep number of epochs as 5, then it takes faster than baseline model. We also observe that the accuracies are lower and the model size is also less.

Hence, Adam is the better optimizer for MNIST classification.

Note: There can be other applications where SGD may be the better choice.

6.4 Model 2: Different learning rate (Increase to 0.01)

Let us change the learning rate to 0.01 and keep other paramters same as baseline model. Figures 7 and 7.2 shows the statistics.



(a) Confusion Matrix (b) Accuracy and loss plots

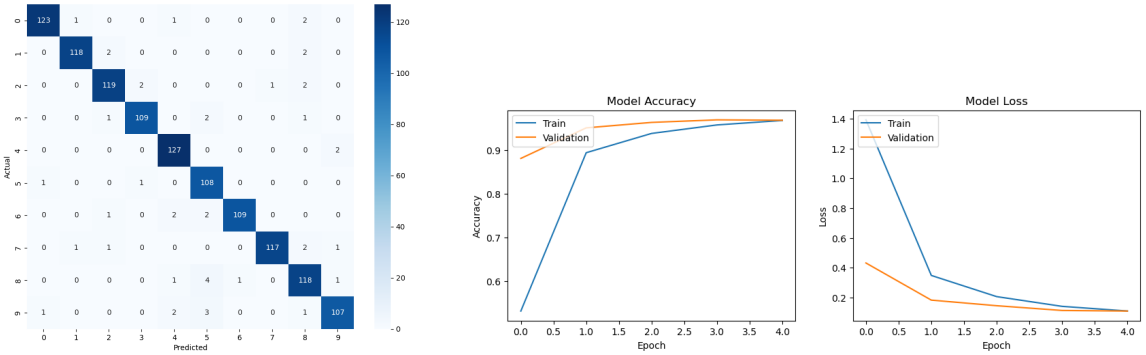
Figure 7.2: Model 2 - Increased Learning rate

Analysis: We can see from Figure 7 and 7.2 that the training time greatly reduced due to the increased learning rate. The model size remains same as only one value is changed.

However the accuracies are reduced. A more dangerous trend is observed in the accuracy and loss plots. As the learning rate is higher, it has overshot the optimal point and the validation accuracy is lower than training accuracy at the end. Hence, we need to be careful about increasing learning rate. We should always keep a small learning rate if erratic trends are observed in the accuracy and loss plots. Hence, learning rate = 0.001 is more optimal than 0.01 in this experiment.

6.5 Model 3: More filters (increase to [64, 128, 256])

Let us increase (double) the number of filters at each convolution layer and compare with baseline model. Figures 7 and 7.3 show the statistics.



(a) Confusion Matrix (b) Accuracy and loss plots

Figure 7.3: Model 3 - Increased number of filters at each layer

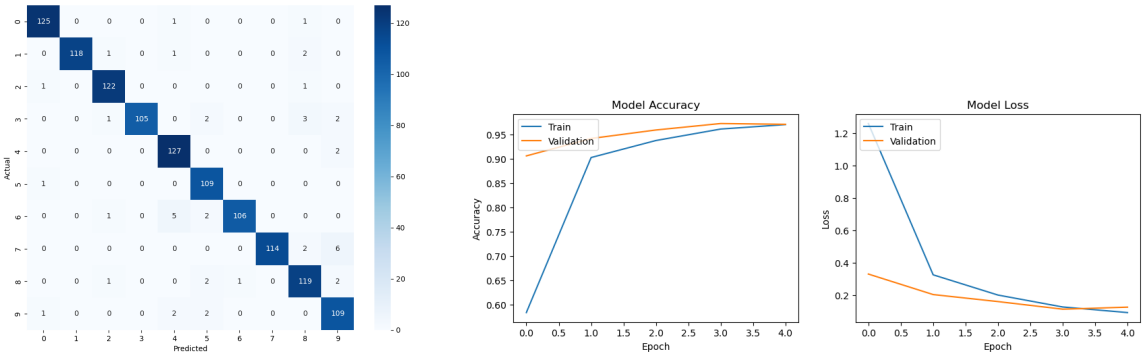
Analysis: We can see that the there is a considerable increase in training times and also the model size. This is expected as the complexity of model and number of model weights are increased. Note that the reported baseline model training time include some time for GPU initialization, hence, Model 3 has more training time.

From Figure 7, The test accuracy has also improved compared to the baseline model.

The choice of if increased number of filters is desirable depends on the use case and resource availability. If more resources are available, then increased number of filters is desirable. Otherwise, reduced number of filters at slight accuracy cost is the best choice.

6.6 Model 4: Larger feature map size

Let us increase the feature map size from 3 to 5. Now, the architecture contains 5x5 filters instead of 3x3 filters. Figures 7 and 7.4 show the statistics.



(a) Confusion Matrix (b) Accuracy and loss plots

Figure 7.4: Model 4 - Increased Feature map size

Analysis: We can see from Figure 7 that the training time is considerably high as more number of calculations are required for each convolution step when we increase feature map size (Note : baseline model training time includes GPU initialization).

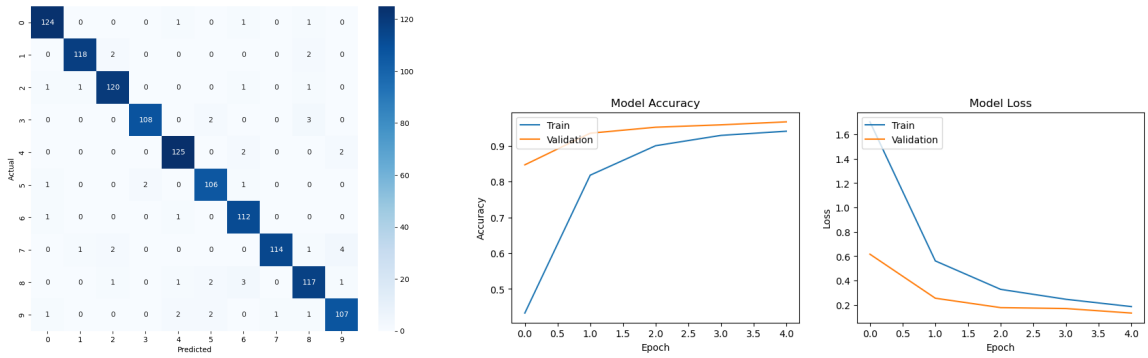
The model size is also increased due to increased feature map size.

We also see that this model produced the highest training and validation accuracy and relatively high test accuracy. This is expected as the increased number of feature maps better understands the data and produces a better model.

Similar to the previous model, the question of if increased feature map size is desirable depends on use case and resource availability.

6.7 Model 5: Higher dropout rate

Let us increase the dropout rate to 0.4. Figures 7 and 7.5 shows the statistics.



(a) Confusion Matrix (b) Accuracy and loss plots

Figure 7.5: Model 5 - Higher dropout rate

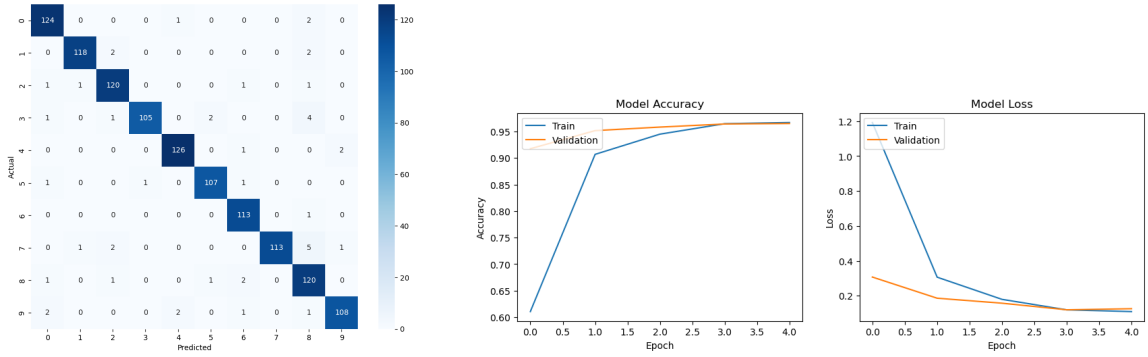
Analysis: We can see from Figure 7 that the training time is reduced by a considerable amount. This is because at each step, certain % of neurons is removed for that training step, reducing training time.

The testing accuracy is higher but training accuracy is lower than baseline model. This is the expected result as because of increased drop out rate, the model becomes more robust at dealing with unseen data.

The model size remains the same as change in dropout rate is just a numerical change.

6.8 Model 6: Smaller batch size

Let us reduce the batch size to 64 keeping other params same as baseline model. Figures 7 and 7.6 shows the statistics.



(a) Confusion Matrix (b) Accuracy and loss plots

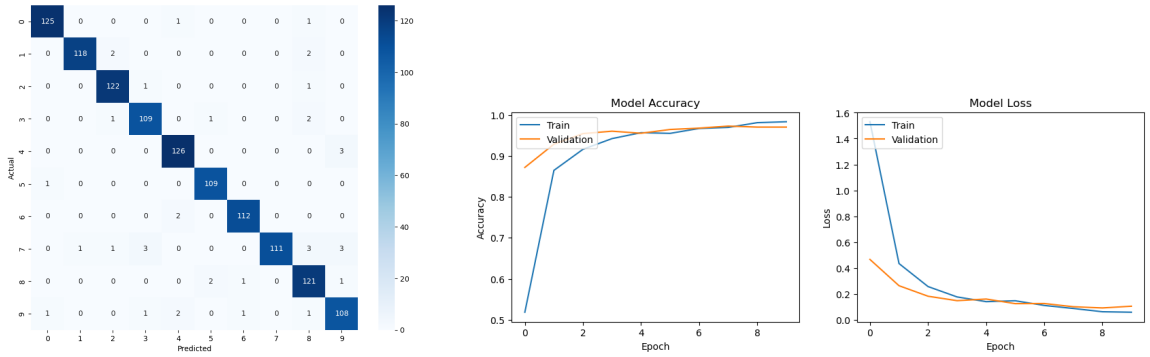
Figure 7.6: Model 6 - Smaller batch size

Analysis: From Figure 7, we see that training time is lower than baseline model but higher than few of the other models. The accuracies are comparable but testing accuracy for smaller batch size is slightly higher than baseline model. The model size remains the same.

We need to see the accuracy and loss plots and compare with baseline model to get a better picture. From Figure 7.6, we see that the there is no considerable improvement after epoch 3 (4 epochs) . This is not the ideal condition as the plots show that it has converged rather erratically. Hence, batch size is 128 is more ideal than 64 when other parameters are kept constant.

6.9 Model 7: More epochs

Let us double the number of epochs to 10. Figures 7 and 7.7 show the statistics.



(a) Confusion Matrix

(b) Accuracy and loss plots

Figure 7.7: Model 7 - More epochs

Analysis: The training time is higher as more number of iterations and calculations are performed. The model size remains same as number of epochs is just a numerical change.

The training and testing accuracy is indeed the highest among all models. However let us look at the accuracy and loss plots. The plots seem erratic and validation accuracy after epoch 4 (5 epochs) is lower than training accuracy at that point. This is not ideally expected. Hence, we need to change other parameters in addition to number of epochs to have stable plots which is the ideal outcome.

6.10 Summary

The confusion matrix also plots some details as to which model performs better for which character. 2 models may have similar overall accuracy but perform differently for different classes.

Example: Comparing confusion matrices of model 3 and model 4 (Figures 7.3 and 7.4), we see that model 3 performs better than model 4 when actual value is 7. Model 4 predicts 7 as 9 wrongly 6 times but model 3 predicts it wrong just once. Insights like these help designers develop better models for particular tasks.

Overall, adam is the better optimizer, learning rate = 0.001 is more optimal, dropout rate = 0.4 would make the model more robust, batch size 128 is more optimal.

The feature map size and number of filters depends on resource availability. If both these are increased, then increased number of epochs would be optimal.

7 Prediction Examples for All 8 models

The model predicts the number in the image provided (assuming there is a single numeric character in the image and the image is of proper size).

I have shown randomly chosen 15 examples from test set, their actual and predicted values for baseline and 7 models. Figures 8.0 - 8.7 show the predictions.

Figure 8.0: Baseline Model Predictions

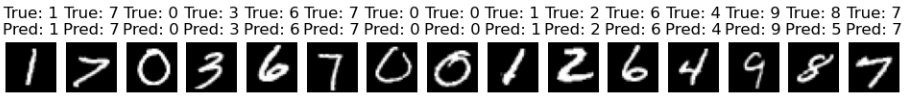


Figure 8.1: Model 1 Predictions

Figure 8.2: Model 2 Predictions

Figure 8.3: Model 3 Predictions

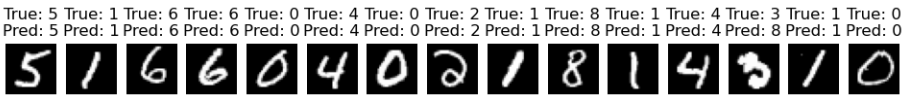


Figure 8.4: Model 4 Predictions

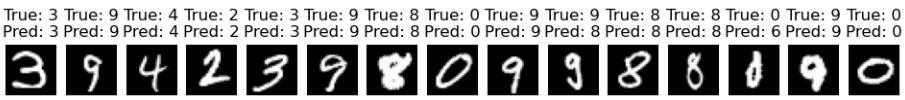


Figure 8.5: Model 5 Predictions

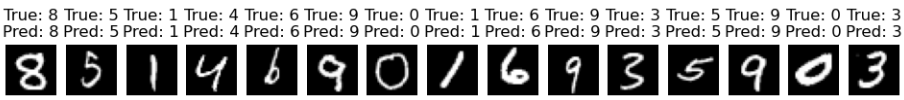


Figure 8.6: Model 6 Predictions

Figure 8.7: Model 7 Predictions

8 Outlier Testing

8.1 Outlier dataset

Outliers are certain data points that are considerably different from the rest. For MNIST classification, I will define outlier as some non numeric characters. I have downloaded 10 random characters from cyrillic-MNIST dataset. Figure 9 shows the outlier images labelled by me as 'a', 'b', 'zeta', 'm', 'e', '6_tilt', 'g', 'z', 'p', 'Oo'.

Figure 9: Outlier dataset

8.2 Outlier predictions

The expectation is that all models predict the characters similarly. However, the results are a little different. I tried predicting each outlier character with all the 8 models and found contrasting results. I have shown the results in this document for the first 5 letters via Figures 10.0 to 10.7

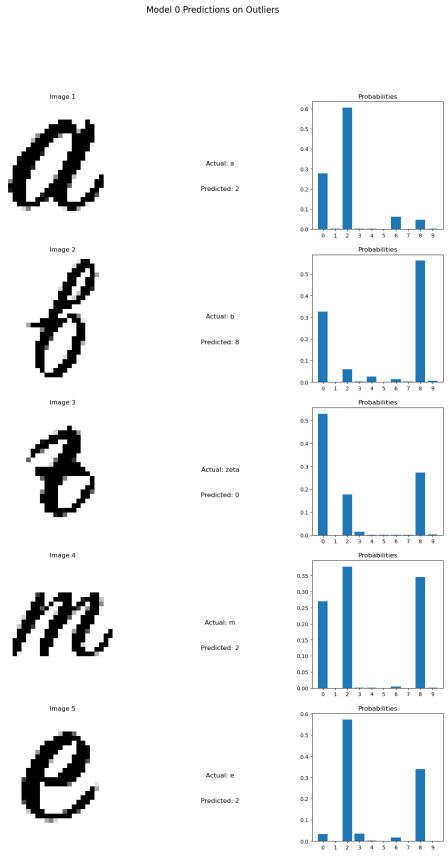


Figure 10.0: Baseline model outlier prediction

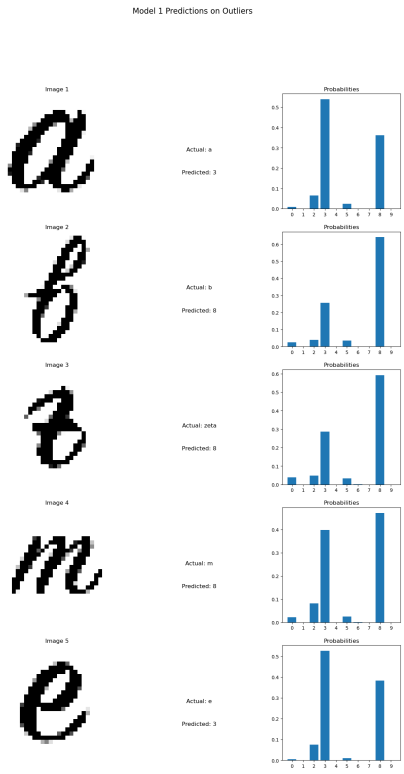


Figure 10.1: Model 1 outlier prediction

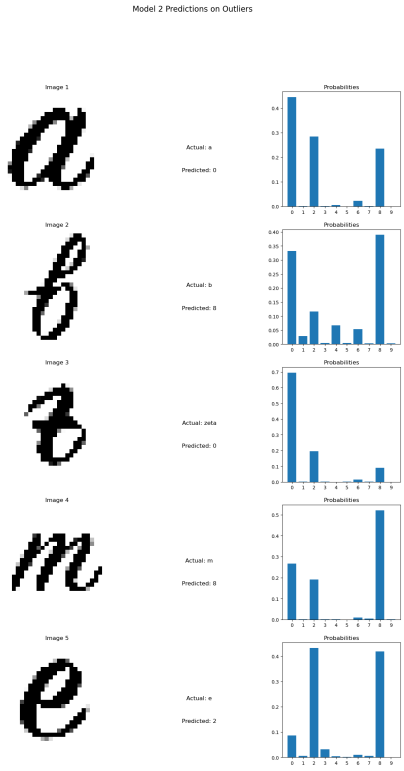


Figure 10.2: Model 2 outlier prediction

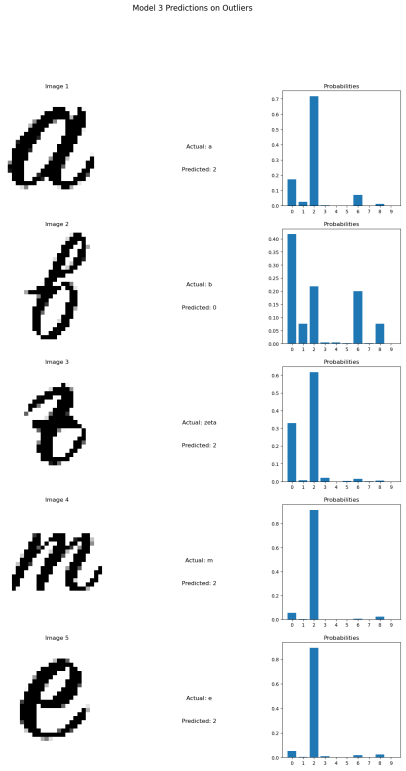


Figure 10.3: Model 3 outlier prediction

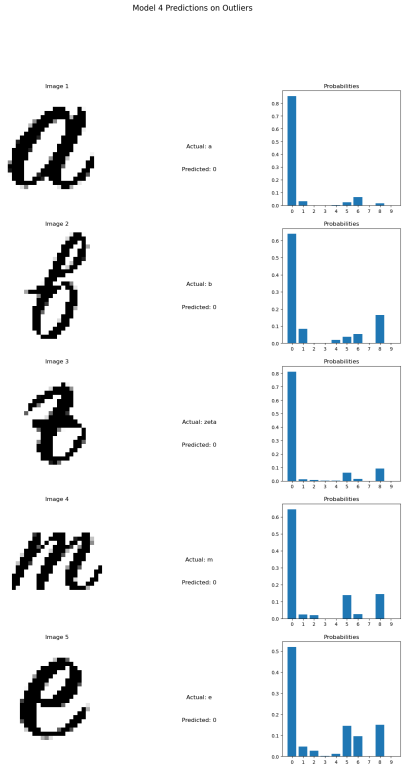


Figure 10.4: Model 4 outlier prediction

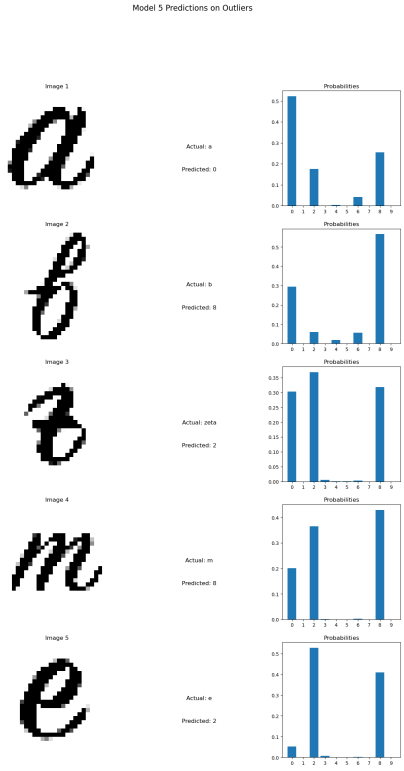


Figure 10.5: Model 5 outlier prediction

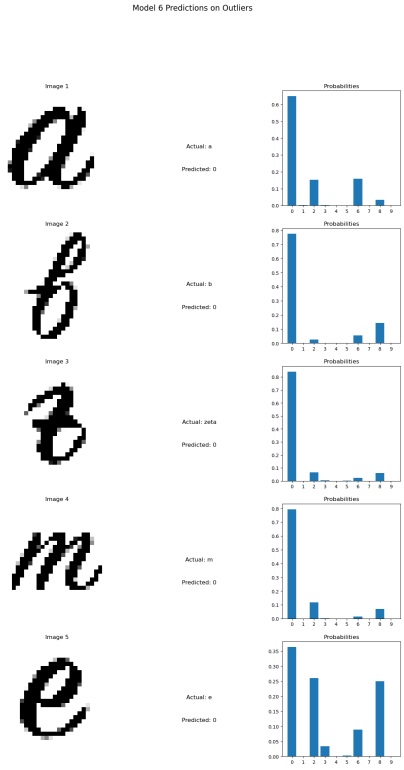


Figure 10.6: Model 6 outlier prediction

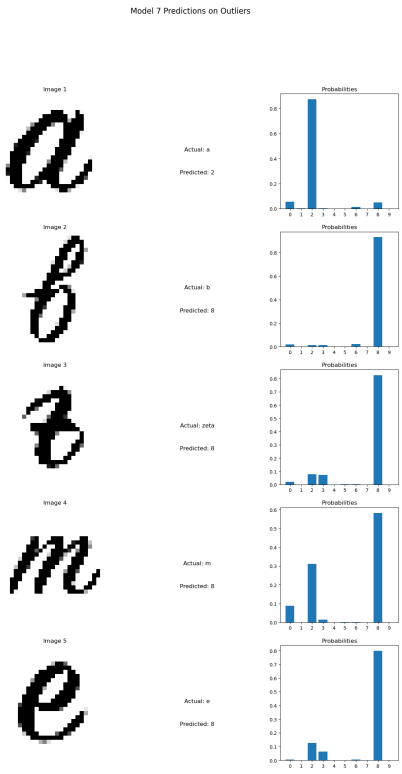


Figure 10.7: Model 7 outlier prediction

8.3 Outlier Testing Analysis

Analysis: We can see that each model predicts each characters with different probabilities. This is because the model is not trained to predict based on these characters.

Consider Figures 10.1 and 10.2. Model 1 predicts 'a' as 3, but model 2 predicts 'a' as 0. Model 1 predicts 'zeta' as 8 but model 2 predicts 'zeta' as 0.

A trend that I observed is that most predictions for the outlier character dataset is for the classes 0, 2 and 8. This is because these appear more similar to the characters in the outlier set.

Hence, if we encounter more outliers, we may have to determine if these are valid cases. If yes, we need to include these in the training process and we may have to include new corresponding classes also. Otherwise, we can eliminate these from the training process.

Analysis for repeated outlier query on the same model: In my case, it produced the same output most of the times, but ideally, it should not produce the same output frequently as the model was not trained for these outliers. In my case, the outlier characters were almost 'similar' in nature to numeric characters of MNIST. Hence, it produced same result most of the time.

Suppose I include a picture of a car as an outlier. In this case, the model will predict different outcomes very frequently as image of car is very different from characters or numbers.

9 Comparision with Standard Models

9.1 Standard Models

Additionally I have compared the baseline model with standard models. I have chosen LeNet, ResNet50 and DenseNet121.

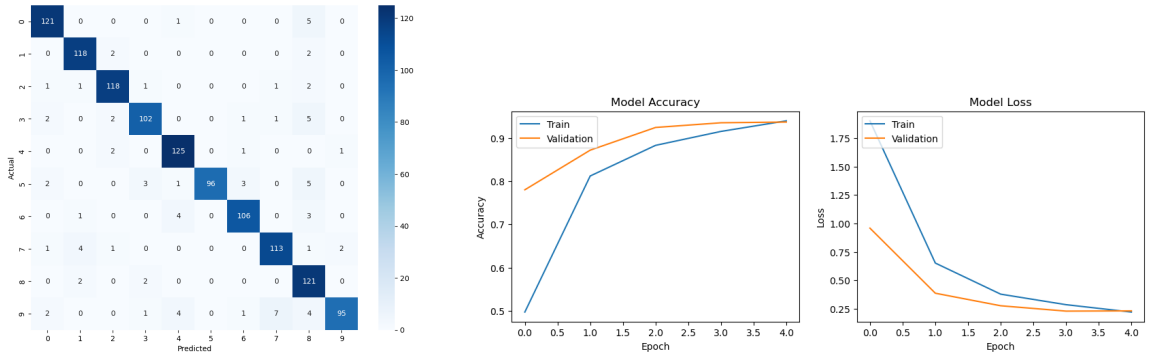
LeNet is very similar to the 3 layer CNN baseline model. However ResNet50 and DenseNet121 are different in the following ways.

- Both these architectures are designed for 3 colours (RGB). Hence, we need to add these channels to the grayscale MNIST dataset
- Both these accept input size of 32 x 32. Hence we need to preprocess the 28 x 28 MNIST images
- Simply training the model for same number of parameters provide horrible accuracy for MNIST classification as these models have significantly higher number of parameters. These models are designed for large number of classes and require huge amount of data to train. Hence to mitigate this to draw reasonable comparision with baseline model, we import the weights, then fine tune these. Tensorflow has these models conviniently available to use.

I have written separate auxiliary functions for importing, pre-processing images, training and evaluation. The model architectures and parameter descriptions are provided separately in the directory "images/architectures" and "Model_Architectures.txt".

The dataset is the same as we used for baseline model. The hyper-parameters such as optimizer, learning rate epochs, etc are same as baseline model except for some features like number of layers which is architecture dependent. The results are shown in the following sections (Figures 11.1, 11.2 and 11.3).

9.2 Model: LeNet

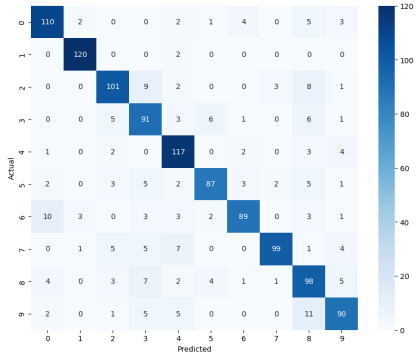


(a) Confusion Matrix

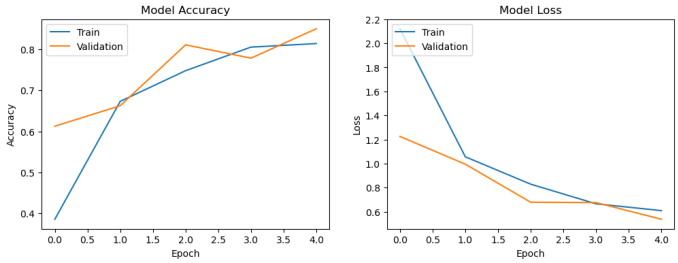
(b) Accuracy and loss plots

Figure 11.1: LeNet

9.3 Model : ResNet50



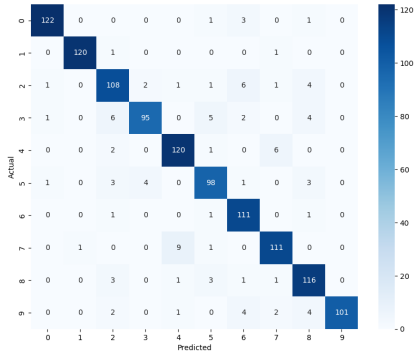
(a) Confusion Matrix



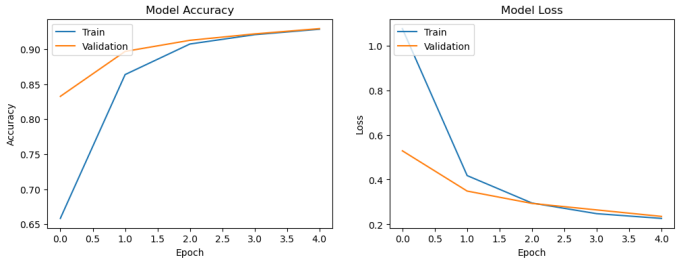
(b) Accuracy and loss plots

Figure 11.2: ResNet50

9.4 Model : DenseNet121



(a) Confusion Matrix



(b) Accuracy and loss plots

Figure 11.3: DenseNet121

9.5 Comparison Results

Figures 12, 13, 14, 15, 16 and 17 shows the comparision of Standard models vs the models that we designed. Please note that the numbers are '1-indexed' in these images (1,...,8) whereas we '0-indexed' them in earlier images (baseline,1,..7).

Only LeNet performs similarly to baseline model. ResNet50 and DenseNet121 perform slightly worse in terms of accuracy. We need to remember that the comparisons are made with hyper-parameters similar to baseline model and hence, the standard models' results look weak. With ideal hyper-parameters and more number of epochs, standard models should outperform the baseline model that we designed. DenseNet121 and ResNet50 will perform better for advanced tasks if there is very large data (several times MNIST data) and more number of classes.

The training time and testing time for Resnet50 and DenseNet121 are very large owing to their numerous parameters and complex deep structures. The model sizes are also very large for these models. However an interesting observation is that despite taking more time to train, DenseNet121 has smaller model size.

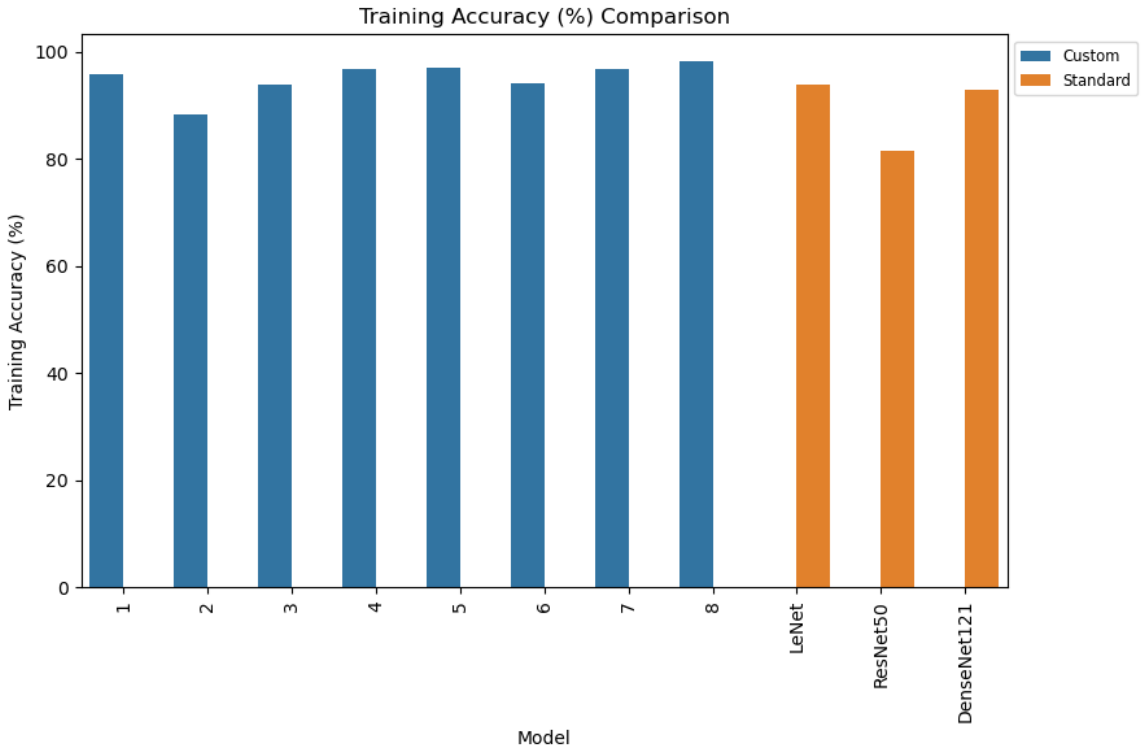


Figure 12: Training Accuracy comparison.

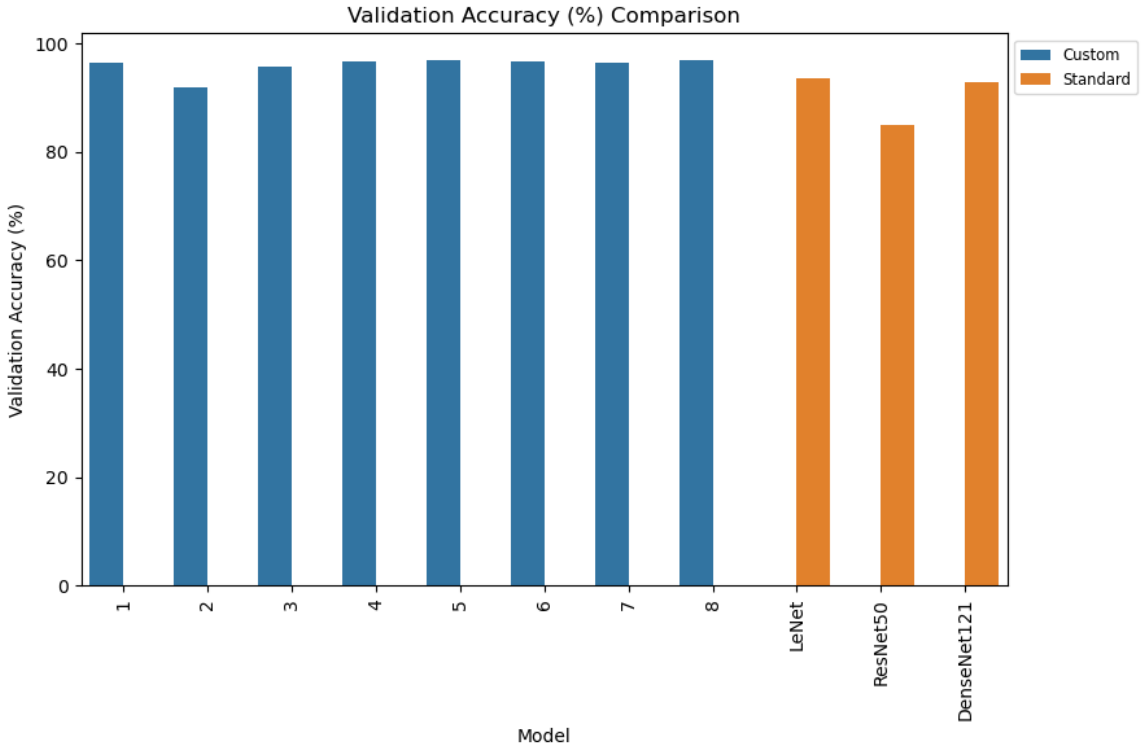


Figure 13: Validation Accuracy comparison.

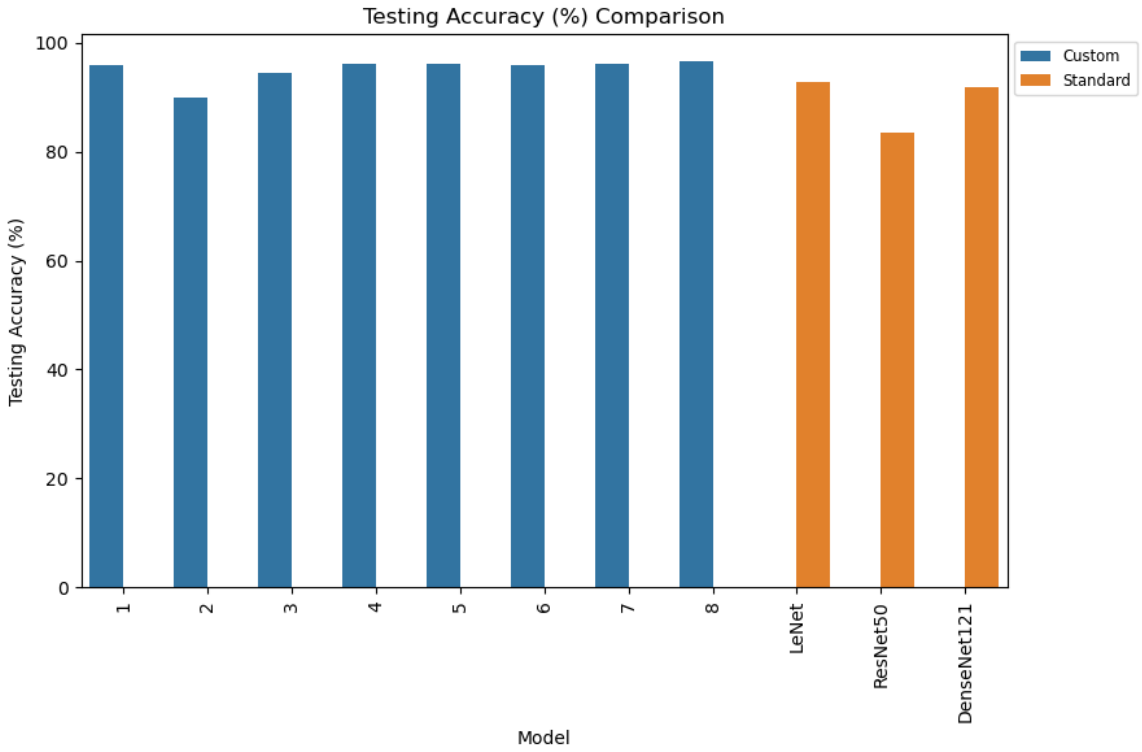


Figure 14: Testing Accuracy comparison.

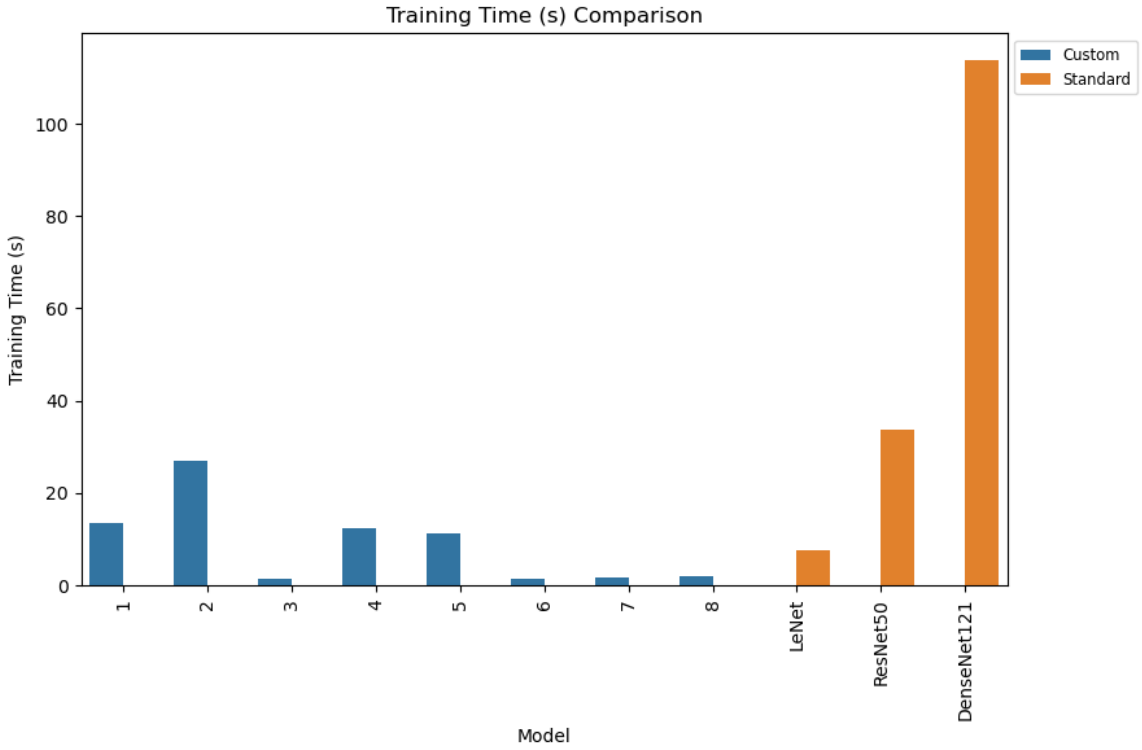


Figure 15: Training Time comparison

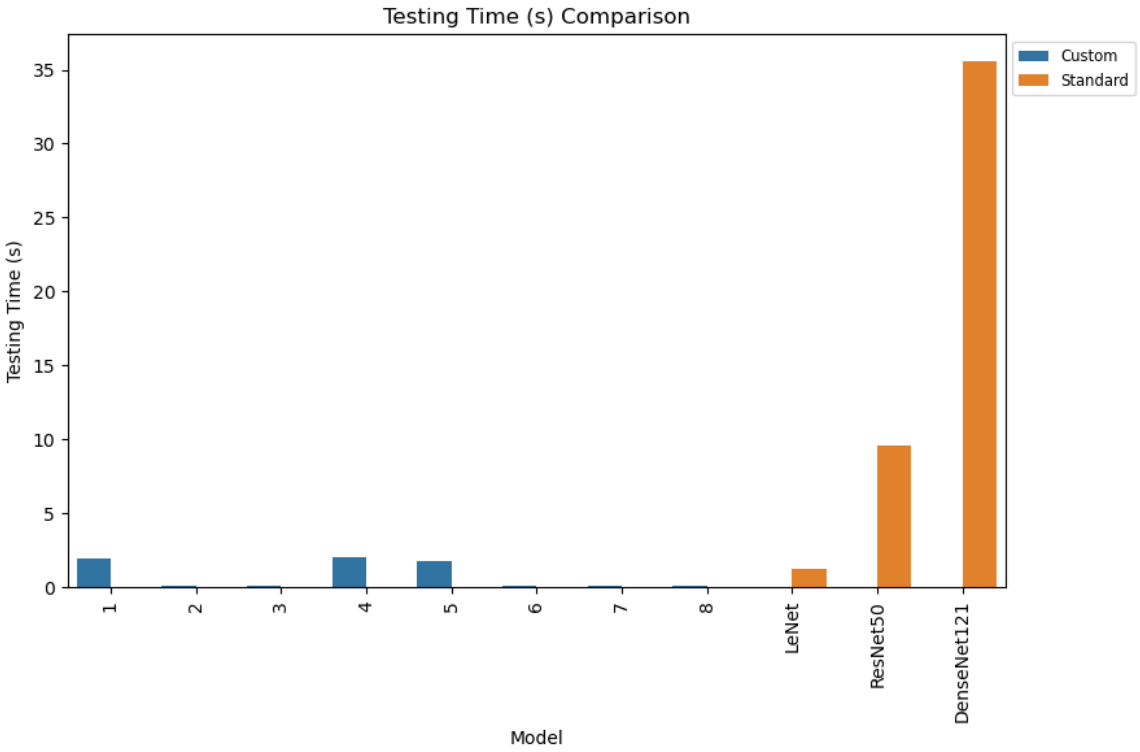


Figure 16: Testing Time comparison

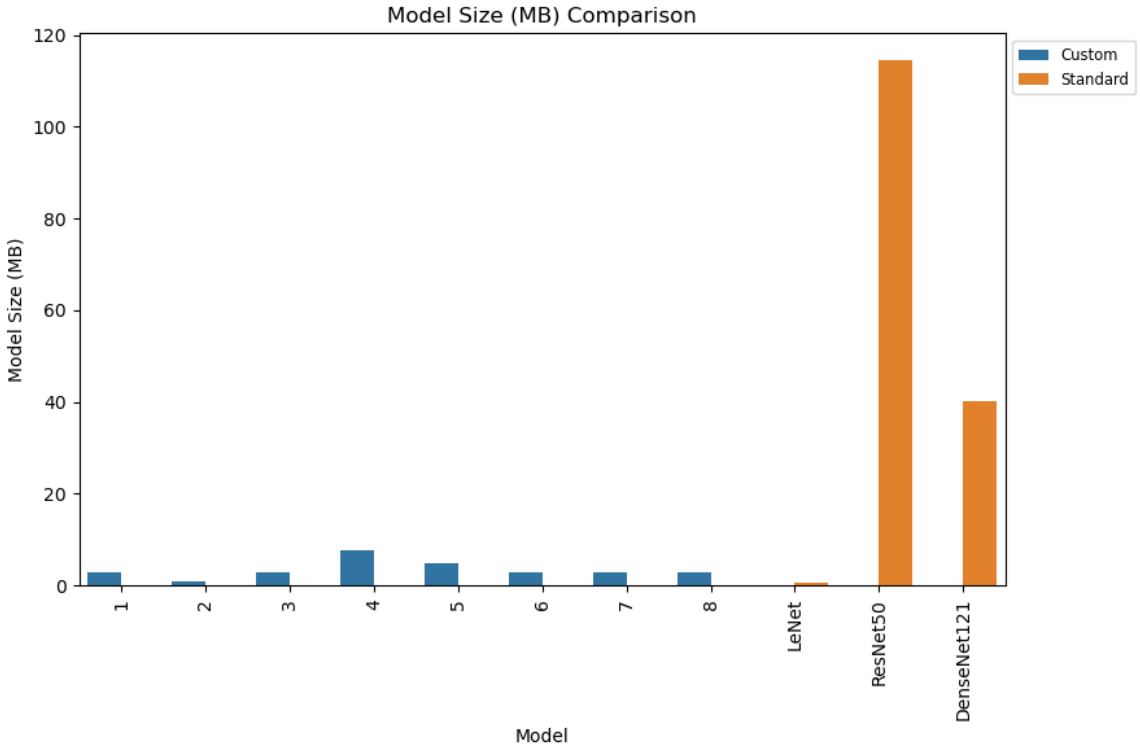


Figure 17: Model size comparison

Model	Training Time (s)	Training Accuracy (%)	Validation Accuracy (%)	Testing Time (s)	Testing Accuracy (%)	Model Size (MB)
Baseline	13.5775	95.8611	96.4167	1.96826	95.8333	2.81599
1	27.011	88.1944	92	0.105424	90.0833	0.953392
2	1.36816	93.9444	95.75	0.105299	94.4167	2.81612
3	12.407	96.7778	96.8333	2.06804	96.25	7.67359
4	11.1577	97.0278	97.0833	1.79213	96.1667	4.69694
5	1.5602	94.0556	96.6667	0.10499	95.9167	2.81632
6	1.70952	96.6944	96.5	0.104138	96.1667	2.81632
7	2.10391	98.3056	97	0.104476	96.75	2.81632
LeNet	7.61194	93.9444	93.6667	1.26797	92.9167	0.554039
ResNet50	33.813	81.3889	85	9.60473	83.5	114.607
DenseNet121	113.872	92.8333	92.9167	35.61	91.8333	40.0805

Table 1: Comparison Summary of Experiments

Model	Optimizer	Learning Rate	Number of filters	Feature map size	Dropout rate	Batch size	Epochs
Baseline	adam	0.001	[32, 64, 128]	3	0.25	128	5
Model 1	SGD	0.001	[32, 64, 128]	3	0.25	64	100
Model 2	adam	0.01	[32, 64, 128]	3	0.25	128	5
Model 3	adam	0.001	[64, 128, 256]	3	0.25	128	5
Model 4	adam	0.001	[32, 64, 128]	5	0.25	128	5
Model 5	adam	0.001	[32, 64, 128]	3	0.4	128	5
Model 6	adam	0.001	[32, 64, 128]	3	0.25	64	5
Model 7	adam	0.001	[32, 64, 128]	3	0.25	128	10
LeNet	adam	0.001	[6, 16]	Varies per layer	0	128	5
ResNet50	adam	0.001	[64, 128, 256, 512,...,]	Varies per layer	0	128	5
DenseNet121	adam	0.001	[x, 32x,,...,]	Varies per layer	0	128	5

Table 2: Comparison Summary of Models

10 Screenshots of Execution Environment

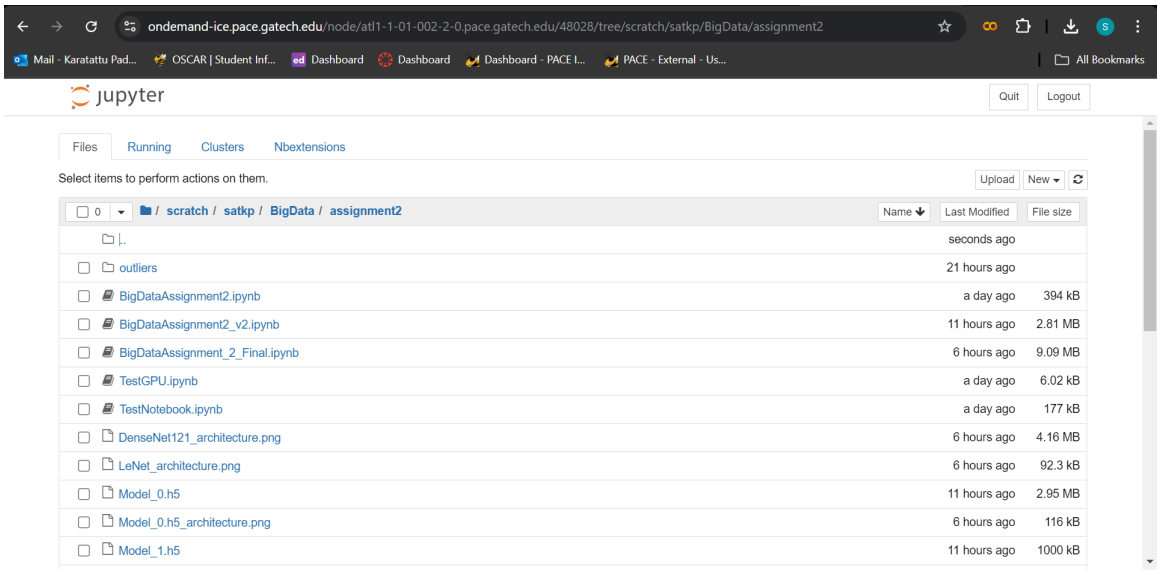


Figure 18: Jupyter Notebook directory on PACE

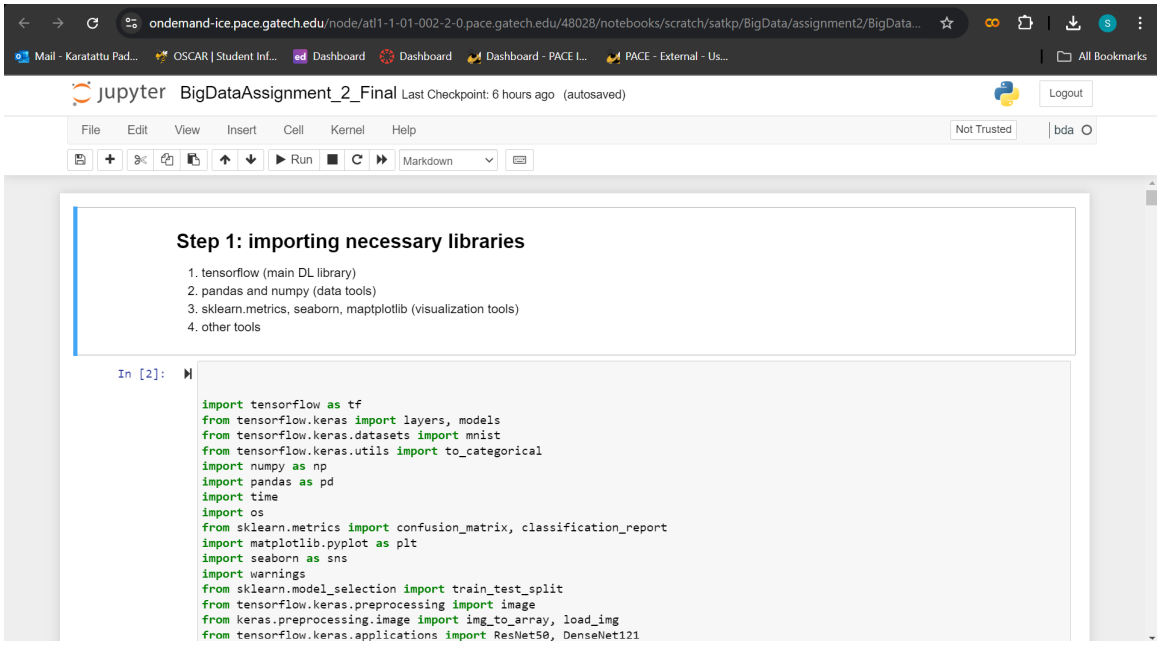


Figure 19: Notebook beginning : Note the kernel is 'bda' as mentioned in Introduction section

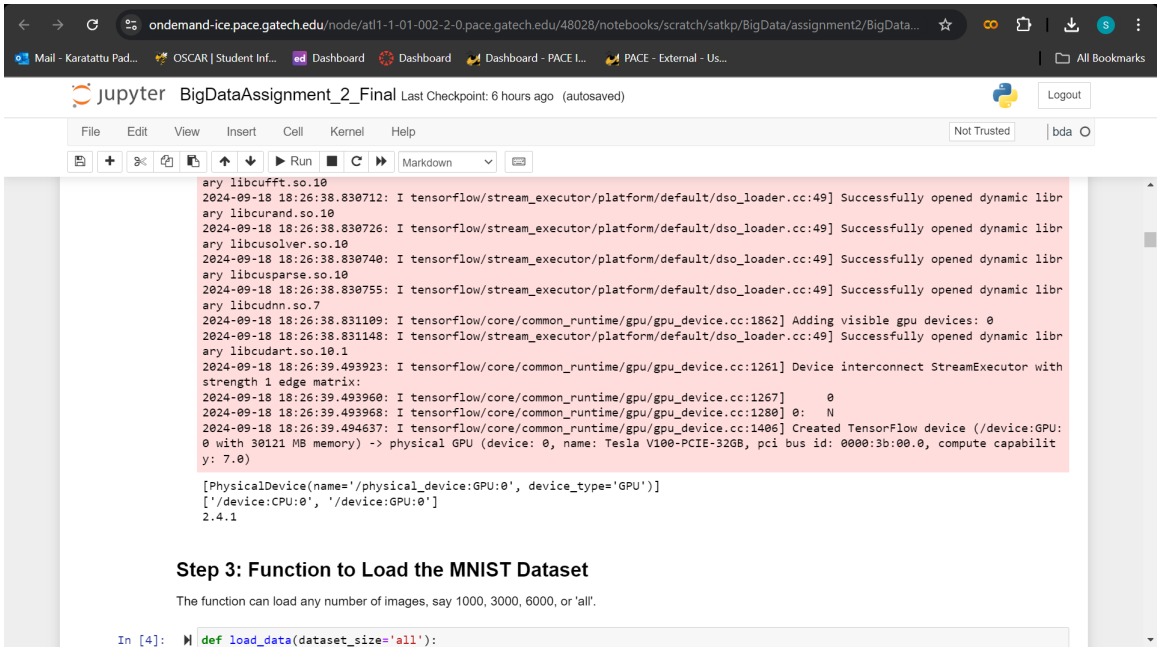


Figure 20: Showcasing the GPU

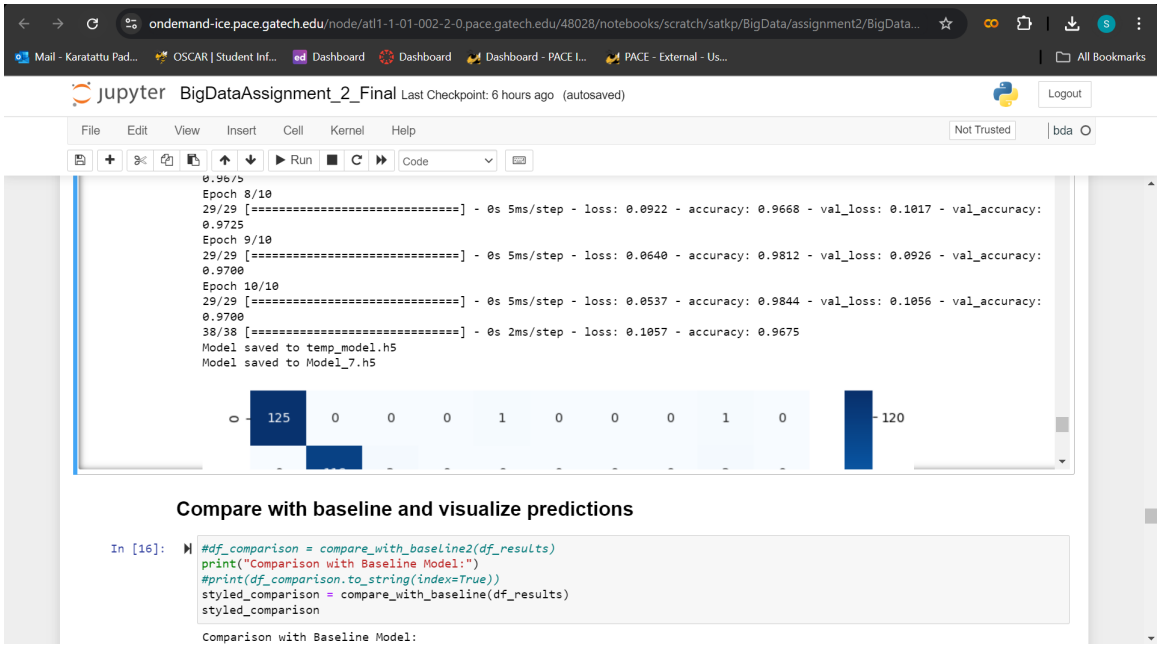


Figure 21: Execution of Training, Epochs shown in display

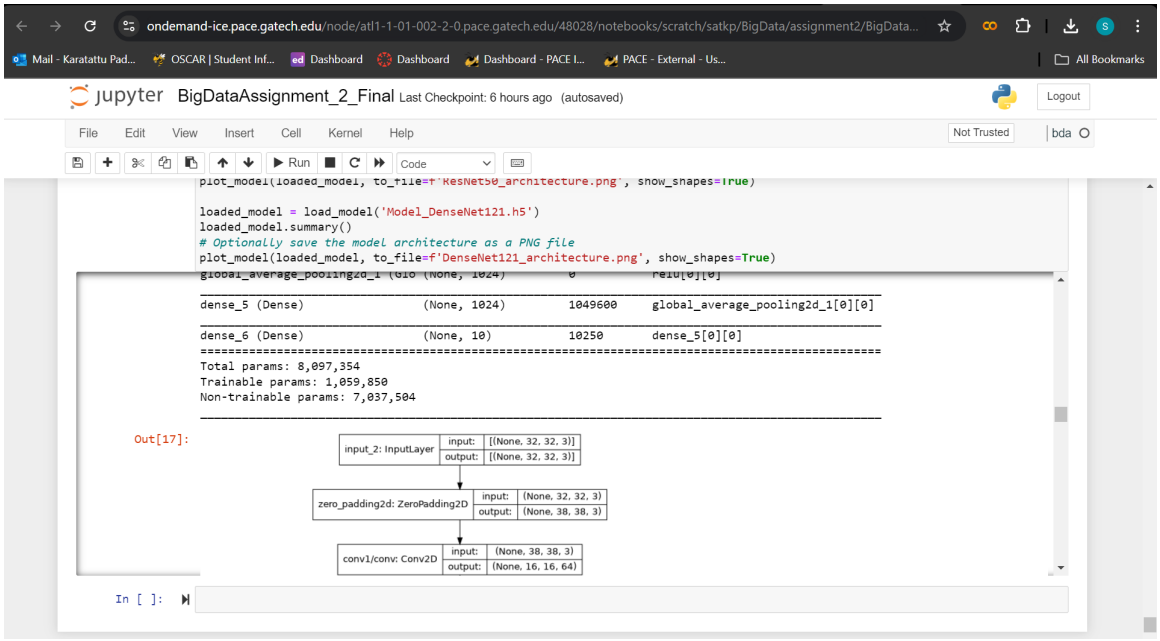


Figure 22: Printing Model weights and Structure

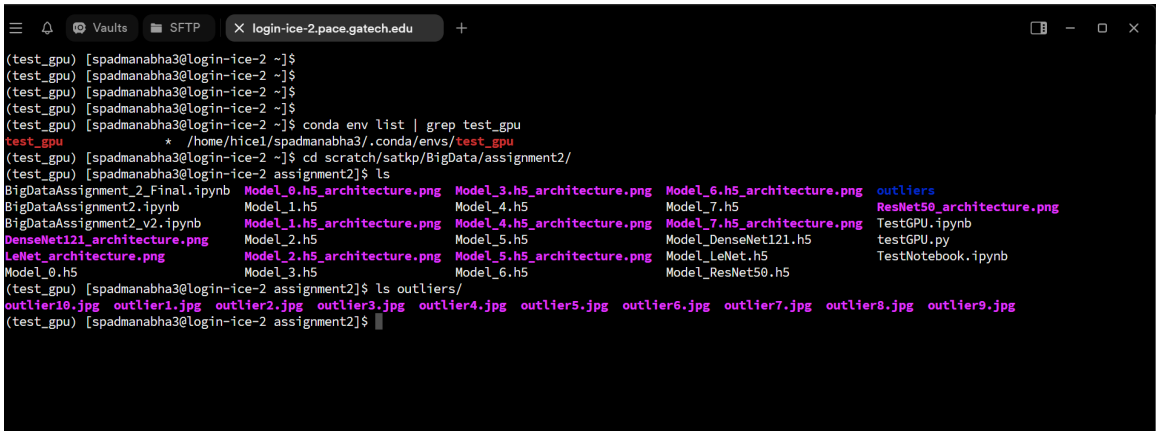


Figure 23: Conda environment, directory location with notebooks, models and architecture images viewed from a commandline tool

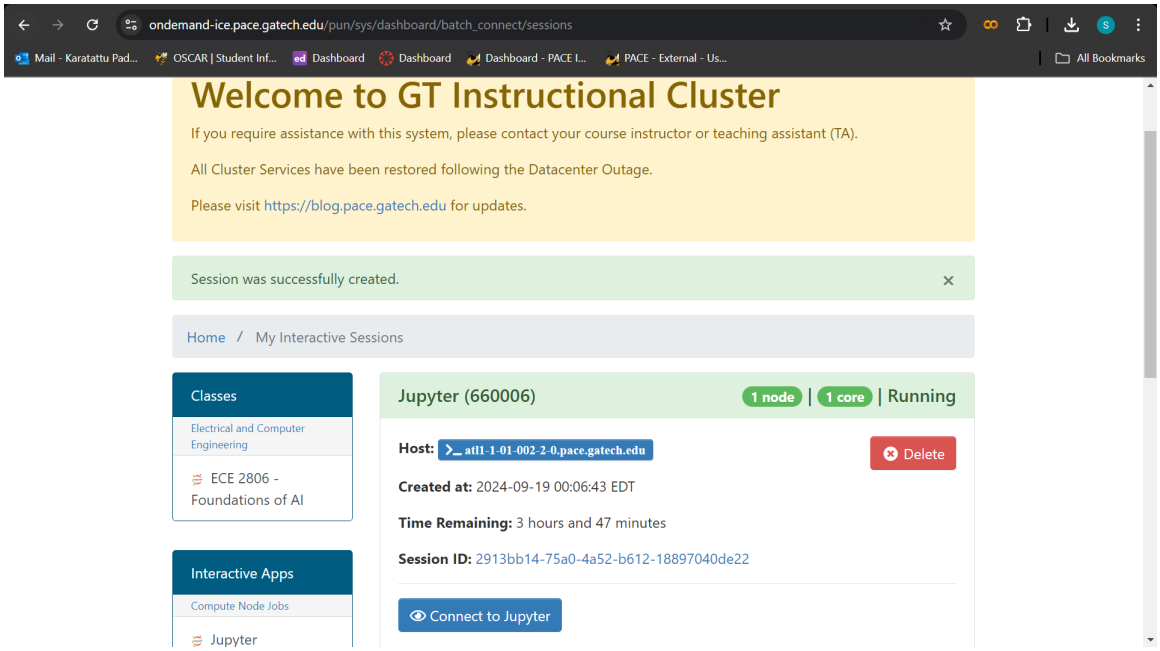


Figure 24: PACE Jupyter session creation view

11 Workflow Diagrams

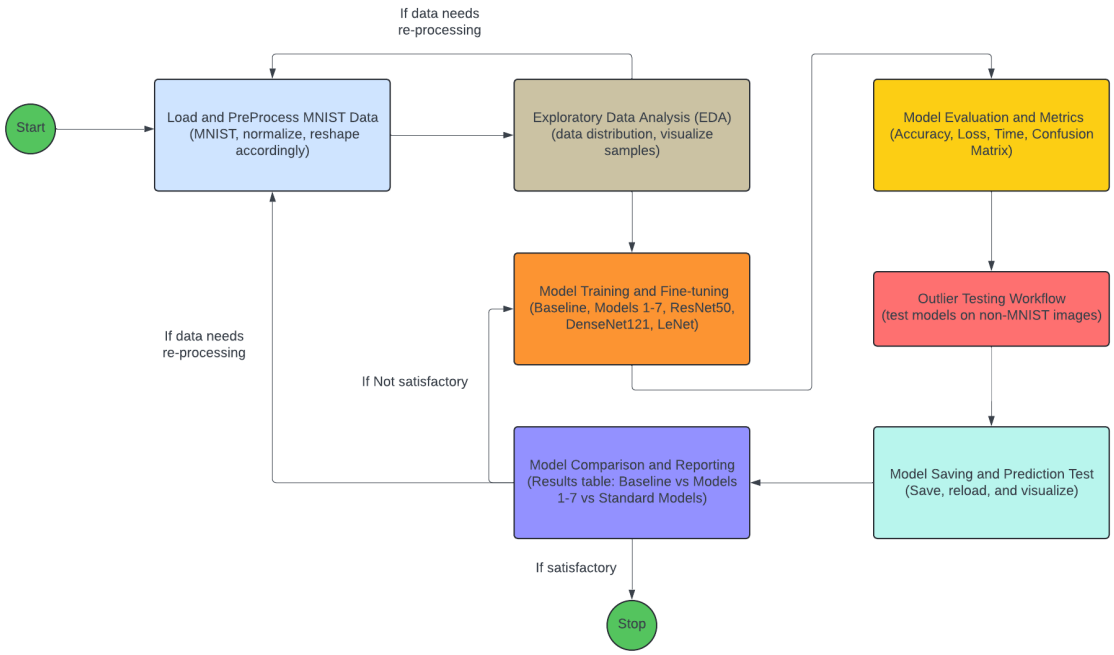


Figure 25: Overall Machine Learning Workflow Process

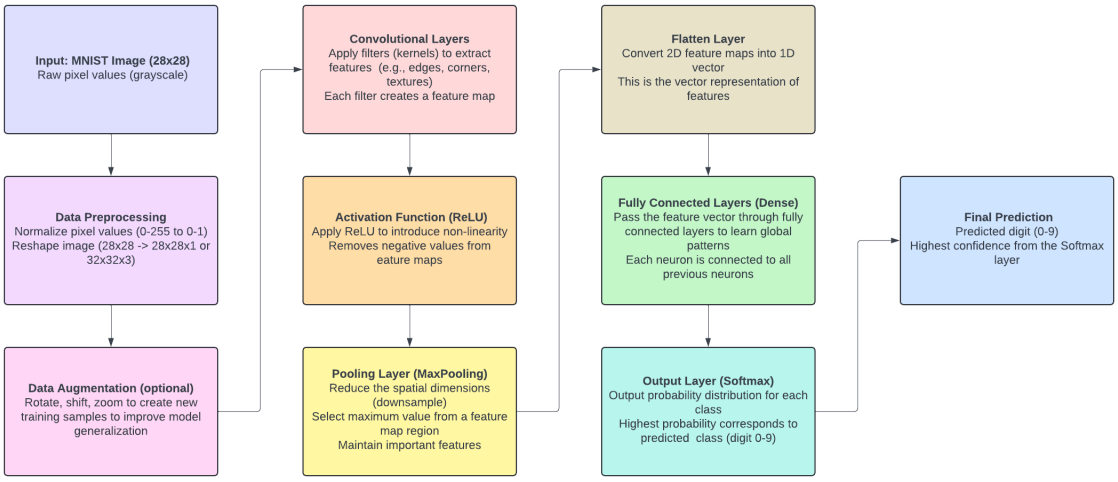


Figure 26: CNN Workflow process - input image to output prediction

12 Conclusion

- Hence, I successfully trained a CNN image classifier for the MNIST dataset where we predict the numeric character present in the image.
- I successfully wrote the code for various functions
- I was able to perform exploratory data analysis and print the input data statistics
- I successfully designed the model architecture and determined the hyper-parameters to test
- Adam optimizer is better than SGD for this task. (Note : SGD may be more optimal for other use cases)
- Low learning rate produces better results and avoids overshooting the optimal point. (Note : sometimes, we may want higher learning rate to ensure fast convergence)
- Ideal Feature map size and number of filters are determined by resource availabiltiy and use case
- batch size and dropout rates determine how robust the model is and they affect the accuracy and loss curves
- Number of epochs should be determined by the accuracy and loss curves and it also depends on all other parameters.
- While testing on outliers, different models show different outputs to the same outliers. We can observe certain trends. In this case, outliers such as alphabet characters was predicted mostly to be either '0', '2' or '8'.
- Ideally, testing the same outliers should produce different results, but as the alphabet characters were similar to numeric characters of MNIST dataset, they showed same results.
- I also tested on standard datasets; LeNet, ResNet50 and DenseNet121. Only LeNet provided similar results as our baseline model
- ResNet50 and DenseNet121 are deep complex networks used for classifying large number of classes. I imported the weights and simply fine tuned over very less number of epochs. Hence, it performed poorly for simple MNIST. But for advanced tasks, these models are State-of-the-art and will outperform our baseline model.

13 References

1. https://docs.anaconda.com/working-with-conda/environments/
2. https://medium.com/@Rahul_Meduri/install-cuda-cudnn-in-conda- virtual-environment-and-setup-gpu-support-using-tensorflow-f8a4c942b6ea
3. https://www.tensorflow.org/install/source?hl=en#gpu
4. https://www.kaggle.com/code/gpreda/simple-introduction-to-cnn-for-mnist-99-37
5. https://www.kaggle.com/code/heeraldedhia/mnist-classifier-first-deep-learning-project
6. https://www.kaggle.com/code/prashant111/mnist-deep-neural-network-with-keras
7. https://www.tensorflow.org/datasets/keras_example
8. https://en.wikipedia.org/wiki/Convolutional_neural_network
9. https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939
10. http://neuralnetworksanddeeplearning.com/chap1.html
11. https://github.com/dariodata/TensorFlow-MNIST/blob/master/TensorFlow-MNIST.ipynb
12. https://github.com/leriomaggio/deep-learning-keras-tensorflow/tree/master
13. https://github.com/pgeedh/Hyperparameter-Tuning-with-Keras-Tuner/tree/main
14. https://www.deeplearning.ai/ai-notes/optimization/index.html
15. https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac
16. https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e
17. https://www.geeksforgeeks.org/machine-learning-outlier/
18. https://aclanthology.org/2022.lrec-1.510/
19. https://www.kaggle.com/code/donatastamosauskas/using-resnet-for-mnist
20. https://www.kaggle.com/code/zhouchen1998/mnist-keras-densenet
21. https://medium.com/analytics-vidhya/lenet-with-tensorflow-a35da0d503df

14 Code

The code is present in the .zip file. Both the .ipynb and .py files are present. Make sure to have proper libraries. Refer the Introduction section for conda instructions.

```
#!/usr/bin/env python
# coding: utf-8

# ## Step 1: importing necessary libraries
#
# 1. tensorflow (main DL library)
# 2. pandas and numpy (data tools)
# 3. sklearn.metrics, seaborn, maptplotlib (visualization tools)
# 4. other tools

# In[2]:


import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np
import pandas as pd
import time
import os
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing import image
from keras.preprocessing.image import img_to_array, load_img
from tensorflow.keras.applications import ResNet50, DenseNet121
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.applications import ResNet50, DenseNet121
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Input
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.image import resize



# ## Step 2: Check for GPU
#

# In[3]:


#Check GPU and suppress warnings
print(tf.config.list_physical_devices('GPU'))
from tensorflow.python.client import device_lib
```

```
def get_available_devices():
    local_device_protos = device_lib.list_local_devices()
    return [x.name for x in local_device_protos]

print(get_available_devices())
print(tf.__version__)

warnings.filterwarnings('ignore', category=FutureWarning)


# ## Step 3: Function to Load the MNIST Dataset
#
# The function can load any number of images, say 1000, 3000, 6000, or 'all'.

# In[4]:


def load_data(dataset_size='all'):
    (train_images, train_labels), (test_images, test_labels) = mnist.load_data()

    # Combine train and test datasets for easier splitting
    all_images = np.concatenate([train_images, test_images], axis=0)
    all_labels = np.concatenate([train_labels, test_labels], axis=0)

    # Reshape and normalize the images
    all_images = all_images.reshape((all_images.shape[0], 28, 28, 1)).astype('float32') / 255

    # Convert labels to one-hot encoding
    all_labels = to_categorical(all_labels)

    if dataset_size == 'all':
        selected_images = all_images
        selected_labels = all_labels
    else:
        if not isinstance(dataset_size, int) or dataset_size <= 0:
            raise ValueError("dataset_size must be a positive integer or 'all'.")

        samples_per_class = dataset_size // 10
        selected_indices = []
        for i in range(10):
            class_indices = np.where(all_labels[:, i] == 1)[0]
            if len(class_indices) < samples_per_class:
                raise ValueError(f"Not enough samples for class {i} to meet the dataset_size require
            selected_indices.extend(class_indices[:samples_per_class])

        selected_images = all_images[selected_indices]
        selected_labels = all_labels[selected_indices]

    # Split the selected samples into train, validation, and test sets (60-20-20)
    train_images, test_images, train_labels, test_labels = train_test_split(
        selected_images, selected_labels, test_size=0.4, random_state=42
    )
    test_images, val_images, test_labels, val_labels = train_test_split(
```

```
        test_images, test_labels, test_size=0.5, random_state=42
    )

    return train_images, train_labels, val_images, val_labels, test_images, test_labels


# ## Step 4: Function for exploratory data analysis
#
# We perform exploratory data analysis for the train, test and validation datasets, we print the dat
#
# #Reference https://www.kaggle.com/code/pavansanagapati/3-layer-convolutional-neural-network-cnn

# In[5]:


# Exploratory Data Analysis (EDA)
def perform_eda(train_images, train_labels, val_images, val_labels, test_images, test_labels):
    # Data distribution across classes for train, validation, and test sets
    def plot_data_distribution(labels, dataset_name):
        class_counts = np.sum(labels, axis=0)  # Sum across one-hot encoded labels
        plt.figure(figsize=(10, 6))
        sns.barplot(x=np.arange(10), y=class_counts)
        plt.title(f'Data Distribution for {dataset_name} Set')
        plt.xlabel('Class')
        plt.ylabel('Number of Images')
        plt.show()

    def calculate_metrics(images, labels, dataset_name):
        num_images = images.shape[0]
        num_classes = labels.shape[1]
        class_counts = np.sum(labels, axis=0)
        total_size = num_images * images.shape[1] * images.shape[2] * images.shape[3]  # Total pixel
        avg_image_size = total_size / num_images
        image_size_bytes = images.nbytes  # Size in bytes
        img_height, img_width = images.shape[1], images.shape[2]

        print(f"\n{dataset_name} Set Metrics:")
        print(f"Total Number of Images: {num_images}")
        print(f"Number of Images per Class: {dict(enumerate(class_counts))}")
        print(f"Total Image Size (in pixels): {total_size}")
        print(f"Average Image Size (in pixels): {avg_image_size:.2f}")
        print(f"Total Size of Images (in bytes): {image_size_bytes / (1024 * 1024):.2f} MB")
        print(f"Image Resolution (Height x Width): {img_height} x {img_width}")

    print("Train Set Data Distribution:")
    plot_data_distribution(train_labels, 'Train')

    print("Validation Set Data Distribution:")
    plot_data_distribution(val_labels, 'Validation')

    print("Test Set Data Distribution:")
    plot_data_distribution(test_labels, 'Test')

    # Visualizing 5 images per class for the train, validation, and test sets
```

```
    def plot_images_per_class(images, labels, dataset_name):
        fig, axes = plt.subplots(10, 5, figsize=(15, 20))
        fig.suptitle(f'{dataset_name} Set: 5 Images Per Class', fontsize=16)
        for i in range(10):
            idxs = np.where(labels.argmax(axis=1) == i)[0][:5]
            for j, idx in enumerate(idxs):
                axes[i, j].imshow(images[idx].reshape(images.shape[1], images.shape[2]), cmap='gray'
                axes[i, j].axis('off')
        plt.show()

    print("Train Set: 5 Images Per Class")
    plot_images_per_class(train_images, train_labels, 'Train')

    print("Validation Set: 5 Images Per Class")
    plot_images_per_class(val_images, val_labels, 'Validation')

    print("Test Set: 5 Images Per Class")
    plot_images_per_class(test_images, test_labels, 'Test')

    # Calculate and print metrics
    calculate_metrics(train_images, train_labels, 'Train')
    calculate_metrics(val_images, val_labels, 'Validation')
    calculate_metrics(test_images, test_labels, 'Test')


# ## Step 5: CNN3 Model
#
# We create an auxillary function to create a CNN model with 3 Conv2D layers and provide option to c
#
# #Reference : https://medium.com/@BrendanArtley/mnist-keras-simple-cnn-99-6-731b624aee7f
# #Reference : https://www.kaggle.com/code/pavansanagapati/3-layer-convolutional-neural-network-cnn
# #Reference : https://www.tensorflow.org/tutorials/images/cnn

# In[6]:


# Function to create CNN model
def create_model(optimizer='adam', learning_rate=0.001, num_filters=[32, 64, 128], feature_map_size=
    # Configure optimizer with learning rate
    if optimizer == 'adam':
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer == 'sgd':
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    elif optimizer == 'rmsprop':
        optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate)
    else:
        raise ValueError("Unsupported optimizer")


    #Note to self : without padding, it will give error for [5,5] kernel!
    model = models.Sequential([
        layers.Conv2D(num_filters[0], (feature_map_size, feature_map_size), activation='relu', input
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(num_filters[1], (feature_map_size, feature_map_size), activation='relu', paddi
```

```
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(num_filters[2], (feature_map_size, feature_map_size), activation='relu', paddi
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(dropout_rate),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer=optimizer,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model


# ## Step 6: Auxillary functions for training models
#
# To maintain readablity, reusability and modularity, we define auxillary functions

# In[7]:


# Function to train and evaluate the model
def train_model(model, train_images, train_labels, val_images, val_labels, batch_size=128, epochs=10
    start_time = time.time()  # Start timer for training time
    history = model.fit(train_images, train_labels, epochs=epochs,
                        batch_size=batch_size, validation_data=(val_images, val_labels))
    train_time = time.time() - start_time  # Calculate training time
    return history, train_time

# Function to evaluate the model and compute test time
def evaluate_model(model, test_images, test_labels):
    start_time = time.time()  # Start timer for test time
    test_loss, test_accuracy = model.evaluate(test_images, test_labels)
    test_time = time.time() - start_time  # Calculate test time
    return test_accuracy, test_time

# Function to save a model
def save_model(model, model_name='saved_model.h5'):
    model.save(model_name)
    print(f"Model saved to {model_name}")

# Function to load a model
def load_model(model_name='saved_model.h5'):
    if not os.path.exists(model_name):
        raise FileNotFoundError(f"No model file found at {model_name}")
    model = tf.keras.models.load_model(model_name)
    print(f"Model loaded from {model_name}")
    return model

# Function to calculate model size in MB
def calculate_model_size(model):
    temp_file = 'temp_model.h5'
    save_model(model, temp_file)  # Save model to temporary file
    model_size = os.path.getsize(temp_file) / (1024 * 1024)  # Get size in MB
```

```
    os.remove(temp_file)  # Clean up the temporary file
    return model_size

# Function to make predictions with a model
def predict_with_model(model, data):
    predictions = model.predict(data)
    return predictions

# Function to plot the confusion matrix
def plot_confusion_matrix(test_labels, predictions, class_names):
    cm = confusion_matrix(test_labels, predictions)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_na
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()

# Function to plot training/validation accuracy and loss vs epochs
def plot_training_history(history):
    # Plot training & validation accuracy values
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')

    # Plot training & validation loss values
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')

    plt.show()


# ## Step 6: Auxillary function to calculate results
#
# We write a function that creates models based on configurations passed and evaluates it. It also r

# In[8]:


def collect_results(configurations, train_images, train_labels, val_images, val_labels, test_images,
    results = []

    for i, config in enumerate(configurations):
        print(f"Training model {i + 1} with configuration: {config['description']}")
```

```
        model = create_model(optimizer=config['optimizer'],
                             learning_rate=config['learning_rate'],
                             num_filters=config['num_filters'],
                             feature_map_size=config['feature_map_size'],
                             dropout_rate=config['dropout_rate'])

        history, train_time = train_model(model, train_images, train_labels, val_images, val_labels,
                                          batch_size=config['batch_size'], epochs=config['epochs'])

        test_accuracy, test_time = evaluate_model(model, test_images, test_labels)
        model_size = calculate_model_size(model)
        save_model(model, f'Model_{i}.h5')

        results.append({
            'Model': i + 1,
            'Description': config['description'],
            'Training Time (s)': train_time,
            'Training Accuracy (%)': history.history['accuracy'][-1] * 100,
            'Validation Accuracy (%)': history.history['val_accuracy'][-1] * 100,
            'Testing Time (s)': test_time,
            'Testing Accuracy (%)': test_accuracy * 100,
            'Model Size (MB)': model_size
        })

        predictions = model.predict(test_images).argmax(axis=1)
        test_labels_class = test_labels.argmax(axis=1)
        plot_confusion_matrix(test_labels_class, predictions, [str(i) for i in range(10)])
        plot_training_history(history)

    # Convert results to DataFrame
    df_results = pd.DataFrame(results)
    return df_results


# ## Step 7: More Auxillary functions
#
# 1. compare_with_baseline will generate a table to compare the baseline model with other models
# 2. visualize_predictions and visualize_all_models will predict images using test dataset and visua
#

# In[9]:


#To print neatly in jupyter, I used Jinja2
#Reference : https://anaconda.org/anaconda/jinja2
#Reference : https://gist.github.com/tonyfast/84f13b1460cb8fcbc43a

def compare_with_baseline(df_results):
    # Check if the baseline is present in the results
    if 'Baseline model' not in df_results['Description'].values:
        raise ValueError("Baseline model is not found in the results. Ensure the baseline model is i

    # Extract the baseline results
    baseline = df_results[df_results['Description'] == 'Baseline model'].iloc[0]
```

```
    # Initialize a list to hold comparison results
    comparison_results = []

    # Add baseline model results
    comparison_results.append({
        'Model': 'Baseline model',
        'Training Time (s)': baseline['Training Time (s)'],
        'Training Accuracy (%)': baseline['Training Accuracy (%)'],
        'Validation Accuracy (%)': baseline['Validation Accuracy (%)'],
        'Testing Time (s)': baseline['Testing Time (s)'],
        'Testing Accuracy (%)': baseline['Testing Accuracy (%)'],
        'Model Size (MB)': baseline['Model Size (MB)']
    })

    # Compare baseline model with each model individually
    for _, row in df_results.iterrows():
        if row['Description'] != 'Baseline model':
            comparison_results.append({
                'Model': row['Description'],
                'Training Time (s)': row['Training Time (s)'],
                'Training Accuracy (%)': row['Training Accuracy (%)'],
                'Validation Accuracy (%)': row['Validation Accuracy (%)'],
                'Testing Time (s)': row['Testing Time (s)'],
                'Testing Accuracy (%)': row['Testing Accuracy (%)'],
                'Model Size (MB)': row['Model Size (MB)']
            })

    # Convert comparison results to DataFrame
    df_comparison = pd.DataFrame(comparison_results)

    # Add an index column starting from 1
    df_comparison.index += 1
    df_comparison.index.name = 'Comparison Number'

    # Style the DataFrame for better readability in Jupyter notebooks
    styled_df = df_comparison.style \
        .format({'Training Time (s)': '{:.2f}',
                 'Training Accuracy (%)': '{:.2f}',
                 'Validation Accuracy (%)': '{:.2f}',
                 'Testing Time (s)': '{:.2f}',
                 'Testing Accuracy (%)': '{:.2f}',
                 'Model Size (MB)': '{:.2f}'}) \
        .set_caption('Comparison of Each Model with the Baseline Model') \
        .set_table_styles([{'selector': 'thead',
                            'props': [('background-color', '#f5f5f5'),
                                      ('font-weight', 'bold')]}]) \
        .hide(axis='index')

    return styled_df


# In[10]:
```

```
def visualize_predictions(model, test_images, test_labels, num_samples=5):
    # Predict classes for the test images
    predictions = model.predict(test_images)
    predicted_classes = np.argmax(predictions, axis=1)
    true_classes = np.argmax(test_labels, axis=1)

    # Randomly select indices for visualization
    indices = np.random.choice(len(test_images), num_samples, replace=False)

    plt.figure(figsize=(12, 12))
    for i, idx in enumerate(indices):
        plt.subplot(1, num_samples, i + 1)
        plt.imshow(test_images[idx].reshape(28, 28), cmap='gray')
        plt.title(f"True: {true_classes[idx]}\nPred: {predicted_classes[idx]}")
        plt.axis('off')
    plt.show()

def visualize_all_models(models_range, test_images, test_labels, num_samples=10):
    for i in models_range:
        # Load the model
        model_filename = f'Model_{i}.h5'
        try:
            loaded_model = load_model(model_filename)
            print(f"Visualizing predictions for {model_filename}...")
            # Visualize predictions
            visualize_predictions(loaded_model, test_images, test_labels, num_samples=num_samples)
            visualize_predictions(loaded_model, test_images, test_labels, num_samples=num_samples)
        except Exception as e:
            print(f"Failed to load or visualize {model_filename}. Error: {e}")


# ## Step 8: Outlier Testing Auxillary Functions
#
# To test the outliers, I have written some auxillary functions that predict on models generated in

# In[11]:


# 1. Define a list of outlier image file paths
outlier_file_paths = [
    'outliers/outlier1.jpg', 'outliers/outlier2.jpg', 'outliers/outlier3.jpg', 'outliers/outlier4.jp
    'outliers/outlier6.jpg', 'outliers/outlier7.jpg', 'outliers/outlier8.jpg', 'outliers/outlier9.jp
]

outlier_labels = ['a', 'b', 'zeta', 'm','e', '6_tilt','g', 'z', 'p', 'Oo']  # List of actual labels

# 5. Visualize outlier examples
def visualize_outlier_examples(file_paths, num_examples=5):
    fig, axes = plt.subplots(1, num_examples, figsize=(15, 5))
    for ax, file_path in zip(axes, file_paths[:num_examples]):
        img = image.load_img(file_path, target_size=(28, 28), color_mode='grayscale')
        ax.imshow(img, cmap='gray')
        ax.axis('off')
```

```
    plt.show()

def visualize_outliers(models, outlier_images, outlier_labels, num_samples=10):
    """
    Visualize predictions for outlier images using a list of models.

    Parameters:
    - models: List of trained models
    - outlier_images: List of file paths to outlier images
    - outlier_labels: List of true labels for outlier images
    - num_samples: Number of samples to visualize
    """
    num_models = len(models)
    num_samples = min(num_samples, len(outlier_images))
    results = []

    for i, model in enumerate(models):
        print(f"Model {i}:")
        fig, axes = plt.subplots(num_samples, 3, figsize=(15, num_samples * 5))
        fig.suptitle(f'Model {i} Predictions on Outliers', fontsize=16)

        for j in range(num_samples):
            img = load_img(outlier_images[j], target_size=(28, 28), color_mode='grayscale')
            img_array = img_to_array(img) / 255.0
            img_array = np.expand_dims(img_array, axis=0)

            # Predict
            prediction = model.predict(img_array)
            predicted_class = np.argmax(prediction)
            predicted_prob = prediction[0]

            # Display image
            axes[j, 0].imshow(img_array[0, :, :, 0], cmap='gray')
            axes[j, 0].set_title(f'Image {j + 1}')
            axes[j, 0].axis('off')

            # Display actual and predicted class
            axes[j, 1].text(0.5, 0.5, f'Actual: {outlier_labels[j]}', fontsize=12, ha='center')
            axes[j, 1].text(0.5, 0.3, f'Predicted: {predicted_class}', fontsize=12, ha='center')
            axes[j, 1].axis('off')

            # Display prediction probabilities
            axes[j, 2].bar(range(10), predicted_prob)
            axes[j, 2].set_title('Probabilities')
            axes[j, 2].set_xticks(range(10))
            axes[j, 2].set_xticklabels(range(10))
            # Prepare results
            result = {
                'Model': i,
                'Image': f'Image_{j + 1}',
                'Actual Label': outlier_labels[j],
                'Predicted Label': predicted_class,
                'Class Probabilities': predicted_prob
            }
```

```
            results.append(result)

        plt.show()

    # Create a DataFrame
    df_results = pd.DataFrame(results)
    print("Outlier Test Results:")
    pd.set_option('display.max_columns', None)  # Show all columns
    print(df_results)

    return df_results


def load_trained_models(model_paths):
    models = []
    for path in model_paths:
        model = load_model(path)
        models.append(model)
    return models


# ## Step 9: Auxillary Functions to compare with Standard Models
#
# I have compared with LeNet, Resnet50 and Densenet121. I have written auxillary functions to load,
#
# Note : For Resnet50 and Densenet121, I am using pretrained model weights and fine tuning them. I a

# In[12]:


# Preprocessing function to convert grayscale images to 32x32 RGB
def preprocess_images(images, target_shape=(32, 32)):
    # Expand dimensions to add the channel (from (28, 28) to (28, 28, 1))
    #images = np.expand_dims(images, axis=-1)
    # Resize images to target shape and replicate across 3 channels (from 1 to 3 channels)
    images_rgb = np.concatenate([resize(images, target_shape)] * 3, axis=-1)
    return images_rgb

# Function to load and modify a pretrained ResNet50 model
def load_resnet_finetune(input_shape=(32, 32, 3), num_classes=10):
    base_model = ResNet50(weights='imagenet', include_top=False, input_shape=input_shape)
    for layer in base_model.layers:
        layer.trainable = False
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation='relu')(x)
    output = Dense(num_classes, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=output)
    return model

# Function to load and modify a pretrained DenseNet121 model
def load_densenet_finetune(input_shape=(32, 32, 3), num_classes=10):
    base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=input_shape)
    for layer in base_model.layers:
```

```
        layer.trainable = False
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation='relu')(x)
    output = Dense(num_classes, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=output)
    return model

# Function to load the LeNet model
def load_lenet(input_shape=(28, 28, 1), num_classes=10):
    model = Sequential([
        Conv2D(6, kernel_size=(5, 5), activation='relu', input_shape=input_shape),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(16, kernel_size=(5, 5), activation='relu'),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(120, activation='relu'),
        Dense(84, activation='relu'),
        Dense(num_classes, activation='softmax')
    ])
    return model


# Function to train the model and track training time
def train_model_finetune(model, train_images, train_labels, val_images, val_labels, batch_size=128,
    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    start_time = time.time()
    history = model.fit(train_images, train_labels,
                        validation_data=(val_images, val_labels),
                        batch_size=batch_size, epochs=epochs, verbose=1)
    train_time = time.time() - start_time
    return history, train_time


def train_and_evaluate_standard_models():
    # Preprocess MNIST data for models expecting RGB inputs
    train_images_rgb = preprocess_images(train_images)
    val_images_rgb = preprocess_images(val_images)
    test_images_rgb = preprocess_images(test_images)

    # Models to be evaluated
    models = [
        ('LeNet', load_lenet((28, 28, 1), num_classes=10), train_images, val_images, test_images),
        ('ResNet50', load_resnet_finetune((32, 32, 3), num_classes=10), train_images_rgb, val_images
        ('DenseNet121', load_densenet_finetune((32, 32, 3), num_classes=10), train_images_rgb, val_i
    ]

    results = []

    for model_name, model, train_images_used, val_images_used, test_images_used in models:
        print(f"Training {model_name} model")
```

```
        # Train the model and track training time
        history, train_time = train_model_finetune(model, train_images_used, train_labels, val_image
                                                   batch_size=128, epochs=5)

        # Evaluate the model and track testing time
        test_accuracy, test_time = evaluate_model(model, test_images_used, test_labels)

        # Calculate model size
        model_size = calculate_model_size(model)
        save_model(model, f'Model_{model_name}.h5')

        # Collect results for this model
        results.append({
            'Model': model_name,
            'Training Time (s)': train_time,
            'Training Accuracy (%)': history.history['accuracy'][-1] * 100,
            'Validation Accuracy (%)': history.history['val_accuracy'][-1] * 100,
            'Testing Time (s)': test_time,
            'Testing Accuracy (%)': test_accuracy * 100,
            'Model Size (MB)': model_size
        })

        # Plot confusion matrix and accuracy/loss curves
        predictions = model.predict(test_images_used).argmax(axis=1)
        true_labels = test_labels.argmax(axis=1)
        plot_confusion_matrix(true_labels, predictions, [str(i) for i in range(10)])
        plot_training_history(history)

    # Convert results to a DataFrame for display
    return pd.DataFrame(results)


def visualize_comparison_stats(comparison_df):
    metrics = ['Training Accuracy (%)', 'Validation Accuracy (%)', 'Testing Accuracy (%)',
               'Training Time (s)', 'Testing Time (s)', 'Model Size (MB)']

    for metric in metrics:
        plt.figure(figsize=(12, 6))
        sns.barplot(x='Model', y=metric, hue='Source', data=comparison_df)
        plt.title(f'{metric} Comparison')
        plt.xticks(rotation=90)
        plt.legend(loc='upper left', bbox_to_anchor=(1, 1), fontsize='small')
        plt.tight_layout(rect=[0, 0, 0.75, 1])  # Make room for legend
        plt.show()


#
# # ACTUAL EXECUTION STARTS NOW!!
#
#
#

# ## Load Data and perform EDA
```

```
# In[13]:


# Load the data
train_images, train_labels, val_images, val_labels, test_images, test_labels = load_data(6000)


# In[13]:


#perform_eda(train_images, train_labels, test_images, test_labels)
perform_eda(train_images, train_labels, val_images, val_labels, test_images, test_labels)


# ## Define Configurations and train, evaluate the model
#
# 1. Baseline model
# 2. Model 1: Different optimizer (SGD)
# 3. Model 2: Different learning rate (reduced)
# 4. Model 3: More filters ([64, 128, 256])
# 5. Model 4: Larger feature map size (feature_map_size: 5)
# 6. Model 5: Higher dropout rate (0.4)
# 7. Model 6: Smaller batch size (64)
# 8. Model 7: More epochs (10)

# In[14]:


# Define configurations
configurations = [
    # Baseline model
    {'optimizer': 'adam', 'learning_rate': 0.001, 'num_filters': [32, 64, 128], 'feature_map_size':
    # Model 1: Different optimizer
    {'optimizer': 'sgd', 'learning_rate': 0.001, 'num_filters': [32, 64, 128], 'feature_map_size': 3
    # Model 2: Different learning rate
    {'optimizer': 'adam', 'learning_rate': 0.01, 'num_filters': [32, 64, 128], 'feature_map_size': 3
    # Model 3: More filters
    {'optimizer': 'adam', 'learning_rate': 0.001, 'num_filters': [64, 128, 256], 'feature_map_size':
    # Model 4: Larger feature map size
    {'optimizer': 'adam', 'learning_rate': 0.001, 'num_filters': [32, 64, 128], 'feature_map_size':
    # Model 5: Higher dropout rate
    {'optimizer': 'adam', 'learning_rate': 0.001, 'num_filters': [32, 64, 128], 'feature_map_size':
    # Model 6: Smaller batch size
    {'optimizer': 'adam', 'learning_rate': 0.001, 'num_filters': [32, 64, 128], 'feature_map_size':
    # Model 7: More epochs
    {'optimizer': 'adam', 'learning_rate': 0.001, 'num_filters': [32, 64, 128], 'feature_map_size':
]


# In[15]:


# Collect and display results
df_results = collect_results(configurations, train_images, train_labels, val_images, val_labels, tes
```

```
print(df_results)


# ## Compare with baseline and visualize predictions

# In[16]:


#df_comparison = compare_with_baseline2(df_results)
print("Comparison with Baseline Model:")
#print(df_comparison.to_string(index=True))
styled_comparison = compare_with_baseline(df_results)
styled_comparison


# In[17]:


loaded_model = load_model('Model_6.h5')

visualize_all_models(range(8), test_images, test_labels, num_samples=15)


# ## Outlier Testing
#
# I have used 10 outlier characters from cyrillic-mnist. It is characters instead of numbers.

# In[14]:


visualize_outlier_examples(outlier_file_paths, 10)

# List of saved model paths
model_paths = [f'Model_{i}.h5' for i in range(8)]  # Adjust if you have a different number of models

# Load trained models
models = load_trained_models(model_paths)


# Visualize predictions for outliers
visualize_outliers(models, outlier_file_paths, outlier_labels, num_samples=5)


# ## Train and Evaluate with standard models, Compare with earlier results
#
# 1. Lenet
# 2. Resnet50
# 3. Densenet121

# In[23]:


std_results = train_and_evaluate_standard_models()
print(std_results)
```

```
# In[20]:


# Example columns for consistency
common_columns = [
    'Model', 'Training Time (s)', 'Training Accuracy (%)', 'Validation Accuracy (%)',
    'Testing Time (s)', 'Testing Accuracy (%)', 'Model Size (MB)'
]

df_results = df_results[common_columns]
std_results = std_results[common_columns]

# Ensure consistent columns and add a source identifier
df_results = df_results[common_columns].copy()
df_results['Source'] = 'Custom'

std_results = std_results[common_columns].copy()
std_results['Source'] = 'Standard'

# Concatenate DataFrames
comparison_df = pd.concat([df_results, std_results], ignore_index=True)

# Display comparison results
print(comparison_df.to_markdown(index=False))


# In[21]:


visualize_comparison_stats(comparison_df)


# In[15]:


from tensorflow.keras.utils import plot_model

def plot_models_new(models_range):
    # Ensure that models_range is iterable
    if isinstance(models_range, int):
        models_range = range(models_range)  # Convert to range if an integer is passed

    for i in models_range:
        # Load the model
        model_filename = f'Model_{i}.h5'
        try:
            loaded_model = load_model(model_filename)
            print(f"Visualizing predictions for {model_filename}...")

            # Display the model architecture
            loaded_model.summary()
```

```
            # Optionally save the model architecture as a PNG file
            plot_model(loaded_model, to_file=f'{model_filename}_architecture.png', show_shapes=True)
        except Exception as e:
            print(f"Failed to load or visualize {model_filename}. Error: {e}")



# In[16]:


plot_models_new(8)


# In[17]:


loaded_model = load_model('Model_LeNet.h5')
loaded_model.summary()
# Optionally save the model architecture as a PNG file
plot_model(loaded_model, to_file=f'LeNet_architecture.png', show_shapes=True)


loaded_model = load_model('Model_ResNet50.h5')
loaded_model.summary()
# Optionally save the model architecture as a PNG file
plot_model(loaded_model, to_file=f'ResNet50_architecture.png', show_shapes=True)

loaded_model = load_model('Model_DenseNet121.h5')
loaded_model.summary()
# Optionally save the model architecture as a PNG file
plot_model(loaded_model, to_file=f'DenseNet121_architecture.png', show_shapes=True)


# In[ ]:
```