Question 2: Classification : Convolutional Neural Networks

For Question 2, we mainly tried four CNN models and two RestNet models separately. So, in this report, design, implementation and results of each model would be explained one by one with comparison of different models followed in between and in the end.

1. CNN-Model 1

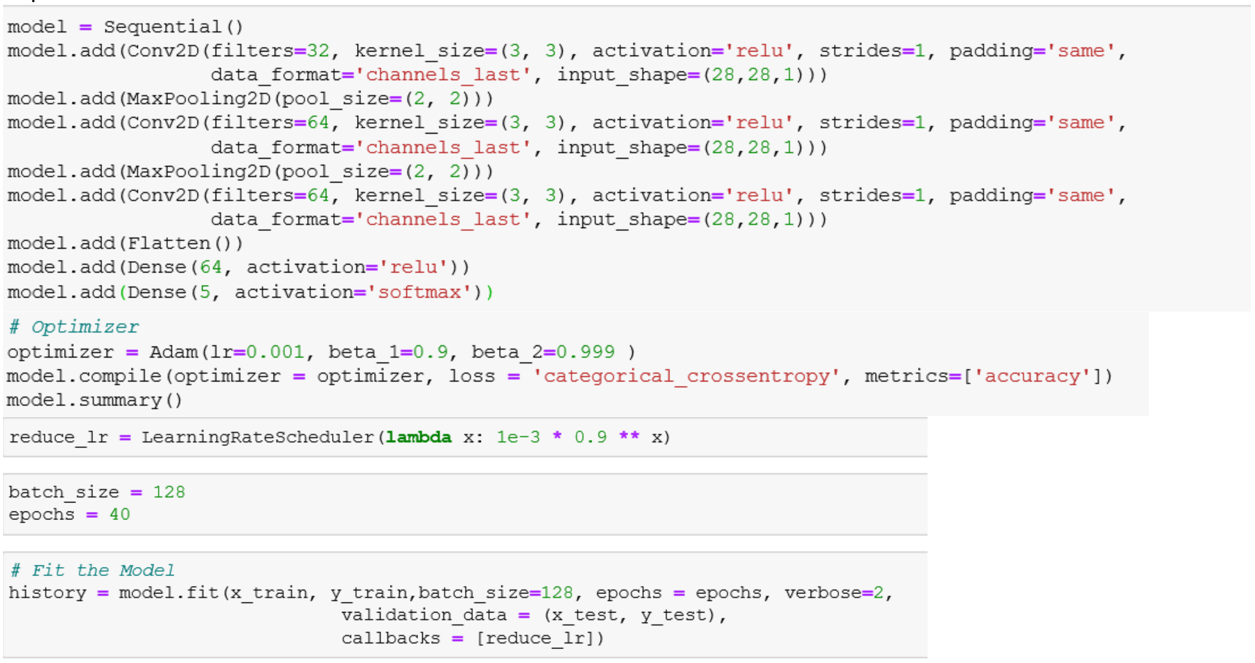
• Model design

As first convolutional neural network model, we decided to try a simple network architecture with 8 layers sequentially. CNN has three main types of layers to construct a model including Convolution layer (CONV), Pooling layer (POOL) and Fully connected layers (FC). Convolutional Layers are the Building blocks of ConvNets; Pooling Layers are used for Dimensionality Reduction or DownSampling the Input; and Fully connected layers in the end combines all the features of the Previous Layers. There are also another two necessary layers: Flatten layer to map the input to a 1D vector and Output Layer that has units equal to the number of classes to be identified.

In model 1, the basic structure is CONV layer-> POOL layer-> CONV layer-> POOL layer-> CONV layer -> Flatten layer-> FC layer-> OUTPUT layer. Since for this dataset, there are 5 classes in total for classification, three CONV layers would be enough to capture features to classify them all.

• Implementation

For CNN models, we used Sequential Keras API which is just a linear stack of layers. The main code blocks to implement model 1 are listed below:



```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                 data_format='channels_last', input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                 data_format='channels_last', input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                 data_format='channels_last', input_shape=(28,28,1)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(5, activation='softmax'))
```

```
# Optimizer
optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999 )
model.compile(optimizer = optimizer, loss = 'categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

```
reduce_lr = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```

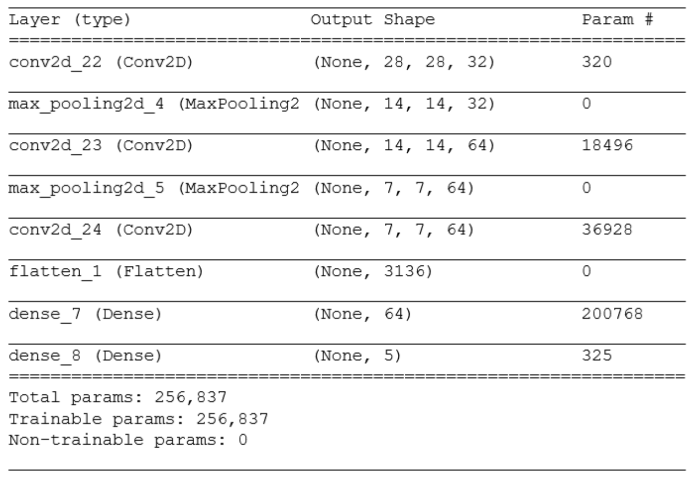
```
batch_size = 128
epochs = 40
```

```
# Fit the Model
history = model.fit(x_train, y_train,batch_size=128, epochs = epochs, verbose=2,
                              validation_data = (x_test, y_test),
                              callbacks = [reduce_lr])
```

• Explanation of model

In model 1, the basic structure is CONV layer-> POOL layer-> CONV layer-> POOL layer-> CONV layer -> Flatten layer-> FC layer-> OUTPUT layer. The first CONV layer has a set of 32 independent filters and the second and third CONV layers both have 64 filters. All three CONV layers chose the most common size of filter-3x3, RELU activation function, one step moving stride and same convolutions padding option. For pooling layers, 2x2 pool size are used for downsampling. After mapping the input to a 1D vector, one full connected layer is added to combine all features of the previous layers and connected to 64 outputs using RELU activation function. And the output layer used 'softmax' activation function in case of Multi-Class Classification to classify the outputs into 5 classes. The model summary is as following figure, total trainable parameters are 256,837 and output shape of every layer is listed.

For the optimizer used by the model, there are many choices like Adam, RMSprop etc. listed in Keras documents. Adam is the commonly used optimizer, so we chose to use it here. The loss function for the neural network which we want to minimize is specified as "categorical_crossentropy" for multi-class Classification. We chose accuracy as the metric to evaluate our models performance.



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_22 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 7, 7, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_7 (Dense)              (None, 64)                200768
_________________________________________________________________
dense_8 (Dense)              (None, 5)                 325
=================================================================
Total params: 256,837
Trainable params: 256,837
Non-trainable params: 0
_________________________________________________________________
```

For learning rate decay, the Learning rate should be properly tuned, such that it is not too high to take very large steps, neither it should be too small, which would not alter the Weights and Biases. We used LearningRateScheduler here, which takes the step decay function as argument and return the updated learning rates for use in optimizer at every epoch stage. Basically, it outputs a new learning rate at every epoch stage.

Before fitting the model, the batch size means the number of training examples in one forward/backward pass, here since we have a relatively big sample 128 was set as the batch size. Epoch means one forward pass of all the training examples, and here we chose 40 epochs to try first.

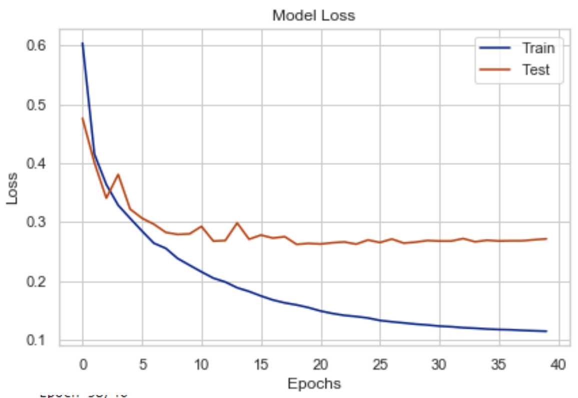
Since this is our first CNN model, most parameters are not familiar to us. Most parameters are chose according to the "Fashion-MNIST (CNN-Keras)◆[Accuracy-93%]" example from Kaggle (https://www.kaggle.com/fuzzywizard/fashion-mnist-cnn-keras-accuracy-93/#4)-Training-a-Convolutional-Neural-Network).

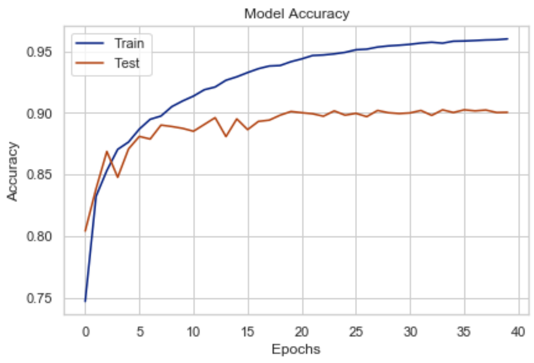
• **Runtime performance for training and testing.**
The runtime of model1 for training is approximately 30.5 second per epoch which is acceptable. And the runtime for separated test set is less than 1 second.

• **any plots to explain the performance of your approach**.
As it shows in the following two plots, the left plot is model loss vs. epochs including curves for train set and test set. The loss of train set decreases fast as the epochs increase from 0 to 10, and then starts to decrease slowly and gradually approaches to 0.1. The loss of validation set decreases in a similar way but with more fluctuations and finally approaches to 0.27. The exact loss of train set and validation set are shown in the figure below.

In the model accuracy plot, the accuracy of train set increases to 0.9097 quickly as the epochs increase from 0 to 10, and then gradually increases and approaches to 0.96 at the last epoch. The accuracy of validation set increases in a similar way but with more fluctuations and finally approaches to 0.90. After epoch 20, the accuracy of validation set is always fluctuating around 0.90 without significant increase.

Comparing the loss and accuracy curves between train set and test set, there is a significant difference between the loss that is more than 0.1 and between the accuracy that is mor than 0.05. So, the trained model is overfitting at a small extent.





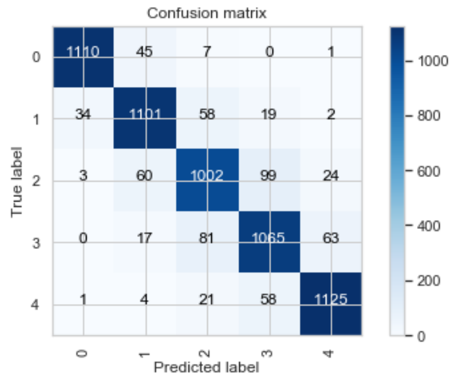
```
31s - loss: 0.1161 - acc: 0.9591 - val_loss: 0.2683 - val_acc: 0.9023
Epoch 39/40
31s - loss: 0.1153 - acc: 0.9593 - val_loss: 0.2700 - val_acc: 0.9003
Epoch 40/40
31s - loss: 0.1146 - acc: 0.9600 - val_loss: 0.2715 - val_acc: 0.9005
```

```
1]: score = model.evaluate(x_test, y_test)

    print('Loss: {:.4f}'.format(score[0]))
    print('Accuracy: {:.4f}'.format(score[1]))
```

```
5824/6000 [============================>.] - ETA: 0sLoss: 0.2715
Accuracy: 0.9005
```

• **Evaluate your code with other metrics on the training data and argue for the benefit of you approach**.
A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. Below is the performance of our classification model 1 on the validation data using Confusion Matrix. From the Confusion Matrix, a large number (99) of class2 are misclassified as class3 and 81 of class3 are misclassified as class2.

Also, from the classification report at right below, we can see a summary of classification result. In the report, the Precision is defined as the ratio of true positives to the sum of true and false positives; the Recall is defined as the ratio of true positives to the sum of true positives and false negatives; the F1 score is a weighted harmonic mean of precision and recall between 0 and 1.0 and Support is the number of actual occurrences of the class in the specified dataset. We can have similar results from f1-score that our model1 predicted 85% of class2 correctly and 86% of class3 correctly which are less than percentage of other three class that all have a rate higher or equal to 90%.



```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96      1163
           1       0.90      0.91      0.90      1214
           2       0.86      0.84      0.85      1188
           3       0.86      0.87      0.86      1226
           4       0.93      0.93      0.93      1209

    accuracy                           0.90      6000
   macro avg       0.90      0.90      0.90      6000
weighted avg       0.90      0.90      0.90      6000
```

The predict accuracy of model1 based on test.csv data on Kaggle is 0.90180. It is already a pretty high score, so with some improvement in parameters or architecture it might lead to score improvement.

2. CNN-Model 2

• Model design

Since model 1 already achieved a relatively high score in Kaggle, we think a little improvement in model 1 could lead to an improvement. At first, we tried to add an extra CONV layer that add more total parameters and more computation time, but the accuracy of validation set and prediction set decreased. Also, we tried different numbers of filters for each CONV layer, the results are all decreases. These trials are all failed, so we did not keep the results in code or prediction excel sheets, the submissions of prediction on Kaggle could have shown our process of trials.

From performance plots of model 1, we found that model1 is a little overfitting. The Dropout between layers could avoid overfitting based on the "Fashion-MNIST (CNN-Keras)◆[Accuracy-93%]" Kaggle example: "This randomly drops some percentage of neurons, and thus the weights gets Re-Aligned. The remaining Neurons learn more features and this reduces the dependency on any one Neuron." So after several trials, we added two 25% dropout after CONV layers and got an improvement in prediction.

This became our second model. In model 2, the basic structure is CONV layer-> POOL layer-> CONV layer-> POOL layer-> Dropout layer-> CONV layer -> Dropout layer-> Flatten layer-> FC layer-> OUTPUT layer.

• Implementation

The main code blocks to implement model 2 are listed below:

```
model2 = Sequential()
model2.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                  data_format='channels_last', input_shape=(28,28,1)))
model2.add(MaxPooling2D(pool_size=(2,2)))

model2.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                  data_format='channels_last'))
model2.add(MaxPooling2D(pool_size=(2,2)))
model2.add(Dropout(0.25))
model2.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                  data_format='channels_last'))

model2.add(Dropout(0.25))
model2.add(Flatten())
model2.add(Dense(64, activation='relu'))
model2.add(Dense(5, activation='softmax'))
```

```
# Optimizer
optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999 )
model2.compile(optimizer = optimizer, loss = 'categorical_crossentropy', metrics=['accuracy'])
model2.summary()
```

```
reduce_lr = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```

```
batch_size = 128
epochs = 40
```

```
# Fit the Model
history2 = model2.fit(x_train, y_train,batch_size=128, epochs = epochs, verbose=2,
                              validation_data = (x_test, y_test),
                              callbacks = [reduce_lr])
```

• Explanation of model

In model 2, the basic structure is CONV layer-> POOL layer-> CONV layer-> POOL layer-> Dropout layer-> CONV layer -> Dropout layer-> Flatten layer-> FC layer-> OUTPUT layer. Model 2 has exact same numbers of parameters, optimizers and same batch size and epochs as model1. The only difference is two extra Dropout layers in the model structure. Dropout is a regularization technique that penalizes the parameters. Here we set DropOutRate as 0.25.

The summary of model2 are listed as below.

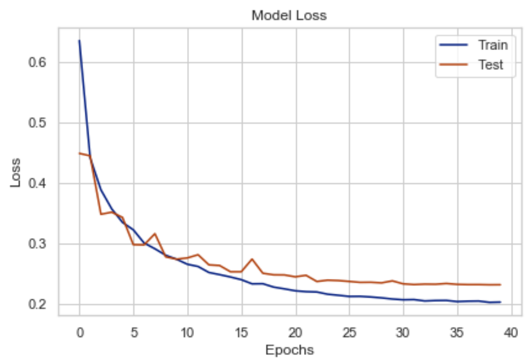
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_19 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
dropout_3 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 7, 7, 64)          36928
_________________________________________________________________
dropout_4 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_10 (Dense)             (None, 64)                200768
_________________________________________________________________
dense_11 (Dense)             (None, 5)                 325
=================================================================
Total params: 256,837
Trainable params: 256,837
Non-trainable params: 0
_________________________________________________________________
```

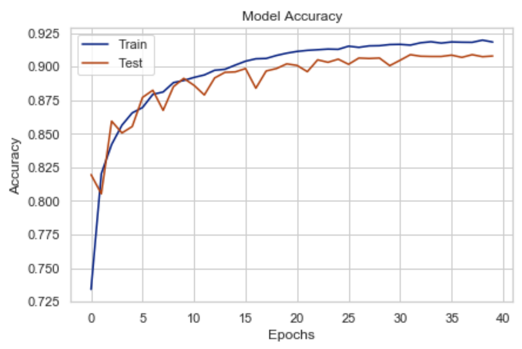
• **Runtime performance for training and testing.**
The runtime of model2 for training is approximately 33 second per epoch which is 1.5 second more than model1 average runtime. And the runtime for separated test set is 1 second. There is a little increase of runtime from model1 to model 2 , since the model2 structure has been added two more dropout layers.

• **plots to explain the performance of your approach**.
As it shows in the following two plots, the left plot is model loss vs. epochs including curves for train set and test set. The loss of train set decreased fast as the epochs increase from 0 to 10, and then starts to decrease slowly and gradually approaches to 0.2 other than 0.1 in model1. The loss of validation set decreases in a similar way but with more fluctuations and finally approaches to 0.23. The exact loss of train set and validation set are shown in the figure below.

In the model accuracy plot, the accuracy of train set increases to 0.9170 quickly as the epochs increase from 0 to 10, and then gradually increases and approaches to 0.937 at the last epoch. The accuracy of validation set increases in a similar way but with more fluctuations and finally approaches to 0.9088. After epoch 20, the accuracy of validation set is always fluctuating around 0.90 without significant increase.

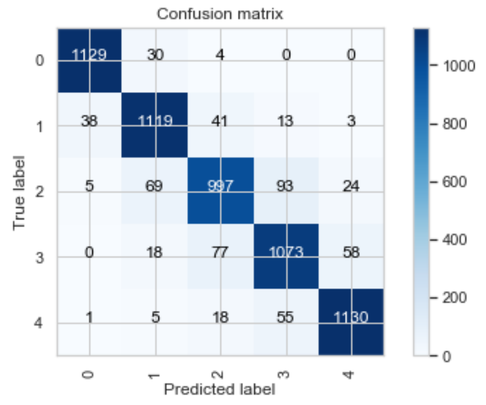
Comparing to plots of model1, the difference of loss between train set and validation set is only 0.08 and the accuracy difference is less than 0.03. The curve of Validation set are more close to curve of train set than plots of model 1. So the problem of overfit from model1 has been improved.





• **Evaluate your code with other metrics on the training data and argue for the benefit of you approach**.
Below is the performance of our classification model 2 on the validation data using Confusion Matrix. From the Confusion Matrix, still a large number (93) of class2 are misclassified as class3 and 77 of class3 are misclassified as class2. Compared to Confusion Matrix of model1, the two numbers are decreased slightly. And the model2 correctly predicted more of class 0,1,3,4 except class2 than model1.

Also, from the classification report at right below, we can see a summary of classification result. We can have similar results from f1-score that our model2 predicted 86% of class2 correctly and 87% of class3 correctly which are less than percentage of other three class that all have a rate higher or equal to 90%. At least, the f1-score of all class have increased slightly by 1 or 2 percent compared to model 1.



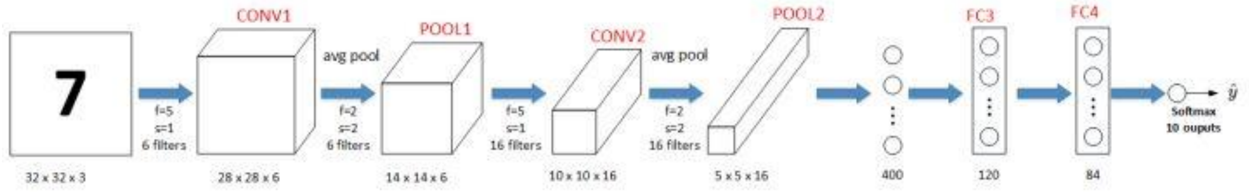
```
              precision    recall  f1-score   support

           0       0.96      0.97      0.97      1163
           1       0.90      0.92      0.91      1214
           2       0.88      0.84      0.86      1188
           3       0.87      0.88      0.87      1226
           4       0.93      0.93      0.93      1209

    accuracy                           0.91      6000
   macro avg       0.91      0.91      0.91      6000
weighted avg       0.91      0.91      0.91      6000
```

The predict accuracy of model2 based on test.csv data on Kaggle is 0.90500. The accuracy has increased 0.0032 which is only a slight change, but before the increase we have tried many methods and rerun many times. Model 2 has the ability to correctly predict 90.5% of classes and it only takes 1-2 second for prediction.

3. CNN-Model 3

• **Model design**

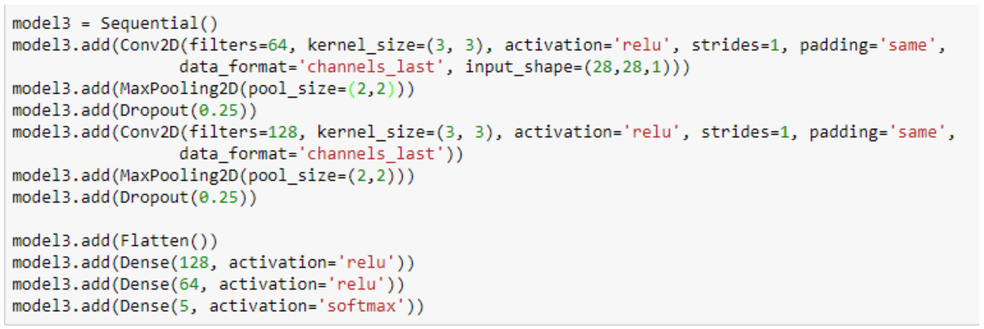
Since adding more CONV layers is not working to improve accuracy, we decided to try delete CONV layers to see the difference. And then we noticed that there is a classic CNN model called LeNet-5. The basic sequential structure of LeNet-5 is CONV layer->POOL layer->CONV layer->POOL layer->FC layer->FC layer->OUTPUT layer. As it showsin the image, the LetNet-5 structure example has a different input size, so we did not copy exactly.



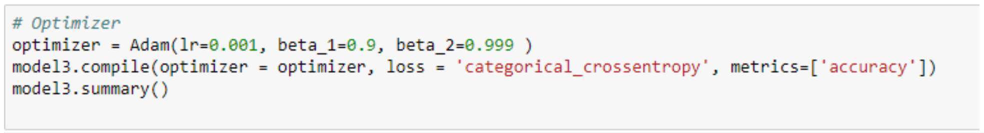
Based on our previous experience of overfitting problem, two dropout layers are added after POOL layers. So the model 3 is constructed based on LetNet-5 and previous models , the basic structure is CONV layer-> POOL layer-> Dropout layer-> CONV layer-> POOL layer-> Dropout layer-> Flatten layer-> FC layer-> FC layer -> OUTPUT layer.

• **Implementation**

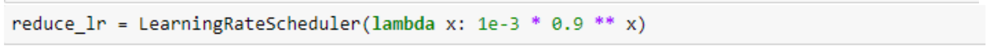
The main code blocks to implement model 2 are listed below:



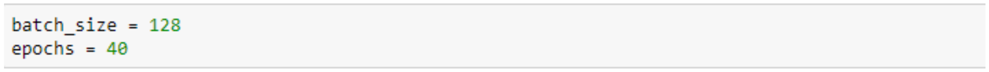
```
model3 = Sequential()
model3.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                 data_format='channels_last', input_shape=(28,28,1)))
model3.add(MaxPooling2D(pool_size=(2,2)))
model3.add(Dropout(0.25))
model3.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                 data_format='channels_last'))
model3.add(MaxPooling2D(pool_size=(2,2)))
model3.add(Dropout(0.25))

model3.add(Flatten())
model3.add(Dense(128, activation='relu'))
model3.add(Dense(64, activation='relu'))
model3.add(Dense(5, activation='softmax'))
```



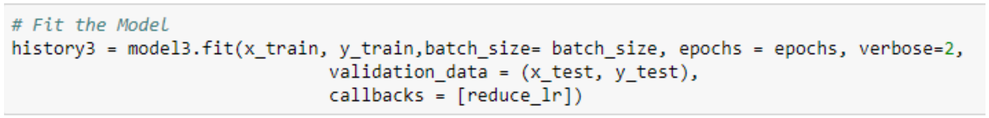
```
# Optimizer
optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999 )
model3.compile(optimizer = optimizer, loss = 'categorical_crossentropy', metrics=['accuracy'])
model3.summary()
```



```
reduce_lr = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```



```
batch_size = 128
epochs = 40
```

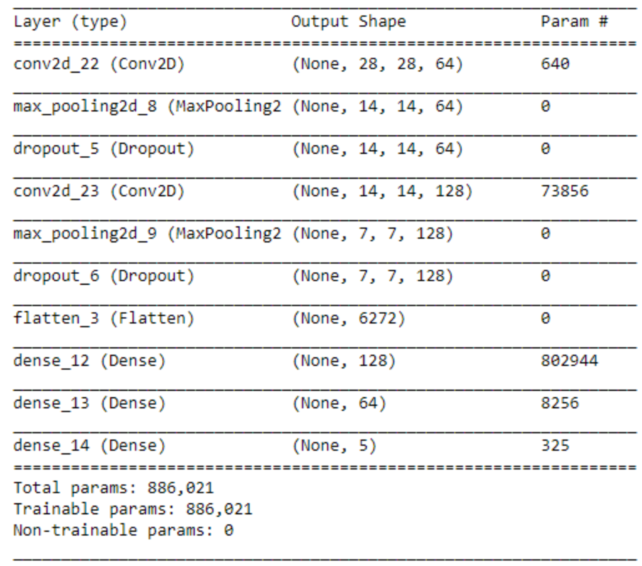


```
# Fit the Model
history3 = model3.fit(x_train, y_train,batch_size= batch_size, epochs = epochs, verbose=2,
                                validation_data = (x_test, y_test),
                                callbacks = [reduce_lr])
```

• **Explanation of model**

In model 3, the basic structure is CONV layer-> POOL layer-> Dropout layer-> CONV layer-> POOL layer-> Dropout layer-> Flatten layer-> FC layer-> FC layer -> OUTPUT layer. The optimizer, learning rate function and batch size and epochs are all same as model 1 and 2. At first, when we are trying different combinations of numbers of filters in CONV layers, we first used 32 and 64 of filters for CONV layers learnt from model 1 and 2, but it turns out they are not enough to capture enough features for classification so the accuracy deceased. After trying to increase the numbers of filters, the accuracy finally increased by almost 0.01. The numbers of filters of two CONV layers finally have changed to 64 and 128, and for two FC layers inputs are first transformed to 128 units and then to 64 units of output.

From the model summary below, we can find that the total parameters has been nearly four times more than model1 and 2. The total parameters are 886,021. More parameters need more computation time but the accuracy would be higher.



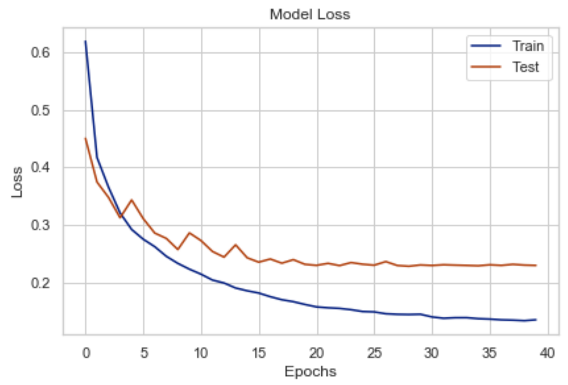
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_22 (Conv2D)           (None, 28, 28, 64)        640       
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 14, 14, 64)        0         
_________________________________________________________________
dropout_5 (Dropout)          (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 14, 14, 128)       73856     
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 7, 7, 128)         0         
_________________________________________________________________
dropout_6 (Dropout)          (None, 7, 7, 128)         0         
_________________________________________________________________
flatten_3 (Flatten)          (None, 6272)              0         
_________________________________________________________________
dense_12 (Dense)             (None, 128)               802944    
_________________________________________________________________
dense_13 (Dense)             (None, 64)                8256      
_________________________________________________________________
dense_14 (Dense)             (None, 5)                 325       
=================================================================
Total params: 886,021
Trainable params: 886,021
Non-trainable params: 0
_________________________________________________________________
```

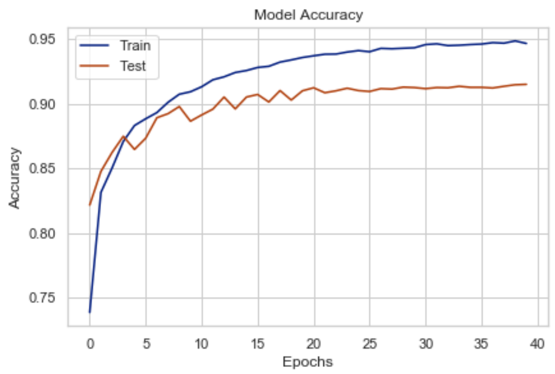
• **Runtime performance for training and testing.**
The runtime of model3 for training is approximately 70 second per epoch which is more than double of model1 and 2 average runtimes. And the runtime for separated test set is 2 second. Since the quantity of total parameters has increased by more than three times, the computation time are doubled than previous models

• **plots to explain the performance of your approach**.
As it shows in the following two plots, the left plot is model loss vs. epochs including curves for train set and test set. The loss of train set decreased fast as the epochs increase from 0 to 10, and then starts to decrease slowly and gradually approaches to 0.13. The loss of validation set decreases in a similar way but with more fluctuations and finally approaches to 0.23. The exact loss of train set and validation set are shown in the figure below.

In the model accuracy plot, the accuracy of train set increases to 0.9093 quickly as the epochs increase from 0 to 10, and then gradually increases and approaches to 0.946 at the last epoch. The accuracy of validation set increases in a similar way but with more fluctuations and finally approaches to 0.9150. After epoch 20, the accuracy of validation set is always fluctuating around 0.91 without significant increase.

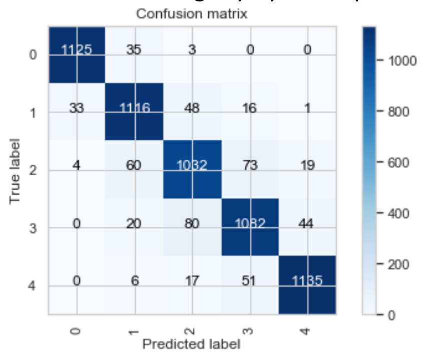
Comparing to plots of model2, the difference of loss and accuracy between train set and validation set are both slightly larger. The distance between curves of validation and train set are becoming wider again than plots of model 2. So the model3 might have problem of overfitting.





• **Evaluate your code with other metrics on the training data and argue for the benefit of you approach**.
Below is the performance of our classification model 3 on the validation data using Confusion Matrix. From the Confusion Matrix, a number (73) of class2 are misclassified as class3 and 80 of class3 are misclassified as class2. Compared to Confusion Matrix of model2, the first numbers are decreased while the second number increased. And the model3 did not predict more of different classes than model2.

Also, from the classification report at right below, we can see a summary of classification result. We can have similar results from f1-score that our model3 predicted 87% of class2 correctly and 88% of class3 correctly which are less than percentage of other three class that all have a rate higher or equal to 90%. At least, the f1-score of class 2,3,4 have increased slightly by 1 or 2 percent compared to model 2.



```
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      1163
           1       0.90      0.92      0.91      1214
           2       0.87      0.87      0.87      1188
           3       0.89      0.88      0.88      1226
           4       0.95      0.94      0.94      1209

    accuracy                           0.92      6000
   macro avg       0.92      0.92      0.92      6000
weighted avg       0.92      0.92      0.91      6000
```

The predict accuracy of model3 based on test.csv data on Kaggle is 0.91460. The accuracy has increased almost 0.01 from model2 so it could be an obvious improvement. But from the confusion matrix and classification report, we could not see a great improvement. And for the accuracy increase, we had a tradeoff of double runtime.

4. CNN-Model 4

• Model design

Since model 3 has much more parameters than model 2, we could add layers to drop some parameters. So we add Batch Normalization to achieve Zero mean and Variance one. It scales down outliers and forces the network to learn features in a distributed way, not relying too much on a Particular Weight and makes the model better Generalize the Images.

Based on our previous experience of overfitting problem, two dropout layers are added after POOL layers. So for the model 4, the basic structure is CONV layer-> POOL layer-> Dropout layer-> CONV layer->Batch Normalization layer-> POOL layer-> Dropout layer-> Flatten layer-> FC layer-> FC layer -> OUTPUT layer.

• Implementation

The main code blocks to implement model 2 are listed below:

```
model4 = Sequential()
model4.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                 data_format='channels_last', input_shape=(28,28,1)))
model4.add(MaxPooling2D(pool_size=(2,2)))
model4.add(Dropout(0.25))
model4.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', strides=1, padding='same',
                 data_format='channels_last'))
model4.add(BatchNormalization())
model4.add(MaxPooling2D(pool_size=(2,2)))
model4.add(Dropout(0.25))

model4.add(Flatten())
model4.add(Dense(128, activation='relu'))
model4.add(Dense(64, activation='relu'))
model4.add(Dense(5, activation='softmax'))
```

```
# Optimizer
optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999 )
model4.compile(optimizer = optimizer, loss = 'categorical_crossentropy', metrics=['accuracy'])
model4.summary()
```

```
reduce_lr = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```

```
batch_size = 128
epochs = 40
```

```
# Fit the Model
history4 = model4.fit(x_train, y_train,batch_size=128, epochs = epochs, verbose=2,
                              validation_data = (x_test, y_test),
                              callbacks = [reduce_lr])
```

• Explanation of model

In model 4 , the basic structure is CONV layer-> POOL layer-> Dropout layer-> CONV layer-> Batch Normalization layer-> POOL layer-> Dropout layer-> Flatten layer-> FC layer-> FC layer -> OUTPUT layer. The optimizer, learning rate function and batch size and epochs are all same as model 3.

From the model summary below, we can find that the total trainable parameters was little more than model 3. The total parameters are 886,533 with 256 non- trainable parameters.

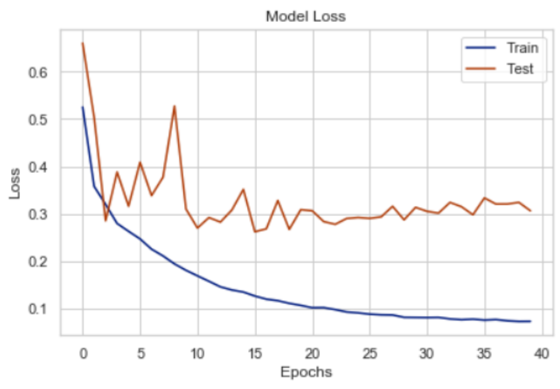
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_28 (Conv2D)           (None, 28, 28, 64)        640       
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 14, 14, 64)        0         
_________________________________________________________________
dropout_11 (Dropout)         (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 14, 14, 128)       73856     
_________________________________________________________________
batch_normalization_24 (Batc (None, 14, 14, 128)       512       
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 7, 7, 128)         0         
_________________________________________________________________
dropout_12 (Dropout)         (None, 7, 7, 128)         0         
_________________________________________________________________
flatten_6 (Flatten)          (None, 6272)              0         
_________________________________________________________________
dense_21 (Dense)             (None, 128)               802944    
_________________________________________________________________
dense_22 (Dense)             (None, 64)                8256      
_________________________________________________________________
dense_23 (Dense)             (None, 5)                 325       
=================================================================
Total params: 886,533
Trainable params: 886,277
Non-trainable params: 256
_________________________________________________________________
```

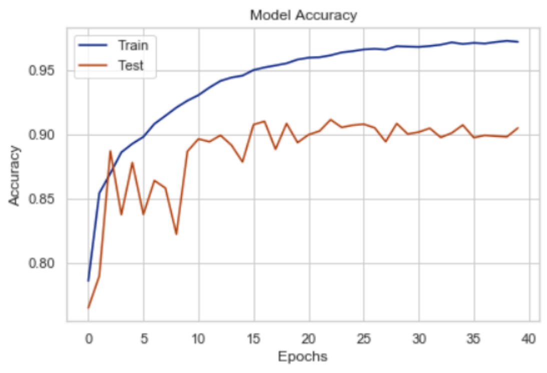
• **Runtime performance for training and testing.**
The runtime of model4 for training is approximately 120 second per epoch which is nearly double of model4 average runtimes. And the runtime for separated test set is 4 second. The total parameters only increased slightly, but one more layer was added in the model that need more computation time.

• **plots to explain the performance of your approach**.
As it shows in the following two plots, the left plot is model loss vs. epochs including curves for train set and test set. The loss of train set decreased fast as the epochs increase from 0 to 10, and then starts to decrease slowly and gradually approaches to 0.073. The loss of validation set also decreases but with more significant fluctuations and finally approaches to 0.3065.

In the model accuracy plot, the accuracy of train set increases to 0.9263 quickly as the epochs increase from 0 to 10, and then gradually increases and approaches to 0.9720 at the last epoch. The accuracy of validation set increases in a similar way but with more dramatic fluctuations and finally approaches to 0.9050. After epoch 20, the accuracy of validation set is always fluctuating around 0.90 without significant increase.

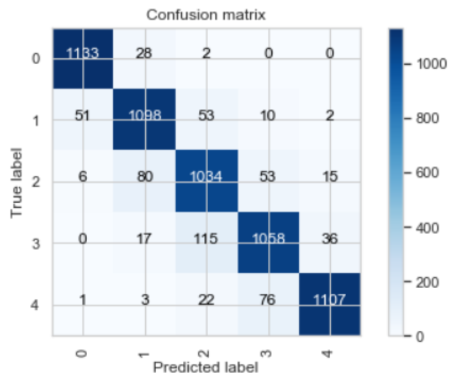
Model 4 might have a more serious overfitting problem than model3.

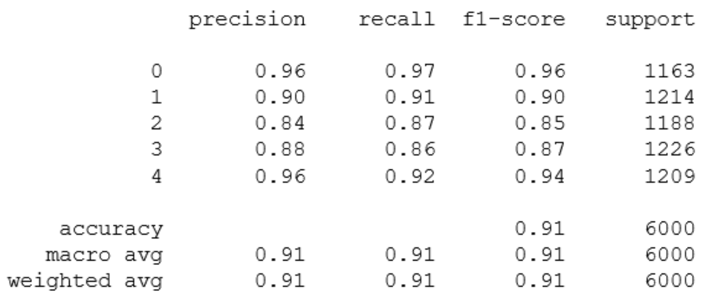




• Evaluate your code with other metrics on the training data and argue for the benefit of you approach.
Below is the performance of our classification model 4 on the validation data using Confusion Matrix. From the Confusion Matrix, a number (80) of class2 are misclassified as class1 and 115 of class3 are misclassified as class2. Compared to Confusion Matrix of model3, the prediction performance is not better.

Also, from the classification report at right below, we can see a summary of classification result. We can have similar results from f1-score that our model3 predicted 85% of class2 correctly and 87% of class3 correctly which are less than percentage of other three class that all have a rate higher or equal to 90%. And the f1-score for class0 and 1 have both decreased one percent.

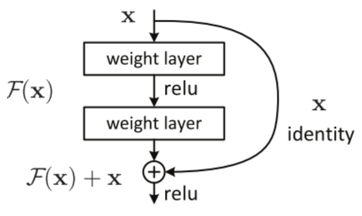



```
              precision    recall  f1-score   support

           0       0.96      0.97      0.96      1163
           1       0.90      0.91      0.90      1214
           2       0.84      0.87      0.85      1188
           3       0.88      0.86      0.87      1226
           4       0.96      0.92      0.94      1209

    accuracy                           0.91      6000
   macro avg       0.91      0.91      0.91      6000
weighted avg       0.91      0.91      0.91      6000
```

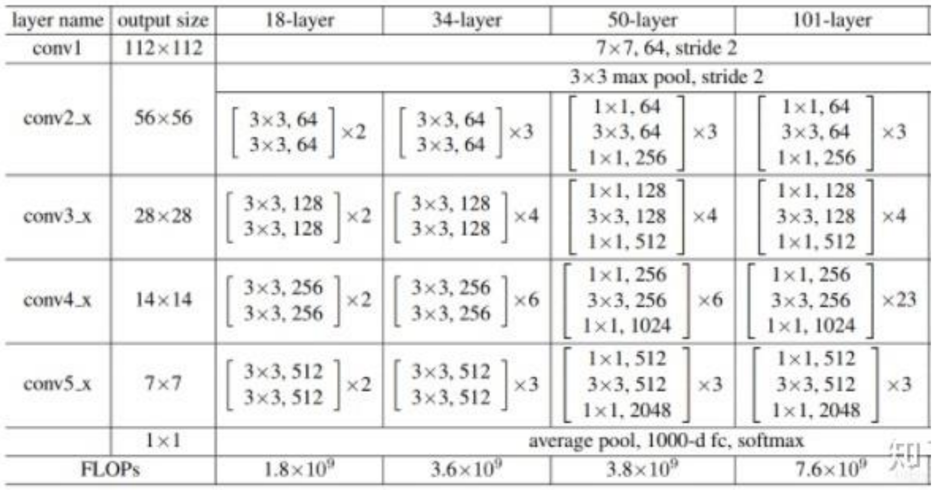
According to performance results of model6, the Batch Normalization layer might not be suitable for this data set that would lead to overfitting problem. Among four CNN models, Model2 and Model 3 have the best performance. Model 3 has the best accuracy performance with a longer runtime and Model 2 has a slightly lower accuracy with less than ½ runtime of Model 3. Both Model 2 and Model 3 are reasonable models that could be used in the future.

5. ResNets-Model 5

• Model design
Using Resnet model, we want to see if the use of Resnet learn a better classifier than a simple CNN. The core idea of ResNet is introducing a so-called "identity shortcut connection" that skips one or more layers. Residual Networks consists of certain numbers of residual blocks, shown in the figure.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
x
identity
a residual block

Here. For model 5 we tried to use pre-trained Resnet model in Keras to simply apply on the train data set. The pre-trained Resnet model available in Keras is ResNet50 that has 50 layers in total. Here in the follow we can find how the 50-layer Resnet model is constructed.



layer name	output size	18-layer	34-layer	50-layer	101-layer
conv1	112×112	7×7, 64, stride 2			
		3×3 max pool, stride 2			
conv2_x	56×56	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax			
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9

- **Implementation**

The main code blocks to implement model 5 are listed below:

```
def prepare_data_for_resnet50(data_to_transform):
    data = data_to_transform.copy().values
    data = data.reshape(-1, 28, 28) / 255
    data = X_rgb = np.stack([data, data, data], axis=-1)
    return data
```

```
x_train2 = prepare_data_for_resnet50(x_train2)
x_test2 = prepare_data_for_resnet50(x_test2)
r_test= prepare_data_for_resnet50(r_test)
```

```
model = Sequential()
model.add(ResNet50(include_top=False, pooling='avg', weights='imagenet'))
model.add(Dense(5, activation='softmax'))
```

```
from keras.optimizers import  Adam
adam = Adam(lr=0.0001)
model.compile(optimizer= adam, loss='categorical_crossentropy', metrics=['accuracy'])
```

- **Explanation of model**

To use the ResNet50 properly, training data is multiplied to three dimensions so ResNet50 could work with it. For ResNet50, we set include_top=False to not include the final pooling and fully connected layer in the original model and added a dense output layer to get outputs classifier in 5 classes. Because the ResNet50 would have a 1000 classes output in the end. Other optimizers are still same as previous models.

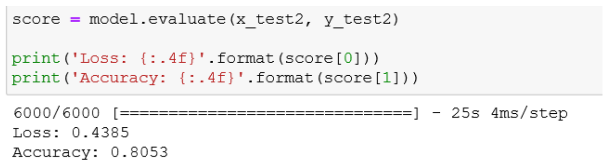
From the model summary below, we can see that there are 23,597,957 parameter in total and non-trainable parameters 53,120. This requires much more computation time. Unfortunately, one of our team members who works on ResNets model were sick and having fever for six days. So we only have one day left to try on this model. So we chose to run a 10 epochs model to see the result.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
resnet50 (Model)             (None, 2048)              23587712  
_________________________________________________________________
dense_8 (Dense)              (None, 5)                 10245     
=================================================================
Total params: 23,597,957
Trainable params: 23,544,837
Non-trainable params: 53,120
_________________________________________________________________
```

- **Runtime performance for training and testing.**

The runtime of model5 for training is approximately 33 minutes per epoch which is way much more than previous model average runtimes. And the runtime for separated test set is 25 second. Since the quantity of total parameters has increased dramatically, the computation time are much more.

• **plots to explain the performance of your approach**.

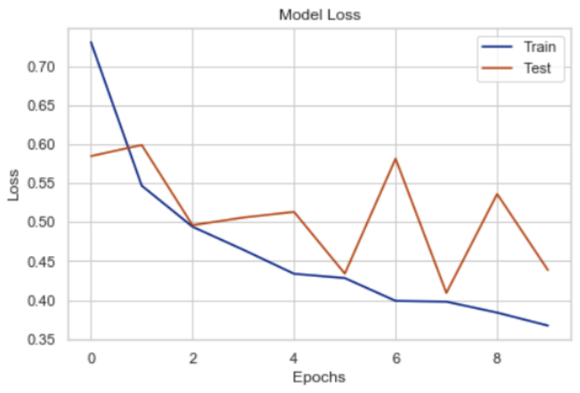


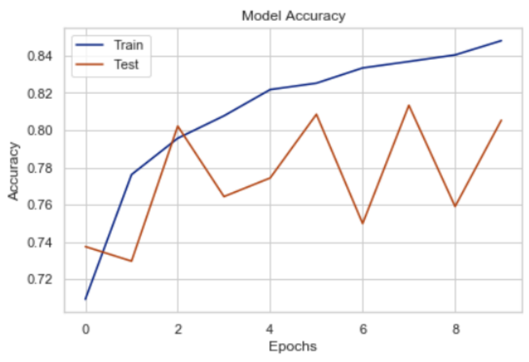
```
score = model.evaluate(x_test2, y_test2)

print('Loss: {:.4f}'.format(score[0]))
print('Accuracy: {:.4f}'.format(score[1]))
```

```
6000/6000 [==============================] - 25s 4ms/step
Loss: 0.4385
Accuracy: 0.8053
```

From 10 epochs, the result accuracy is quite low that is only 0.8053 on validation set. As it shows in the following two plots, the left plot is model loss vs. epochs including curves for train set and test set. The loss of train set decreased from 0.7 to o.36 as the epochs increase from 0 to 10. The loss of validation set also decreases but with more significant fluctuations and finally approaches to 0.43.

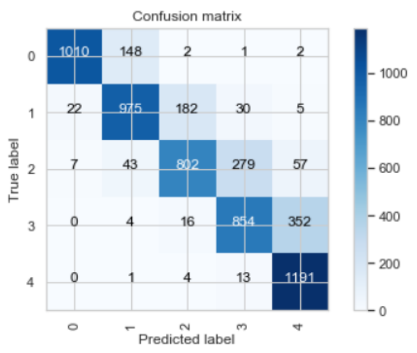
In the model accuracy plot, the accuracy of train set increases to 0.8480. The accuracy of validation set is not gradually increasing but have obvious fluctuations and finally approaches to 0.8053. The performance from the plot is really bad even though we only did 10 epochs compared to CNN models.

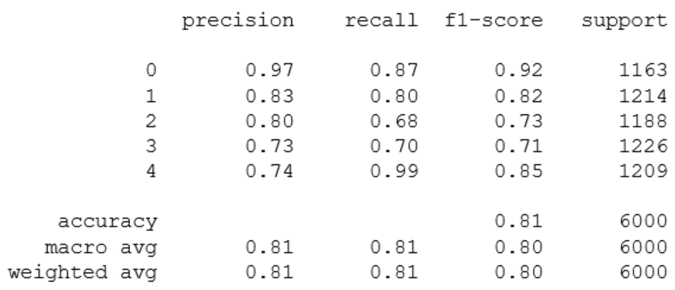




• **Evaluate your code with other metrics on the training data and argue for the benefit of you approach**.
Below is the performance of our classification model 5 on the validation data using Confusion Matrix. From the Confusion Matrix, a number (148) of class0 are misclassified as class1, 182 of class1 are misclassified as class2, 279 of class12 are misclassified as class3 and 352 of class3 are misclassified as class4. The predicting ability for class 2and 3 is still much weaker than other three classes.

From the classification report at right below, we can see a summary of classification result. We can have similar results from f1-score that our model5 predicted only 73% of class2 correctly and 71% of class3 correctly which are less than percentage of other three class that all have a rate higher than 80%.





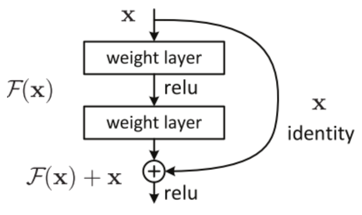
```
              precision    recall  f1-score   support

           0       0.97      0.87      0.92      1163
           1       0.83      0.80      0.82      1214
           2       0.80      0.68      0.73      1188
           3       0.73      0.70      0.71      1226
           4       0.74      0.99      0.85      1209

    accuracy                           0.81      6000
   macro avg       0.81      0.81      0.80      6000
weighted avg       0.81      0.81      0.80      6000
```

This model5 has a bad performance from all kinds of result aspects. Although trying more epochs and adding layers might have a better performance. But the ResNet50 model is still not suitable for this dataset since the runtime is too long. So this model is not recommended for this dataset classification. This model5 is constructed based on Kaggle example" Using ResNet for mnist"(https://www.kaggle.com/donatastamosauskas/using-resnet-for-mnist).

6. ResNets-Model 6

• Model design

Since using pretrained model ResNet 50 is not suitable for this data set. We could try another way of using ResNet. The core idea of ResNet is introducing a so-called "identity shortcut connection" that skips one or more layers. Residual Networks consists of certain numbers of residual blocks, shown in the figure.

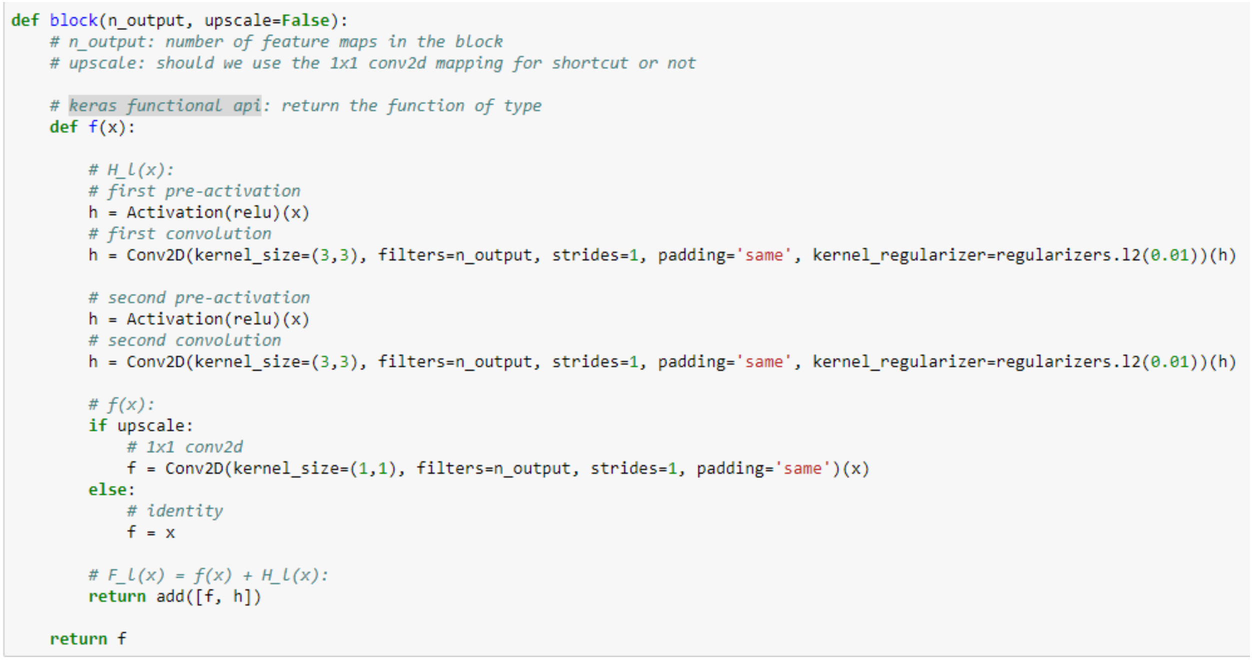


a residual block

So for model 6 we tried to build one residual block using keras functional API to apply on the train data set. So the model 6 structure would be like the residual block in the above figure.

• Implementation

The main code blocks to implement model 5 are listed below:



```
def block(n_output, upscale=False):
    # n_output: number of feature maps in the block
    # upscale: should we use the 1x1 conv2d mapping for shortcut or not

    # keras functional api: return the function of type
    def f(x):

        # H_l(x):
        # first pre-activation
        h = Activation(relu)(x)
        # first convolution
        h = Conv2D(kernel_size=(3,3), filters=n_output, strides=1, padding='same', kernel_regularizer=regularizers.l2(0.01))(h)

        # second pre-activation
        h = Activation(relu)(x)
        # second convolution
        h = Conv2D(kernel_size=(3,3), filters=n_output, strides=1, padding='same', kernel_regularizer=regularizers.l2(0.01))(h)

        # f(x):
        if upscale:
            # 1x1 conv2d
            f = Conv2D(kernel_size=(1,1), filters=n_output, strides=1, padding='same')(x)
        else:
            # identity
            f = x

        # F_l(x) = f(x) + H_l(x):
        return add([f, h])

    return f
```

```
# input tensor is the 28x28 grayscale image
input_tensor = Input((28, 28, 1))

# first conv2d with post-activation to transform the input data to some reasonable form
x = Conv2D(kernel_size=3, filters=32, strides=1, padding='same', kernel_regularizer=regularizers.l2(0.01))(input_tensor)
x = Activation(relu)(x)

# F_1
x = block(32)(x)

# last activation of the entire network's output
x = Activation(relu)(x)

x = MaxPooling2D(pool_size=(2,2))(x)
x= Flatten()(x)

# dropout for more robust learning
x = Dropout(0.2)(x)

# last softmax layer
x = Dense(5, kernel_regularizer=regularizers.l2(0.01))(x)
x = Activation(softmax)(x)

model = Model(inputs=input_tensor, outputs=x)
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
lr = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```

```
batch_size = 128
epochs = 20
history_r = model.fit(x_train, y_train,batch_size= batch_size, epochs = epochs, verbose=2,
                              validation_data = (x_test, y_test), callbacks=[lr])
```

• **Explanation of model**

First define a residual block function which consists of two 3x3 convolutional layers and the shortcut connection. Then define the model, the first conv2d with post-activation is to transform the input data to some reasonable form and then add one residual block . After that adding pooling layer, flatten layer, dropout layer and output layer as normal CNN models. Other optimizers are still same as previous models. Here we tried only 20 epochs because of time issue.

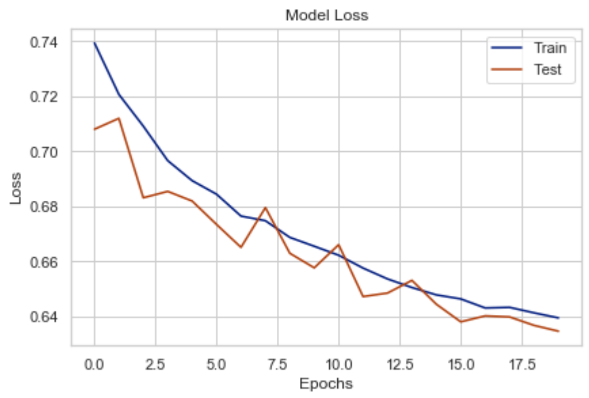
From the model summary below, we can see that there are 100,293 parameter in total that is even less than model 1. So a higher depth of CONV filters might help to improve this model.

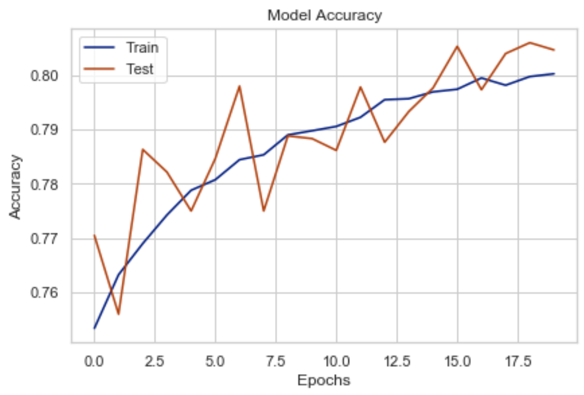
```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
input_6 (InputLayer)             (None, 28, 28, 1)     0
____________________________________________________________________________________________________
conv2d_10 (Conv2D)               (None, 28, 28, 64)    640         input_6[0][0]
____________________________________________________________________________________________________
activation_16 (Activation)       (None, 28, 28, 64)    0           conv2d_10[0][0]
____________________________________________________________________________________________________
activation_18 (Activation)       (None, 28, 28, 64)    0           activation_16[0][0]
____________________________________________________________________________________________________
conv2d_12 (Conv2D)               (None, 28, 28, 64)    36928       activation_18[0][0]
____________________________________________________________________________________________________
add_3 (Add)                      (None, 28, 28, 64)    0           activation_16[0][0]
                                                                   conv2d_12[0][0]
____________________________________________________________________________________________________
activation_19 (Activation)       (None, 28, 28, 64)    0           add_3[0][0]
____________________________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)   (None, 14, 14, 64)    0           activation_19[0][0]
____________________________________________________________________________________________________
flatten_2 (Flatten)              (None, 12544)         0           max_pooling2d_2[0][0]
____________________________________________________________________________________________________
dropout_2 (Dropout)              (None, 12544)         0           flatten_2[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                  (None, 5)             62725       dropout_2[0][0]
____________________________________________________________________________________________________
activation_20 (Activation)       (None, 5)             0           dense_2[0][0]
====================================================================================================
Total params: 100,293
Trainable params: 100,293
Non-trainable params: 0
____________________________________________________________________________________________________
```

• **Runtime performance for training and testing.**
The runtime of model5 for training is approximately 130 seconds per epoch which is more than previous model average runtimes. And the runtime for separated test set is 4 second. Since the quantity of total parameters has increased dramatically, the computation time are much more.

• **plots to explain the performance of your approach**.
From 20 epochs, the result accuracy is quite low that is only 0.8047 on validation set slightly less than model 5. As it shows in the following two plots, the left plot is model loss vs. epochs including curves for train set and test set. The loss of train set decreased from 0.74 to 0.639 as the epochs increase from 0 to 20. The loss of validation set also decreases but with more significant fluctuations and finally approaches to 0.6347.

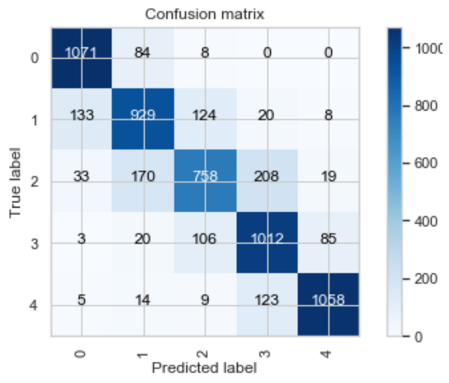
In the model accuracy plot, the accuracy of train set increases to 0.8003. The accuracy of validation set is gradually increasing but have obvious fluctuations and finally approaches to 0.8047.

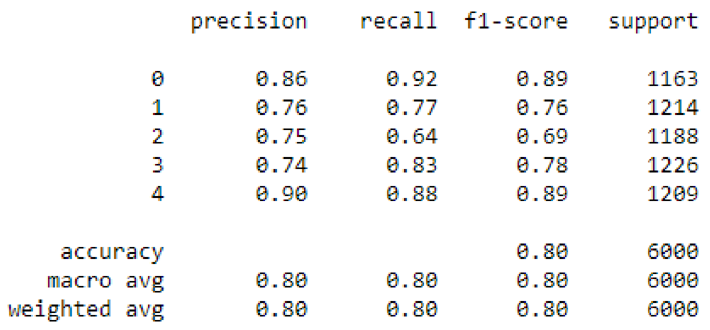




• **Evaluate your code with other metrics on the training data and argue for the benefit of you approach**.
Below is the performance of our classification model 6 on the validation data using Confusion Matrix. From the Confusion Matrix, almost every class have a large number of input have been misclassified. The predicting ability for class 1, 2and 3 is still much weaker than other two classes.

From the classification report at right below, we can see a summary of classification result. We can have similar results from f1-score that our model5 predicted only 76% of class1 , 69% of class2 correctly and 78% of class3 correctly which are less than percentage of other two class that both have a rate of 89%.




```
              precision    recall  f1-score   support

           0       0.86      0.92      0.89      1163
           1       0.76      0.77      0.76      1214
           2       0.75      0.64      0.69      1188
           3       0.74      0.83      0.78      1226
           4       0.90      0.88      0.89      1209

    accuracy                           0.80      6000
   macro avg       0.80      0.80      0.80      6000
weighted avg       0.80      0.80      0.80      6000
```

If we could have time to try more epochs and increase depth of filters might have a much better performance. Model 6 could be more suitable for this dataset although the final accuracy is lower than model 5. But at least the runtime performance of model6 is much more acceptable to go on exploring. This model6 is constructed based on Kaggle example" Tiny ResNet with Keras (99.314%)"(https://www.kaggle.com/meownoid/tiny-resnet-with-keras-99-314).

Kaggle Competition Score

The highest score of our submissions received on Kaggle is 0.91460 from Model 3.

Summary

We have tried several CNN and ResNets models, among those models Model 3 has the best accuracy performance. According to runtime, Model 2 has the best performance with a slightly lower accuracy. People could chose between model 2 and model 3 according to their preference and situation. And of course there might methods to improve model 2 and 3 that we do not know yet. The model6 might have a lot potential if we could have enough time to try more combinations.

Thanks for professor and Tas, and please consider our team for one team member was sick for 6 days in between. So we lost some time to go further in Resnet or other Inception models that might work better than CNN.