Project Title: Kaggle Competition 1 IFT 6390/3395

Full name: **Rajesh Kumar Raju**

Student number: **20241035**

Kaggle username**: rajeshkumarraju**

Introduction:

The goal of the Kaggle competition is to is to design machine learning (ML) algorithms that can automatically train a dataset of images and classify the images. The images contain two digits and are of dimensions 26 x 56. The class for the image is the sum of the digits in the image. For example, if an image contains the digits 8 and 9, the class is 17. The training dataset consists of 50,000 samples and test dataset consists of 10,000 examples and the datasets were generated by modifying the existing MNIST dataset. I have split the dataset into train and validation sets with a ration of 0.80 and 0.20 respectively, resulting in a training dataset of 40,000 examples and a validation dataset of 10,000 examples.

I have used three ML classifier models for this competition:

- Logistic Regression (LR)
- Random Forest (RF)
- Convolutional Neural Networks (CNN)

Out of the three models that I have built, CNN outperforms LR and RF. The overall performance of the models follows the following order: CNN > RF > LR. With extensive hyperparameter search, the maximum training and validation accuracy achieved for my LR model was 0.2730 and 0.2175, respectively. The model could beat the TA baselines in the Kaggle competition when trained on the entire dataset of 50,000 examples with the same hyperparameters and received a private score of 0.2194 and public score of 0.2136. For CNN, my best model shows a validation accuracy of 0.9863 with only 137 wrong predictions on the validation dataset with 10,000 examples. The mode received a public score of 0.9968 and private score of 0.9958 in the Kaggle competition when trained on the entire dataset. For RF, I have two models with validation accuracies of 0.7813 and 0.7783. The first model achieved a private score of 0.7304 and a public score of 0.7288 in the Kaggle completion trained on the entire training dataset examples. This model was built on the hyperparameter grid search with a fewer number of parameters. The second model which was built on the hyperparameter grid search with a larger number of parameters shows a private score of 0.7834 and a public score of 0.7830. I submitted the test predictions based on the second RF model as a late submission in the Kaggle competition to see how accurate was my second RF model compared to the first model.

Feature Design:

Each image consists of 1568 pixels that represent the features of the two digits. The maximum and minimum values for the features in the training dataset is 0.9961 and -2.1097e-16 and a similar maximum and minimum values of 0.9961 and -1.7145e-16 for the test dataset. For the normal dataset we perform normalization by dividing the pixel values by 255, the maximum value of the features. However, for the training and test dataset I did not perform such an operation as the values are already normalized between zero and 1. Technically, I have found a very small negative values such as -2.1097e-16 which essentially corresponds to a zero value. During feature engineering, I have converted all these negligibly small negative values to zero. For logistic regression (LR), I have tested two methods: with and without converting small negative values to zero. I have found LR converges to the same point and gave same training accuracy for different sets of hyper parameters. Based on this observation, I have not converted negative values to zero for other ML models and used the entire feature values as such.

Algorithms:

The first model that I have developed is Logistic regression muti-class classifier model. Logistic regression is a classification algorithm and multinomial logistic regression algorithm is an extension to the binary class logistic regression model that involves changing the loss function to cross-entropy loss and predict probability distribution to a multinomial probability distribution to natively support multi-class classification problems. In multiclass logistic regression, a softmax function is used which provide the probability value for each class label. Overfitting of the logistic regression model can be reduced by adding a penalty term, the coefficient of this penalty term is one of the hyperparameter to tune in the logistic regression. The regularization term forces the model to seek smaller model weights by adding weighted sum of the model coefficients to the loss function, encouraging the model to reduce the size of the weights along with the error while fitting the model. L2 and L1 penality namely Ridge regression and Lasso Regression are two regularization techniques that commonly employed. Ridge regression adds "*squared magnitude*" of coefficient as penalty term to the loss function whereas Lasso Regression adds "*absolute value of magnitude*" of coefficient as penalty term to the loss function. In my model, I have used the ridge regression method for regularization. The lambda λ value, the weighting of the coefficients can be used that reduces the strength of the penalty from full penalty to a very slight penalty.

Second model that I have tried was Convolutional Neural Network which is a class of deep learning methods which has become dominant in various computer vision tasks and is attracting interest across a variety of domains, including radiology. CNN is designed to learn spatial hierarchies of features automatically and adaptively through backpropagation by using multiple building blocks, such as convolution layers, pooling layers, and fully connected layers.

Third model that I have built for the project was Random Forest Classifier. Random forest classifier consists of a large number of individual decision trees that operates as an ensemble. Random forests create decision trees on randomly selected data samples, and each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction.

Methodology:

[A] For logistic regression, the entire training dataset that consists of 50000 training examples was divided into train and validation sets with 80% random training examples were allocated to training dataset and 20% of the training examples were allocated to validation dataset. For controlling the overfitting of the data, regularization parameters via L2 Norm or L2 penalty (Ridge regression) were introduced in the model. The λ parameter controls how much emphasis is given to the penalty term and is the coefficient of the regularization term. The higher the λ value, the more coefficients in the regression will be pushed towards zero. There are three hyperparameters to tune namely learning rate, lambda and epochs. Initial hyperparameters searchers were performed with 2000 epochs. For the hyperparameter search, we have used the following parameter sets:
Learning rate lr: [0.001, 0.005, 0.01, 0.05, 0.1]
Lambda (λ): [0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000]

[B] For CNN, I have tried several models by changing the number of convolutional layers (Conv2D) as well as changing the number of filters in the CNN model. Moreover, I have also adopted various regularization strategies by including dropout layers and batch normalization layers. For a limited number of models, I have also studied the effects of padding and stride options. Models were tested for their performance on the validation dataset which consists of 20% of the training examples (40,000 examples) and for best performing models, the model was retrained with the entire training dataset of 50,000 examples.

[C] Random Forest:
Hyperparameter searchers performed for random forest model RandomizedSearchCV method. I have used a the following hyperparameters in the random search:
n_estimators = [100, 200, 300]; max_features = ['auto', 'sqrt'],; max_depth = [10, 30, 60, 90, None]; min_samples_split = [2, 5, 10]; min_samples_leaf = [1, 2, 4]; bootstrap = [True, False]; criterion=['gini', 'entropy']

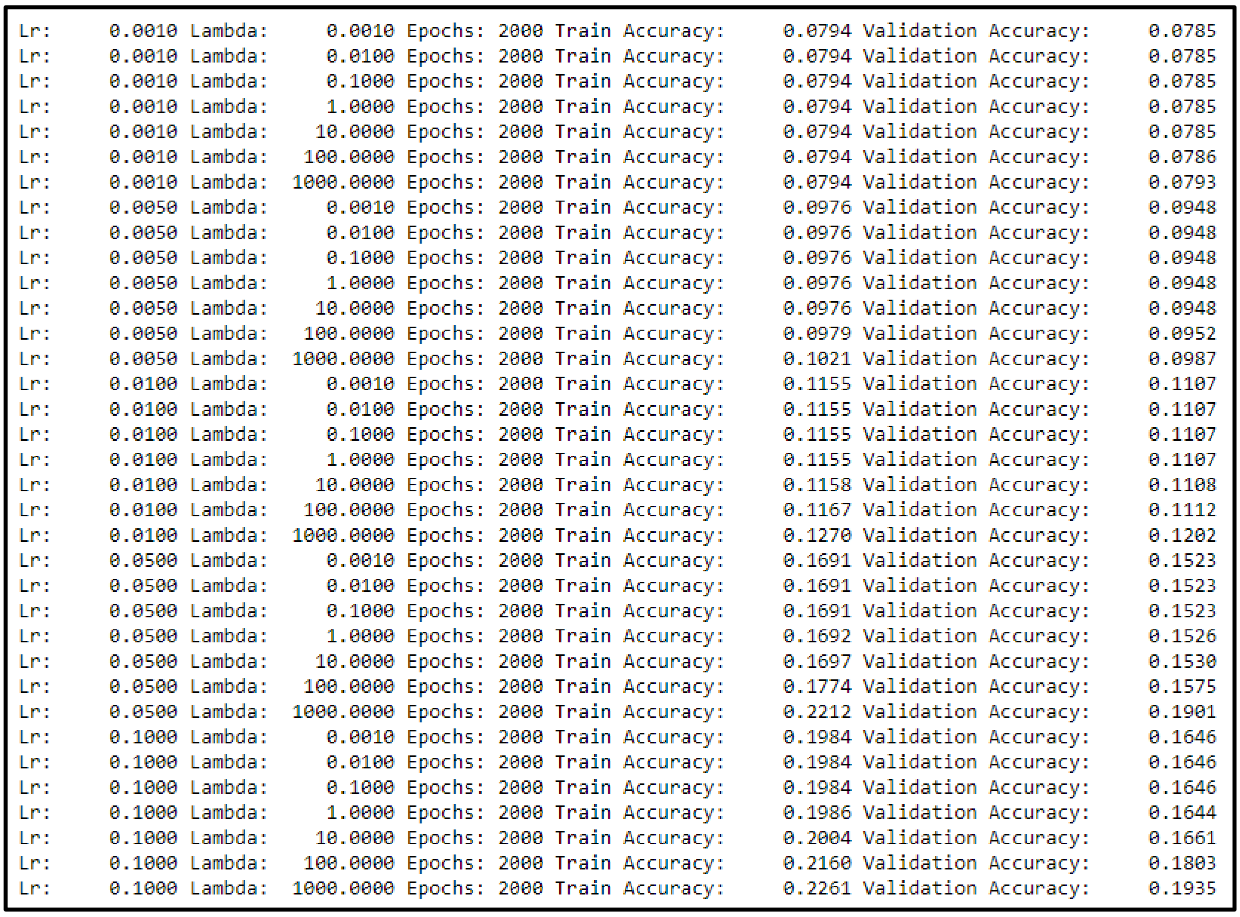
Later, I have performed the random grid search with more number of parameter values on a HPC cluster where I have access. The grid search was performed for the following parameter values:
n_estimators = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]; max_features = ['auto', 'sqrt'],;
max_depth = [10, 20, 30, 40, 50, 60,70, 80, 90, 100, 120, 150, 200, None]; min_samples_split = [2, 5, 10, 20, 30]; min_samples_leaf = [1, 2, 4, 10, 20, 30]; bootstrap = [True, False]; criterion=['gini', 'entropy']

The grid search was performed on training dataset of 40,000 examples. The best parameters from the grid search was used for predicting the validation dataset. The entire dataset of 50,000 examples were then retrained with the best hyperparameters and used for prediction on the test dataset. I have used a 3-fold cross-validation for the first grid search and 10-fold cross validation for the second grid search.

Results:

[A] Logistic Regression (LR):

Figure 1 shows the jupternotebook output for the logistic regression hyperparameter search. We have found that for our LR model, the training accuracy and validation accuracy increases as the learning rate increases. The increased accuracy is achieved with the learning rate 0.1 and lambda values 100 and 1000 for 2000 epochs training. I have also made further analyses and found that the training is not converged with 2000 epochs. Based on this, I have selected the learning rate 0.1 and trained the model for λ values 100 and 1000 for 5000 and 8000 epochs and the results are shown in Figure 2. I found that for extended epochs, the training and validation accuracy decreases for the λ value 1000. From these results, I have decided to proceed with the learning rate value of 0.1 and the regularization parameter (λ) value of 100.0 which shows a training accuracy of 0.2613 and 0.2730 and a validation accuracy of 0.2095 and 0.2175 for epochs 5000 and 8000 respectively.

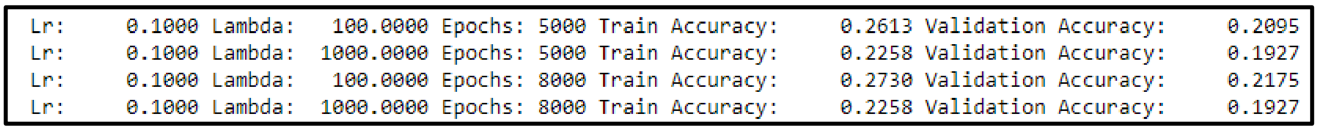


```
Lr:     0.0010 Lambda:     0.0010 Epochs: 2000 Train Accuracy:     0.0794 Validation Accuracy:     0.0785
Lr:     0.0010 Lambda:     0.0100 Epochs: 2000 Train Accuracy:     0.0794 Validation Accuracy:     0.0785
Lr:     0.0010 Lambda:     0.1000 Epochs: 2000 Train Accuracy:     0.0794 Validation Accuracy:     0.0785
Lr:     0.0010 Lambda:     1.0000 Epochs: 2000 Train Accuracy:     0.0794 Validation Accuracy:     0.0785
Lr:     0.0010 Lambda:    10.0000 Epochs: 2000 Train Accuracy:     0.0794 Validation Accuracy:     0.0785
Lr:     0.0010 Lambda:   100.0000 Epochs: 2000 Train Accuracy:     0.0794 Validation Accuracy:     0.0786
Lr:     0.0010 Lambda:  1000.0000 Epochs: 2000 Train Accuracy:     0.0794 Validation Accuracy:     0.0793
Lr:     0.0050 Lambda:     0.0010 Epochs: 2000 Train Accuracy:     0.0976 Validation Accuracy:     0.0948
Lr:     0.0050 Lambda:     0.0100 Epochs: 2000 Train Accuracy:     0.0976 Validation Accuracy:     0.0948
Lr:     0.0050 Lambda:     0.1000 Epochs: 2000 Train Accuracy:     0.0976 Validation Accuracy:     0.0948
Lr:     0.0050 Lambda:     1.0000 Epochs: 2000 Train Accuracy:     0.0976 Validation Accuracy:     0.0948
Lr:     0.0050 Lambda:    10.0000 Epochs: 2000 Train Accuracy:     0.0976 Validation Accuracy:     0.0948
Lr:     0.0050 Lambda:   100.0000 Epochs: 2000 Train Accuracy:     0.0979 Validation Accuracy:     0.0952
Lr:     0.0050 Lambda:  1000.0000 Epochs: 2000 Train Accuracy:     0.1021 Validation Accuracy:     0.0987
Lr:     0.0100 Lambda:     0.0010 Epochs: 2000 Train Accuracy:     0.1155 Validation Accuracy:     0.1107
Lr:     0.0100 Lambda:     0.0100 Epochs: 2000 Train Accuracy:     0.1155 Validation Accuracy:     0.1107
Lr:     0.0100 Lambda:     0.1000 Epochs: 2000 Train Accuracy:     0.1155 Validation Accuracy:     0.1107
Lr:     0.0100 Lambda:     1.0000 Epochs: 2000 Train Accuracy:     0.1155 Validation Accuracy:     0.1107
Lr:     0.0100 Lambda:    10.0000 Epochs: 2000 Train Accuracy:     0.1158 Validation Accuracy:     0.1108
Lr:     0.0100 Lambda:   100.0000 Epochs: 2000 Train Accuracy:     0.1167 Validation Accuracy:     0.1112
Lr:     0.0100 Lambda:  1000.0000 Epochs: 2000 Train Accuracy:     0.1270 Validation Accuracy:     0.1202
Lr:     0.0500 Lambda:     0.0010 Epochs: 2000 Train Accuracy:     0.1691 Validation Accuracy:     0.1523
Lr:     0.0500 Lambda:     0.0100 Epochs: 2000 Train Accuracy:     0.1691 Validation Accuracy:     0.1523
Lr:     0.0500 Lambda:     0.1000 Epochs: 2000 Train Accuracy:     0.1691 Validation Accuracy:     0.1523
Lr:     0.0500 Lambda:     1.0000 Epochs: 2000 Train Accuracy:     0.1692 Validation Accuracy:     0.1526
Lr:     0.0500 Lambda:    10.0000 Epochs: 2000 Train Accuracy:     0.1697 Validation Accuracy:     0.1530
Lr:     0.0500 Lambda:   100.0000 Epochs: 2000 Train Accuracy:     0.1774 Validation Accuracy:     0.1575
Lr:     0.0500 Lambda:  1000.0000 Epochs: 2000 Train Accuracy:     0.2212 Validation Accuracy:     0.1901
Lr:     0.1000 Lambda:     0.0010 Epochs: 2000 Train Accuracy:     0.1984 Validation Accuracy:     0.1646
Lr:     0.1000 Lambda:     0.0100 Epochs: 2000 Train Accuracy:     0.1984 Validation Accuracy:     0.1646
Lr:     0.1000 Lambda:     0.1000 Epochs: 2000 Train Accuracy:     0.1984 Validation Accuracy:     0.1646
Lr:     0.1000 Lambda:     1.0000 Epochs: 2000 Train Accuracy:     0.1986 Validation Accuracy:     0.1644
Lr:     0.1000 Lambda:    10.0000 Epochs: 2000 Train Accuracy:     0.2004 Validation Accuracy:     0.1661
Lr:     0.1000 Lambda:   100.0000 Epochs: 2000 Train Accuracy:     0.2160 Validation Accuracy:     0.1803
Lr:     0.1000 Lambda:  1000.0000 Epochs: 2000 Train Accuracy:     0.2261 Validation Accuracy:     0.1935
```

Figure 1: Hyperparameter tuning outputs from the jupter notebook for the parameters: Learning rate lr: [0.001, 0.005, 0.01, 0.05, 0.1] and Lambda (λ): [0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000] trained on a train and validation datasets of 40000 and 10000 training examples for 2000 epochs.

I have retrained the model with the entire training dataset for the learning rate 0.1 and λ values 10.0, 100.0, and 1000.0 for 5000 epochs to see whether there is a significant change in the training accuracy by including more training examples. I have found that there is no significant difference in the training accuracy by including more examples. Again, the hyperparameter set [lr = 0.10, λ = 100.0] shows better performance. I have further extended the training to see whether there is an improvement in the training accuracy with extended training periods of 8000 and 12000 epochs. The training accuracy climbed to 0.26754 and 0.27228 for 8000 and 12000 epochs. Moreover, I have also performed a sanity check to see whether extended epochs of 8000 improves the performance of the model with the learning parameter 0.01 and λ = 1.0, 10.0 and 100.0 values. I found that learning rate value 0.01 do not outperform with longer epochs and 0.1 seems to be performing better for our model. The results are summarized in Table 1. The training loss curve for extended epoch runs of 12000 is shown in Figure 3.

For beating the TA baselines, I have used the test predictions based on the following hyperparameters: Lr: 0.1, Lambda (λ): 100 and Epochs 8000 on the model trained on entire training data with 50,000 examples. I could beat the TA baseline with a private score of 0.2194 and public score of 0.2136. The training loss curve for extended epoch runs of 12000 is shown in Figure 3.



```
Lr:     0.1000 Lambda:   100.0000 Epochs: 5000 Train Accuracy:     0.2613 Validation Accuracy:     0.2095
Lr:     0.1000 Lambda:  1000.0000 Epochs: 5000 Train Accuracy:     0.2258 Validation Accuracy:     0.1927
Lr:     0.1000 Lambda:   100.0000 Epochs: 8000 Train Accuracy:     0.2730 Validation Accuracy:     0.2175
Lr:     0.1000 Lambda:  1000.0000 Epochs: 8000 Train Accuracy:     0.2258 Validation Accuracy:     0.1927
```

Figure 2: Hyperparameter tuning outputs from the Jupyter notebook for the learning parameter 0.1 for the lambda (λ) values 100 and 1000 for 5000 and 8000 epochs training. The training and validation dataset consists of 40000 and 10000 examples, respectively.

Table 1: Training accuracy for the logistic regression classifier trained on the entire training dataset that consist of 50000 training examples. The hyperparameter searches were limited to the learning rates 0.1 and λ values 10, 100, 1000.

Lr	λ	Epochs	Training Accuracy	Lr	λ	Epochs	Training Accuracy
0.1	10	5000	0.23246	0.01	1.0	8000	0.15786
0.1	**100**	**5000**	**0.25434**	0.01	10	8000	0.1584
0.1	1000	5000	0.22724	0.01	100	8000	0.163
0.1	**100**	**8000**	**0.26754**				
0.1	**100**	**12000**	**0.27228**				

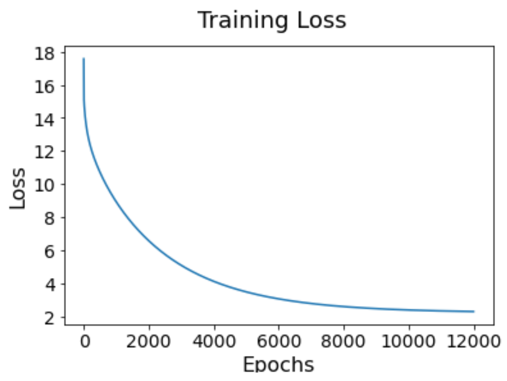


Figure 3: Training loss curve for the model with the hyperparameters lr = 0.10, λ = 100.0 trained on the full training dataset of 5000 examples for

[B] Convolutional Neural Network Models (CNN)

For CNN, I have tested several model architectures and only the most relevant models were discussed here. For all the models, I have compiled the model with "categorical cross entropy" loss function employing "Adam" optimizer. "Relu" activation was used in all the layers except on the final output layer where "softmax" function was used. I have developed a relatively simpler model abbreviated as "**Model-1**" with three convolutional 2D layers with filters 32, 64 and 128 with a kernel size of (3,3). Each of these Conv2D layers was followed by a MaxPool2D layer of pool size (2,2). It is then followed by three dense hidden layers of dimensions 128, 64, 32. The model contains a total of 185,683 trainable parameters and trained on a dataset that contains 40,000 and 10,000 training and validation examples for a total of 100 epochs with early-stopping call-backs option with respect to validation loss. The model achieved a validation prediction accuracy of 0.9820 with a total of 180 wrong predictions on the validation dataset of 10,000 examples. For the test predictions with this model, I have received a public score of 0.9754 and a private score of 0.9766 in the Kaggle submissions.

Table 2: Total number of parameters, trainable parameters and non-trainable parameters for different CNN models.

	Model-1	Model-2	Model-3	Model-4
Total Parameters	185,683	1,088,595	1,088,595	1,090,515
Trainable Parameters	185,683	1,087,699	1,087,699	1,089,619
Non-Trainable Parameters	0	896	896	896

To improve the accuracy further by controlling the overfitting, I have made further modifications to the model by including dropout and batch normalization layers. The new model is abbreviated as M**odel-2**. I have used a dropout rate of 0.3 in the model. This increased the model complexity and the total trainable parameters increased to 1,087,699. When trained with early-stopping options for a total of 100 epochs, The model acquired a validation accuracy of 0.9837 with 163 wrong predictions on the validation dataset. Comparing to Model-1, I have noticed that there is no significant difference in the model's performance on the validation dataset. I have retrained the model with the entire training dataset for a total of 100 epochs without early-stopping call-backs options (**Model-3**). For the test predictions using the trained Model-3, I have received a private score of 0.9914 and a public score of 0.9910. It seems like regularization control through batch normalization and dropout as well as the larger number of epochs without early-stopping contribute to the increased accuracy in the test predictions on Model-3 compared to test predictions based on trained Model-1.

In Model-4, I have used a different CNN architecture with five convolution 2D layers with filters 64, 64, 128, 128 and 256, each with kernel size (3,3). Maxpool2D and Batch normalization layers were added after 2 Conv2D layers and after the fifth Conv2D. Following this, a dense layer of 512 neurons were added before the last classifier layer. The model has a total trainable parameter of 1,089,619. When trained with early-stopping call-backs options for 100 epochs, the model achieved an accuracy of 0.9863 for the validation dataset of 10000 examples with only 137 wrong predictions. The model was then retrained with the entire dataset of 50000 training examples for 100 epochs without early-stopping option. I have received a private score of 0.9958 and a public score of 0.9968 for the test predictions using this trained model. The model architecture for different CNN models is shown in Appendix along with training and validation accuracy plots. The training and validation loss were shown in Figure 4.

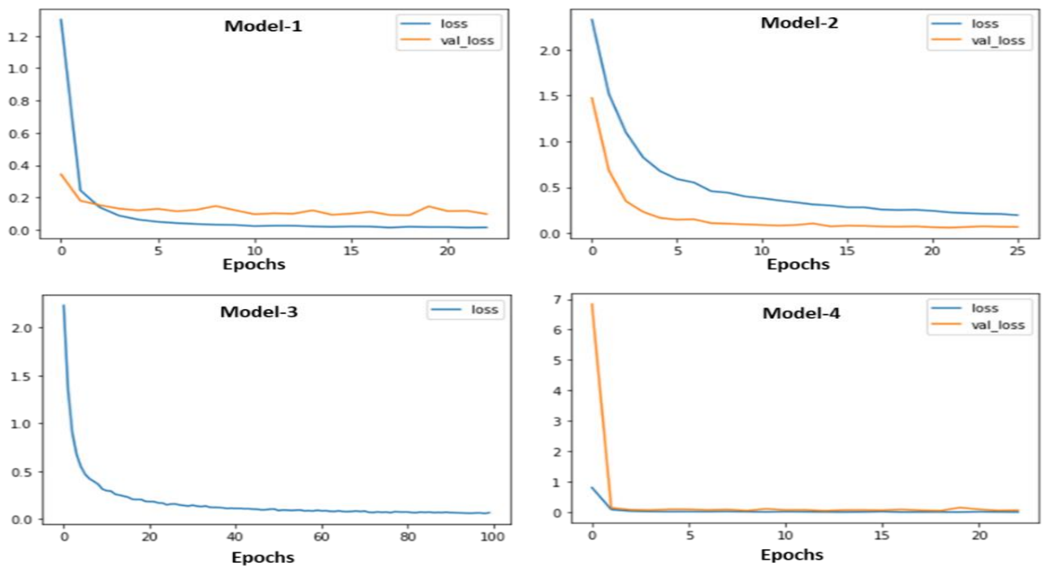


Figure 4: Training and validation loss for different CNN models. Note that Model-3 does not have validation set.

[C] Random Forest (RF)

Initially, I have trained the RF model with 250 trees in the model keeping the default parameters from the sci-kit learn. The model achieved a validation accuracy of 0.7463. Later, I have performed a randomized grid search for finding the best hyperparameters. See Methodology section for two sets of hyperparameter searchers that I have performed.

The first grid search was performed for 100, 200 and 300 trees in the random forest. The best parameter values obtained were n_estimators = 300, max_features = 'sqrt' ; max_depth = 90 ; min_samples_split = 2 ; min_samples_leaf = 1 bootstrap = False; criterion='gini'.

Based on above parameters, the model achieved a validation accuracy of 0.7813 on the validation dataset. The model was retrained on the entire dataset and used for test predictions. When submitted in the Kaggle Competition, the model achieved a private score of 0.7304 and public score of 0.7288.

The second grid search was performed on larger parameter space and obtained the following best parameters. n_estimators = 800, max_features = 'sqrt' ; max_depth = 70 ; min_samples_split = 10; min_samples_leaf = 1 bootstrap = False; criterion='entropy'.

Based on the above model, the model achieved a validation accuracy of 0.7783 when test on the validation dataset. The model then retrained on the entire training dataset and used for test predictions. The test predictions based on the second sets of RF best parameters received a private score of 0.7834 and public score of 0.783 when submitted for Kaggle competitions.
It can be seen that randomized grid search with larger number of parameter space values shows better performance compared to gridsearch parameters with smaller parameter space values.

Discussion:

Most importantly, the logistic regression model was not performing very well for the dataset provided. One reason could be a lack of appropriate feature engineering as the dataset is more complex than normal MNIST dataset as it includes images of two digits. I have built the model with the L2 penalty (Ridge Regression) as the regularization to control the overfitting of the model. L2 penalty decreases the complexity of a model but does not reduce the number of variables since it never leads to a coefficient to zero rather only minimizes it. Hence, the ridge regression is not good for feature reduction techniques. As the dataset contains 1568 features and many of these features are essentially zero, I think changing the L2 regularization to L1 regularization (Lasso Regression) is a good idea to increase the accuracy of the logistic regression model. Lasso regression will automatically select those features that are useful, discarding the useless or redundant features while trying to minimize the cost function.
I have performed extended hyperparameter searches for learning rate, λ and epochs for the logistic regression model. I do not think that further tuning on these hyperparameter values would make the model any better. Another issue with the LR model is training is very slow for the best hyperparameter values that I found and need to train the model more than 8000-10000 epochs to converge. So, the best option would be appropriate feature engineering using Lasso regression or some other feature engineering techniques to remove redundant features or non-relevant features.
The CNN models that I have built shows superior performance. Even the simplest CNN model that I have built achieved an accuracy of 0.97. Adding more layers may overfit the model. To overcome this issue, we can add regularization through dropout and batch normalization. Overall CNN model shows superior performance and very good accuracy. I did not perform tuning of dropout rate. May be this would be a good idea to find the optimal dropout layer rate. From my experience on this dataset, I believe we do not need to build a highly complex CNN model for achieving an accuracy of 0.97 or more.
For random forest (RF), I have performed an extensive search for the different hyperparameters. Still, the best parameters that I got from the randomized grid search do not show significant improvement. I have seen that defaults scikit parameter set ups for Random Forest Classifier itself shows similar performance. For instance, I have performed a test on the RF model with 250 trees with the default setting in the scikit-learn and achieved an accuracy of 0.75. With best parameters obtained after extensive random grid search, the accuracy increased to 0.78 only. I have already used a 10-fold cross validation and it seems like the model is achieved its maximum accuracy with the current dataset. As I mentioned earlier, feature engineering is a good idea to improve the accuracy of the model.

Statement of Contributions.

"I hereby state that all the work presented in this report is that of the author".

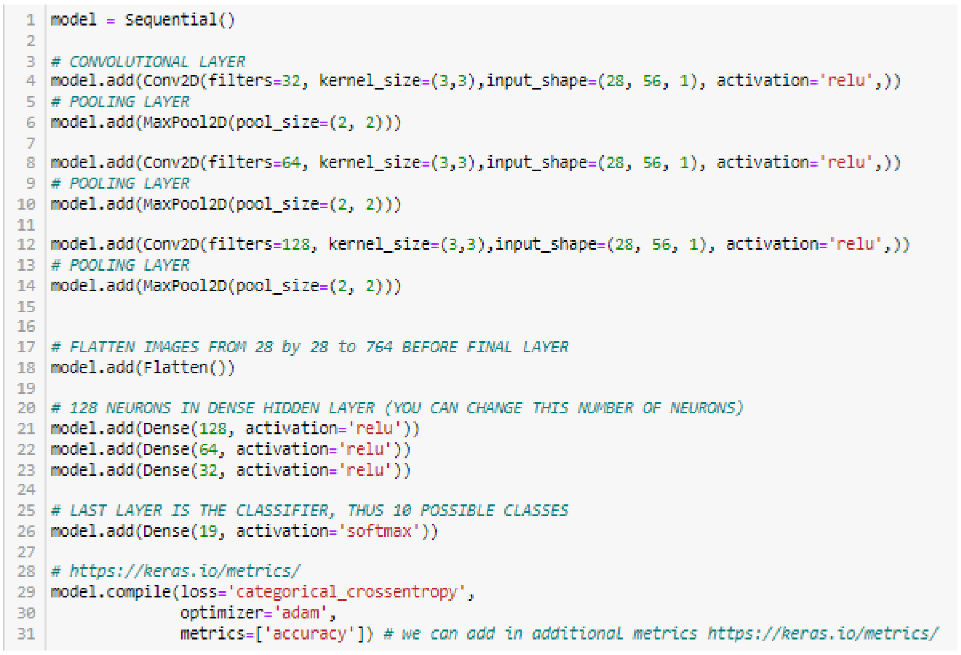
References:

I have received extensive help from the medium in understanding the models. I have not copied any particular code as a whole, Instead I took the idea from the different online resources. Some of the important medium links:

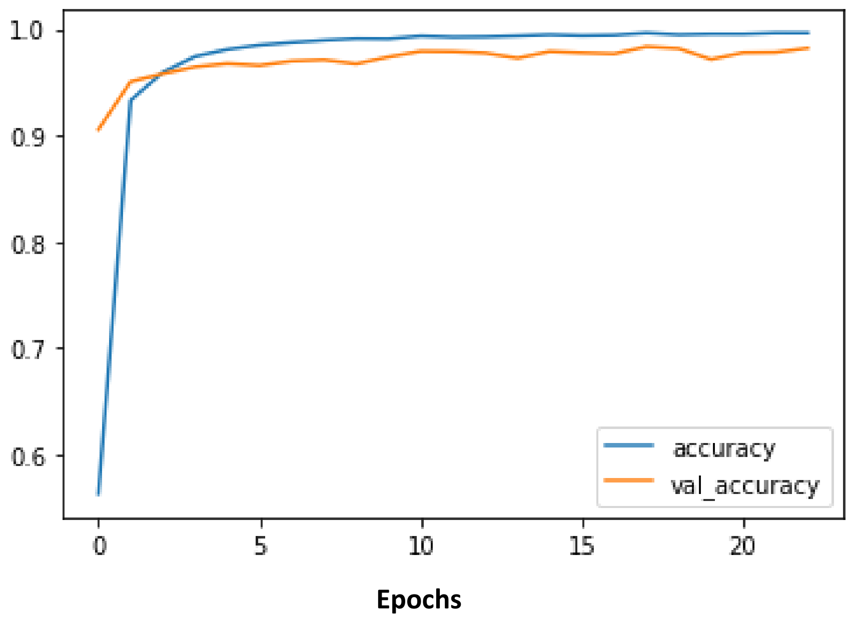
[1] https://towardsdatascience.com/mastering-random-forests-a-comprehensive-guide-51307c129cb1#:~:text=Generally%2C%20we%20go%20with%20a,vary%20between%20each%20decision%20tree.
[2] https://towardsdatascience.com/coding-a-convolutional-neural-network-cnn-using-keras-sequential-api-ec5211126875
[3] https://towardsdatascience.com/multiclass-logistic-regression-from-scratch-9cc0007da372
[4] http://rasbt.github.io/mlxtend/user_guide/classifier/SoftmaxRegression/#overview

Appendix

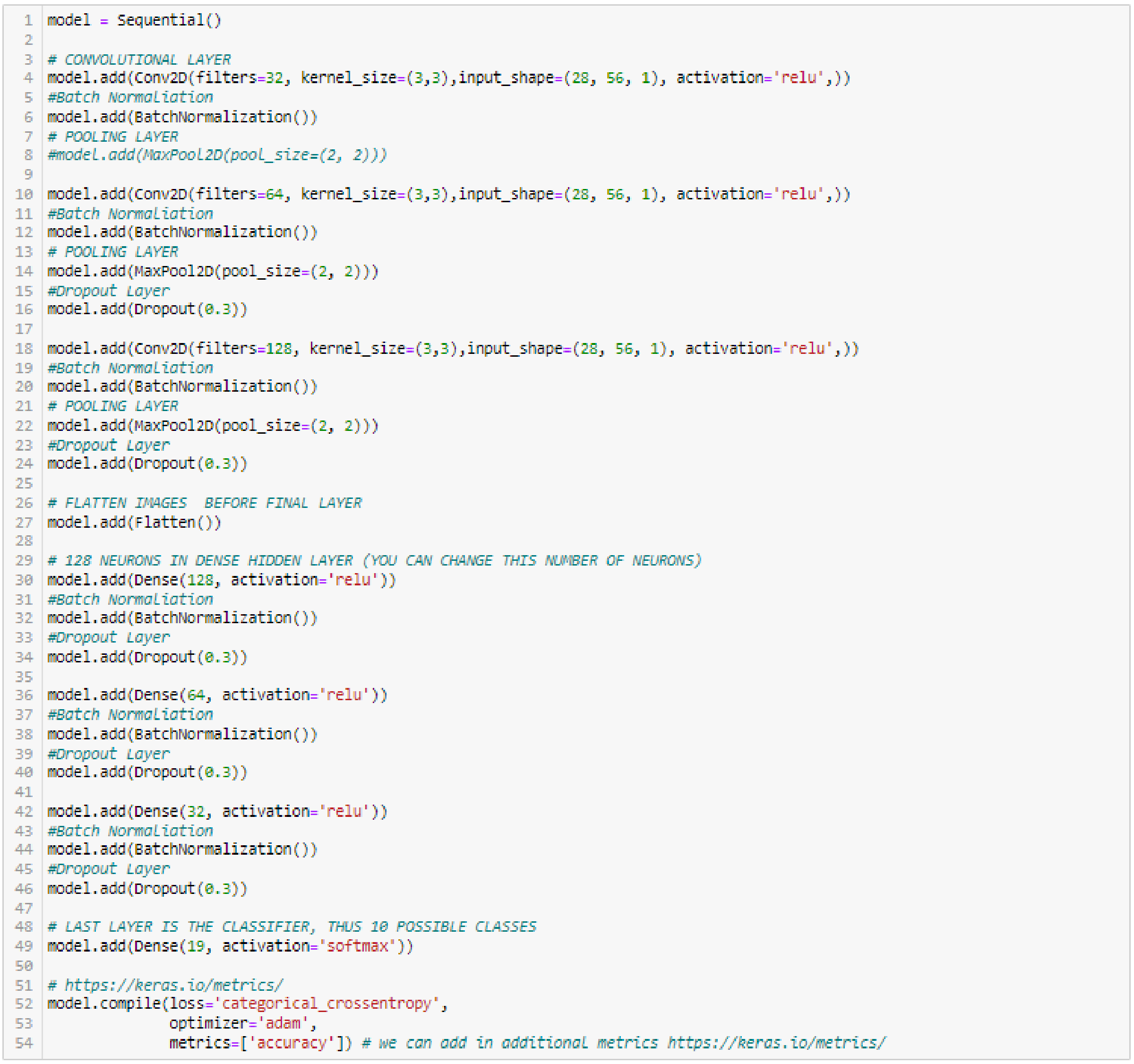
CNN Model-1



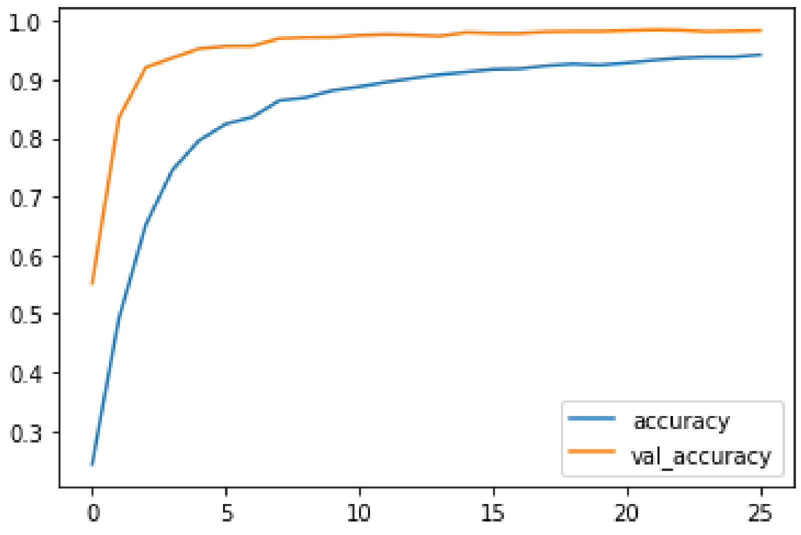
```
model = Sequential()

# CONVOLUTIONAL LAYER
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(28, 56, 1), activation='relu',))
# POOLING LAYER
model.add(MaxPool2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=(28, 56, 1), activation='relu',))
# POOLING LAYER
model.add(MaxPool2D(pool_size=(2, 2)))

model.add(Conv2D(filters=128, kernel_size=(3,3),input_shape=(28, 56, 1), activation='relu',))
# POOLING LAYER
model.add(MaxPool2D(pool_size=(2, 2)))


# FLATTEN IMAGES FROM 28 by 28 to 764 BEFORE FINAL LAYER
model.add(Flatten())

# 128 NEURONS IN DENSE HIDDEN LAYER (YOU CAN CHANGE THIS NUMBER OF NEURONS)
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))

# LAST LAYER IS THE CLASSIFIER, THUS 10 POSSIBLE CLASSES
model.add(Dense(19, activation='softmax'))

# https://keras.io/metrics/
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy']) # we can add in additional metrics https://keras.io/metrics/
```



CNN Model-2



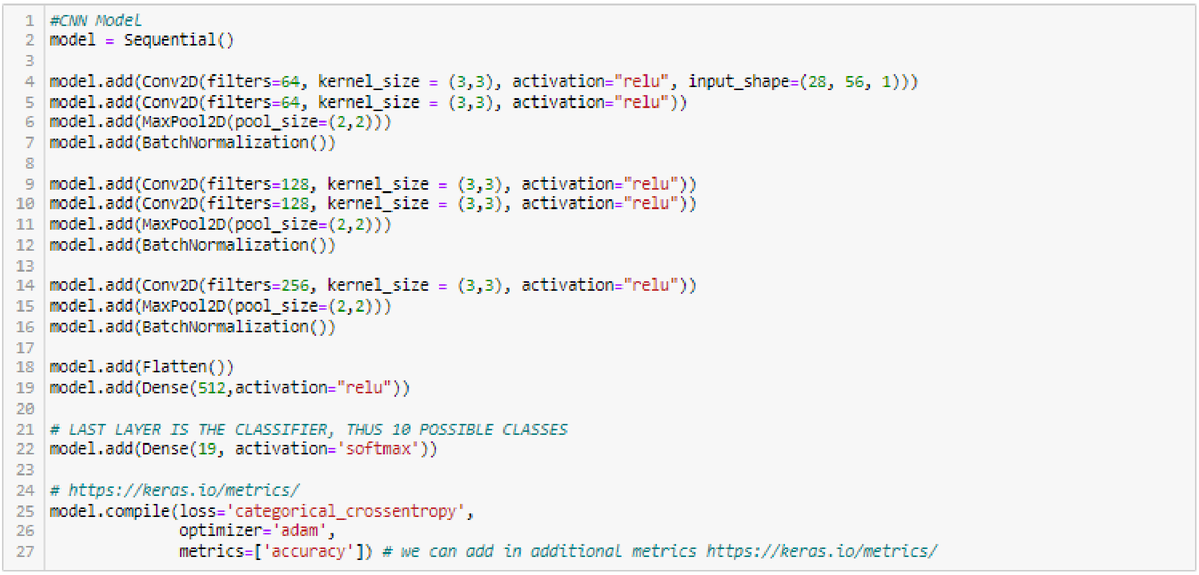
```python
model = Sequential()

# CONVOLUTIONAL LAYER
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(28, 56, 1), activation='relu',))
#Batch Normaliation
model.add(BatchNormalization())
# POOLING LAYER
#model.add(MaxPool2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=(28, 56, 1), activation='relu',))
#Batch Normaliation
model.add(BatchNormalization())
# POOLING LAYER
model.add(MaxPool2D(pool_size=(2, 2)))
#Dropout Layer
model.add(Dropout(0.3))

model.add(Conv2D(filters=128, kernel_size=(3,3),input_shape=(28, 56, 1), activation='relu',))
#Batch Normaliation
model.add(BatchNormalization())
# POOLING LAYER
model.add(MaxPool2D(pool_size=(2, 2)))
#Dropout Layer
model.add(Dropout(0.3))

# FLATTEN IMAGES  BEFORE FINAL LAYER
model.add(Flatten())

# 128 NEURONS IN DENSE HIDDEN LAYER (YOU CAN CHANGE THIS NUMBER OF NEURONS)
model.add(Dense(128, activation='relu'))
#Batch Normaliation
model.add(BatchNormalization())
#Dropout Layer
model.add(Dropout(0.3))

model.add(Dense(64, activation='relu'))
#Batch Normaliation
model.add(BatchNormalization())
#Dropout Layer
model.add(Dropout(0.3))

model.add(Dense(32, activation='relu'))
#Batch Normaliation
model.add(BatchNormalization())
#Dropout Layer
model.add(Dropout(0.3))

# LAST LAYER IS THE CLASSIFIER, THUS 10 POSSIBLE CLASSES
model.add(Dense(19, activation='softmax'))

# https://keras.io/metrics/
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy']) # we can add in additional metrics https://keras.io/metrics/
```



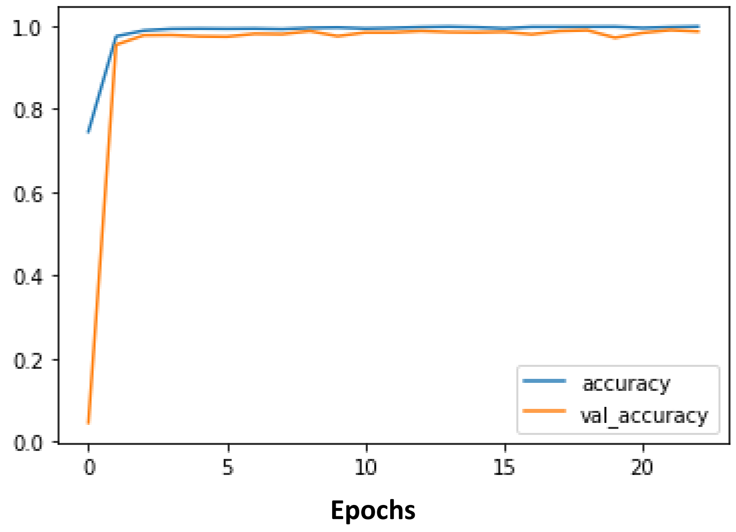
Epochs

Model-3 CNN architecture is the same as Model-2. Only difference, no validation dataset and training was performed on the entire training dataset. Also, training was performed for a total of 100 epochs without early-stopping callbacks.

CNN Model-4



```python
#CNN Model
model = Sequential()

model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=(28, 56, 1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(512,activation="relu"))

# LAST LAYER IS THE CLASSIFIER, THUS 10 POSSIBLE CLASSES
model.add(Dense(19, activation='softmax'))

# https://keras.io/metrics/
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy']) # we can add in additional metrics https://keras.io/metrics/
```


Epochs
accuracy
val_accuracy