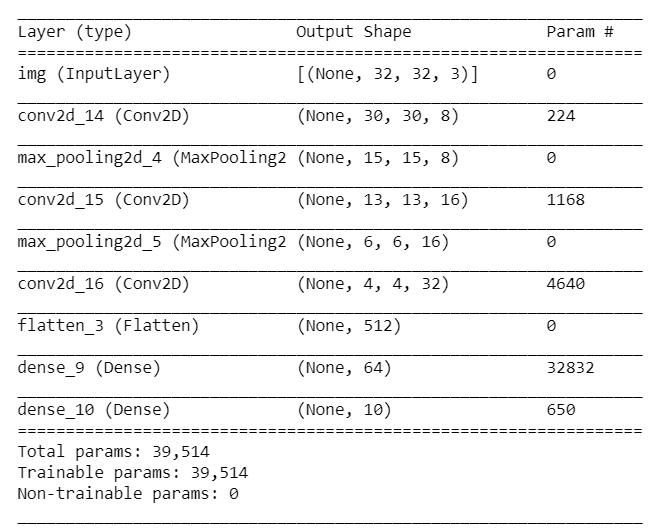
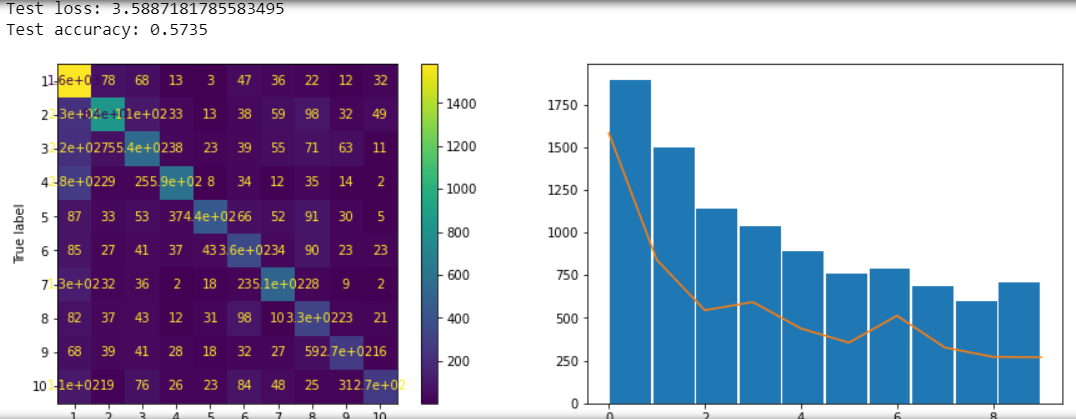
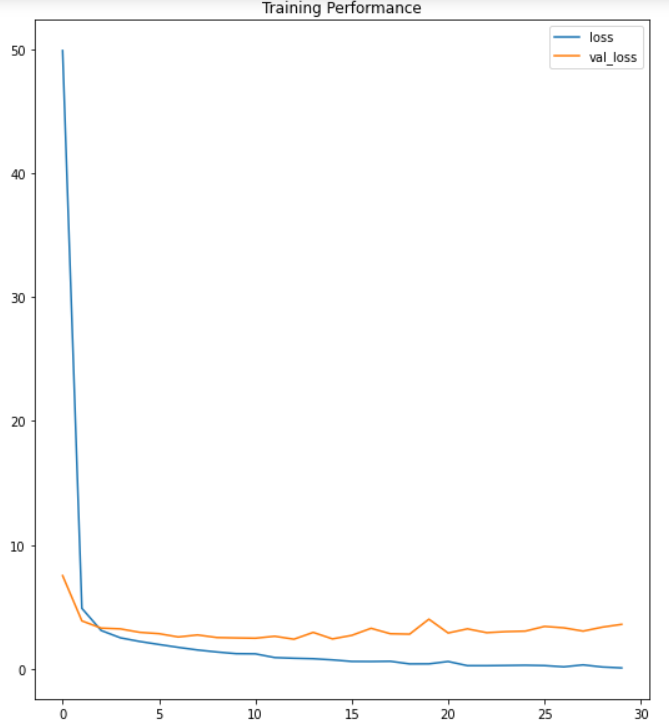
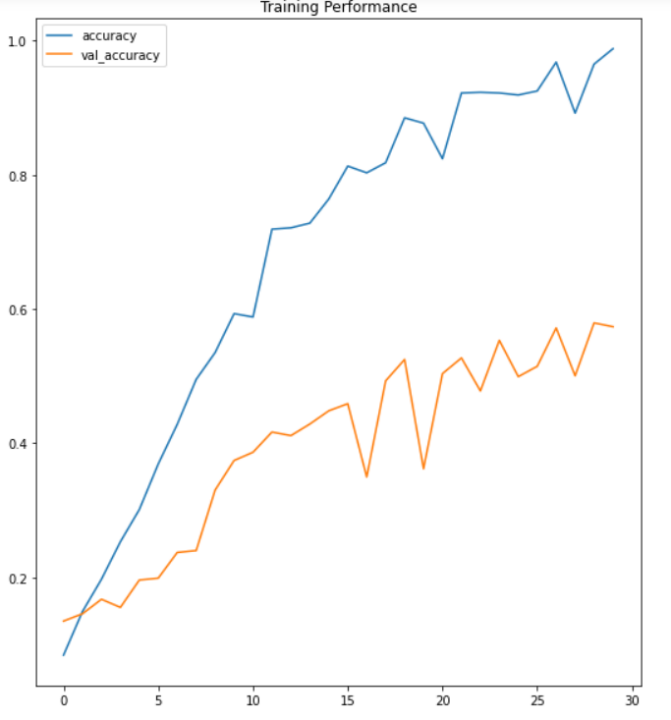
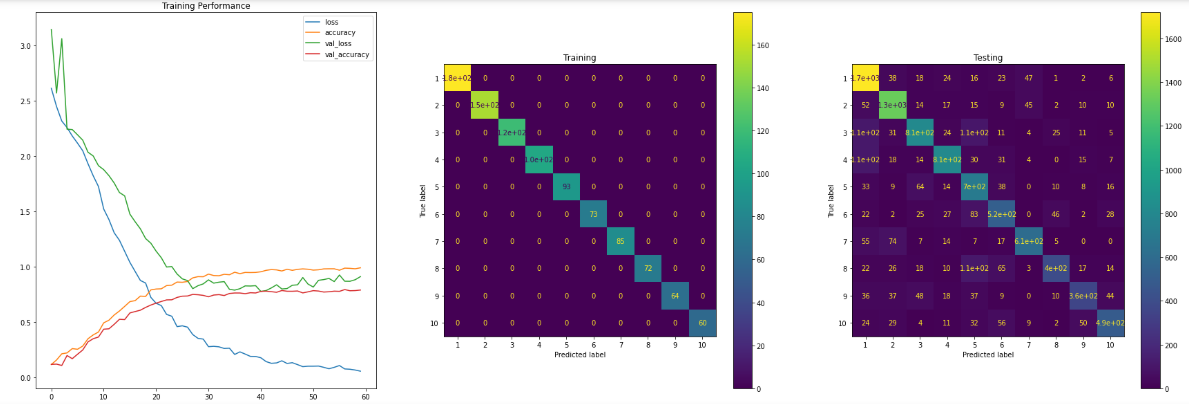
# Problem 3: Training and Adapting Deep Networks

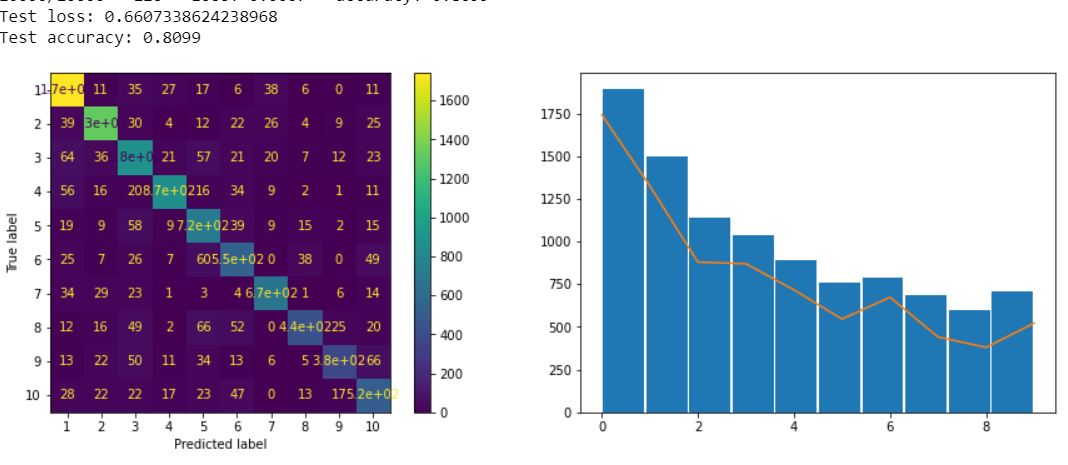
Train a model from scratch with no data augmentation

The data given is in 32 \* 32 \* 3 format with Y value range from 1 to 10. A few steps were taken to trim the data into an ideal format for Keras. After transposing and reshaping the arrays, the data was printed out to ensure the integrity. Once data Is ready, we start to design our model. To test our understanding of each layer and Keras functional APIs, the team created a simple network with 3 Conv2D (filter in range [8, 16, 32]) and 2 MaxPooling2D (pool\_size = 2). It then passes though a flatten layer and a core dense layer. Output layer is with 10-unit length since we have 10 classes.

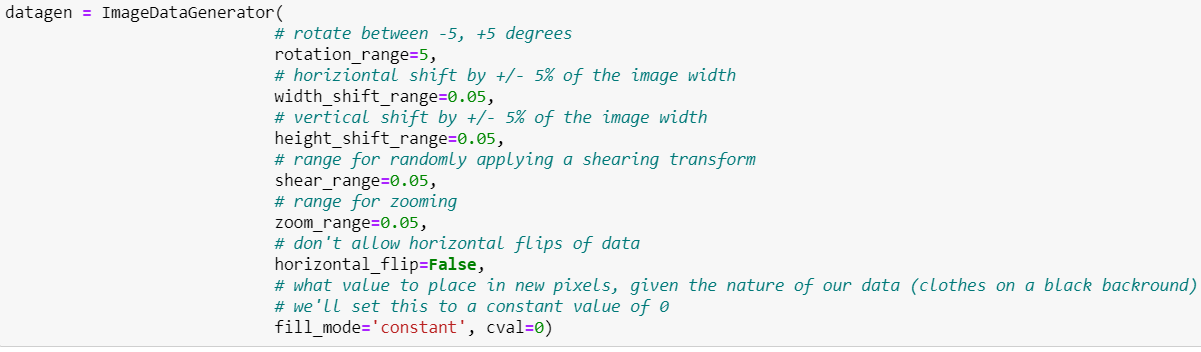
Layers in the beginning of a network learn fewer convolutional filters whereas they learn more as network grows deeper. Hence, the choice of 8, 16, 32 were made. Max pooling is a discretization process to down-sample input from above layers. It is used to reduce dimensionality and help with over-fitting as well as the reduction of cost to compute. Parameters above mean the previous dimension will be reduced by half. The Flatten layer flattens Matrix into a single array before parsing to Dense layer. Dense layer is a common densely connected NN layer. The last Dense layer is the output layer with 10-unit length meaning 10 classes. The result is of course not optimal since this is the first model run through. Optimizing the model requires trial and error, analyzing the result and seeking the best possible combination of parameters.

After playing around with several networks from tutorial examples, the best achieved result has a test accuracy of 57.35% and test loss of 3.589. The model is overfitting and the overall performance is not ideal considering the simplicity of the network.

Accepting defeat, we then borrowed one of the VGG networks from the examples with 578,674 total parameters. It includes layers as batch normalization, activation, spatial dropout and dropout. This has seen tremendous results in training the KMNIST and MNIST datasets. Batch normalization will standardize the input and enforce a mean of 0 and a standard deviation of 1. This results in layers learning on a balanced distribution of inputs and thus accelerate the training process. The performance will have limited improvement as well because of regularization effect. The choice of Activation is rather straight forward. Due to ReLu’s non-saturation of its gradient, it is deemed superior. The convergence of stochastic gradient descent occurs in timely manner because of this. Dropout is a technique to randomly ignore some neurons in training to prevent overfitting. They help with generalize the performance of the model. Batch size is the number of samples that will be passed through in one iteration. One epoch is a complete pass of all the training data. Smaller batch size indicates less memory usage but at the same time less accuracy in gradient estimates. Large epoch size might result in overfitting in the later stage of epochs. 

The VGG network (Batch size is at 128, Epoch is at 60) performs rather well with a training accuracy of 95.2% and testing accuracy of 77.47%. The validation loss was reduced to 0.7742 from the initial 3.14. Our testing data is vastly larger than the training data. This creates a hurdle for the model to gauge on unseen patterns. This model converged at 41st epoch. Models in the later epochs are clearly overfitting the training data. We tried to tweak the model by introducing VGG 3 stage network with the intent to achieve a better result. The model (Batch size is at 128, Epoch is at 35) improved slightly with a test accuracy of 80.99%. Weight balance function was introduced in all the models after carefully considering the issue of imbalance class weights in the training dataset. Other networks from ResNet were tested as well but due to subpar performance they will not be included in the discussion.

Train a model from scratch with data augmentation

We decide to use ImageDataGenerator for the data augmentation task. It comes with parameters which allow predefining the modification intended for existing dataset. Some of the common features are change of rotation range, width shift range, height shift range, shear range, zoom range, horizontal flip, and fill mode. The datagen.flow function will randomly generate data patches for fitting in the model based on the predefined parameters in ImageDataGenerator function. Most of the parameters are set to change in a small range because this is a natural way of data augmentation which benefits machine vision tasks. Rather than distorting the images, randomized small alteration will act as a creation of additional data for machine to learn the patterns reside in the new image dataset.

The next step is to train the data in a previously chosen network. We selected the first VGG network with the most parameters which is considerably worse than the second one. In this case, we can then compare the two by fitting in more data into the worse network to verify whether more data entries will yield a better result even if the network is less optimal.

Due to the computational constraint, the epoch is set at 100 for this task. Although the model hasn’t converged, it clearly has a good performance with no overfitting issue shown so far during the 100 epochs. It has achieved a better performance than the VGG 3 stage network from previous tests already. The test accuracy has increased to 82.45% (original accuracy is 77.47%) with a validation loss of 0.6026 (original loss from previous model is 0.7742).

