Dataset – Fashion MNIST

Fashion MNIST training dataset consists of 60,000 images of each 28 x 28 pixels and 1 channel (Grayscale) associated with a label from 5 classes. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. The pixel values in the dataset are already normalized to unit dimensions so the values lie between 0 to 1 (0 for white and 1 for black). This allows gradient descent to converge faster thereby reducing the training time.

[CM4] Design and Implementation Choices of your Model

4 CNN models with different architectures and parameters were implemented and their performances compared.

Model 1: 3 convolution layer

- 3 convolution layer (32 , 64 , 128 (3 x 3) filters)
- 2 max pooling layer (2 x 2)
- 1 fully connected layer (128 units)
- Adam optimizer

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320       
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496     
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 128)         73856     
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0         
_________________________________________________________________
dense (Dense)                (None, 128)               147584    
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 645       
=================================================================
Total params: 240,901
Trainable params: 240,901
Non-trainable params: 0
_________________________________________________________________
```

Model 2: 3 convolution layer with Dropout and Batch Normalization

- 3 convolution layer (32 , 64 , 128 (3 x 3) filters)
- 2 max pooling layer (2 x 2)
- 4 Batch Normalization layer
- 4 Dropout layer
- 1 fully connected layer (128 units)
- Adam optimizer

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_3 (Conv2D)            (None, 26, 26, 32)        320       
_________________________________________________________________
batch_normalization (BatchNo (None, 26, 26, 32)        128       
_________________________________________________________________
dropout (Dropout)            (None, 26, 26, 32)        0         
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 13, 13, 32)        0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 11, 11, 64)        18496     
_________________________________________________________________
batch_normalization_1 (Batch (None, 11, 11, 64)        256       
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 5, 5, 64)          0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 5, 5, 64)          0         
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 3, 3, 128)         73856     
_________________________________________________________________
batch_normalization_2 (Batch (None, 3, 3, 128)         512       
_________________________________________________________________
dropout_2 (Dropout)          (None, 3, 3, 128)         0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 1152)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 128)               147584    
_________________________________________________________________
batch_normalization_3 (Batch (None, 128)               512       
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 5)                 645       
=================================================================
Total params: 242,309
Trainable params: 241,605
Non-trainable params: 704
_________________________________________________________________
```

Model 3: 4 convolution layer with Batch normalization and Dropout

- 4 convolution layer (32 , 32 , 64 , 128 (3 x 3) filters)
- 2 max pooling layer (2 x 2)
- 6 Batch Normalization layer
- 5 Dropout layer
- 2 fully connected layer (512 , 128 units)
- Adam optimizer

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_6 (Conv2D)            (None, 26, 26, 32)        320       
_________________________________________________________________
```

```
batch_normalization_4 (Batch (None, 26, 26, 32)        128
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 24, 24, 32)        9248
_________________________________________________________________
batch_normalization_5 (Batch (None, 24, 24, 32)        128
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
dropout_4 (Dropout)          (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 10, 10, 64)        18496
_________________________________________________________________
batch_normalization_6 (Batch (None, 10, 10, 64)        256
_________________________________________________________________
dropout_5 (Dropout)          (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
batch_normalization_7 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
dropout_6 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               1049088
_________________________________________________________________
batch_normalization_8 (Batch (None, 512)               2048
_________________________________________________________________
dropout_7 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 128)               65664
_________________________________________________________________
batch_normalization_9 (Batch (None, 128)               512
_________________________________________________________________
dropout_8 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_6 (Dense)              (None, 5)                 645
=================================================================
Total params: 1,220,901
Trainable params: 1,219,109
Non-trainable params: 1,792
_________________________________________________________________
```

Model 4: 4 convolution layer with Batch Normalization, Dropout, same padding, and L2 regularization (Best model)

- 4 convolution layer (32 , 32 , 64 , 128 (3 x 3) filters, same padding, L2 kernel regularization)
- 2 max pooling layer (2 x 2)
- 6 Batch Normalization layer
- 5 Dropout layer
- 2 fully connected layer (512 , 128 units)
- Adam optimizer

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_10 (Conv2D)           (None, 28, 28, 32)        320       
_________________________________________________________________
batch_normalization_10 (Batc (None, 28, 28, 32)        128       
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 28, 28, 32)        9248      
_________________________________________________________________
batch_normalization_11 (Batc (None, 28, 28, 32)        128       
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 14, 14, 32)        0         
_________________________________________________________________
dropout_9 (Dropout)          (None, 14, 14, 32)        0         
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 14, 14, 64)        18496     
_________________________________________________________________
batch_normalization_12 (Batc (None, 14, 14, 64)        256       
_________________________________________________________________
dropout_10 (Dropout)         (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 14, 14, 128)       73856     
_________________________________________________________________
batch_normalization_13 (Batc (None, 14, 14, 128)       512       
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 7, 7, 128)         0         
_________________________________________________________________
dropout_11 (Dropout)         (None, 7, 7, 128)         0         
_________________________________________________________________
flatten_3 (Flatten)          (None, 6272)              0         
_________________________________________________________________
dense_7 (Dense)              (None, 512)               3211776   
_________________________________________________________________
batch_normalization_14 (Batc (None, 512)               2048      
_________________________________________________________________
dropout_12 (Dropout)         (None, 512)               0         
_________________________________________________________________
dense_8 (Dense)              (None, 128)               65664     
_________________________________________________________________
batch_normalization_15 (Batc (None, 128)               512       
_________________________________________________________________
dropout_13 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_9 (Dense)              (None, 5)                 645       
=================================================================
Total params: 3,383,589
Trainable params: 3,381,797
Non-trainable params: 1,792
_________________________________________________________________
```

Further, 2 pre-trained model were implemented using Transfer Learning

Transfer learning consists of taking features learned on one problem, and leveraging them on a new, similar problem.

There are many top-performing models for image recognition that can be downloaded and used as the basis for image recognition and related computer vision tasks. Keras provides access to a number of top-performing pre-trained models that were developed for image recognition tasks.

Here we use two of the more popular models:

- **Resnet 50**
- **VGG 16**

These models are both widely used for transfer learning both because of their performance, but also because they were examples that introduced specific architectural innovations, namely consistent and repeating structures (VGG), and residual modules (ResNet).

Transfer learning workflow can be implemented in Keras:

1. Instantiate a base model and load pre-trained weights (Imagenet in this case) into it.
2. When loading the pre-trained model, the "*include_top*" argument can be set to *False*, in which case the fully-connected output layers of the model used to make predictions is not loaded, allowing a new output layer to be added and trained
3. Freeze the weights of all layers in the base model by setting *trainable = False*.
4. Create a new model on top of the output of one (or several) layers from the base model. Here we add the Flatten layer and the output Dense layer for classification into 5 classes.
5. The image dataset is reshaped to 32 x 32 x 3 using zero padding, to fit the required input format of the pre-trained model.
6. Train the new model on the dataset.

CNN Model 4 (Best Model) design:

We use a **Sequential** model to build the CNN model.

The Sequential model is a linear stack of layers. It can be first initialized and then we add layers using add method or we can add all layers at initialization stage. The layers added are as follows:

1st Conv2D is a 2D Convolutional layer (i.e. spatial convolution over images). The parameters used are:

- filters - the number of filters (Kernels) used with this layer; here filters = 32;
- kernel_size - the dimension of the Kernel: (3 x 3);
- padding – **Padding** is the process of adding layers of zeros to our input images so as to avoid shrinkage and loss of information from corners of the image. In this case, we apply "same" padding, such that the output image has the same dimensions as the input image.
- activation - is the activation function used, in this case Rectified Linear Activation (ReLU / relu); Relu is popular and has been proven to work well in neural networks. The **rectified linear activation** function or **ReLU** is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero.
- kernel_initializer – 'he_normal' the function used for initializing the kernel;
- kernel_regularizer - Regularizer to apply a penalty on the layer's kernel. In this case we apply an L2 regularization penalty. **Regularization** controls the model complexity by penalizing higher terms in the model. We can prevent a model from overfitting by controlling the complexity.

- input_shape - is the shape of the image presented to the CNN: in our case is 28 x 28. The input and output of the Conv2D is a 4D tensor.

Batch normalization, as the name suggests, seeks to normalize the activations of a given input volume before passing it into the next layer. It has been shown to be effective at reducing the number of epochs required to train a CNN at the expense of an increase in per-epoch time. The CNN model has a total of 6 Batch Normalization layers.

Dropout is a form of regularization that aims to prevent overfitting. Random connections are dropped to ensure that no single node in the network is responsible for activating when presented with a given pattern. The CNN model has a total of 5 dropout layers. The 1st & 2nd dropout layer will randomly disable 30% of the outputs, while the 3rd dropout layer disables 40%, and the 4th & 5th layers will disable 50% of the respective outputs.

2nd Conv2D with the following parameters:

- filters: 32;
- kernel_size : (3 x 3);
- padding : same
- activation : relu;
- kernel_regularizer: L2

1st MaxPooling2D is a Max pooling operation for spatial data. Pooling layers help to progressively reduce the spatial dimensions of the input volume. Parameters used here are:

- pool_size, in this case (2,2), representing the factors by which to downscale in both directions;

3rd Conv2D with the following parameters:

- filters: 64;
- kernel_size : (3 x 3);
- padding : same
- activation : relu;
- kernel_regularizer: L2

2nd MaxPooling2D with parameter:

- pool_2nd size : (2,2);

4th Conv2D with the following parameters:

- filters: 128;
- kernel_size : (3 x 3);
- padding : same
- activation : relu;
- kernel_regularizer: L2

To complete our model, we need to feed the last output tensor from the convolutional base into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, we flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. The Fashion MNIST dataset has 5 output classes, so you use a final Dense layer with 5 outputs.

Flatten. This layer Flattens the input. Does not affect the batch size. It is used without parameters. Flatten serves as a connection between the convolution and dense layers;

Dense. This layer is a regular fully-connected NN layer. It is used without parameters;

- units - this is a positive integer, with the meaning: dimensionality of the output space; in this case is: 512;
- activation - activation function : relu;

Dense. This layer is a regular fully-connected NN layer. It is used without parameters;

- units - 128;
- activation - activation function : relu;

Dense. This is the final layer (fully connected). It is used with the parameters:

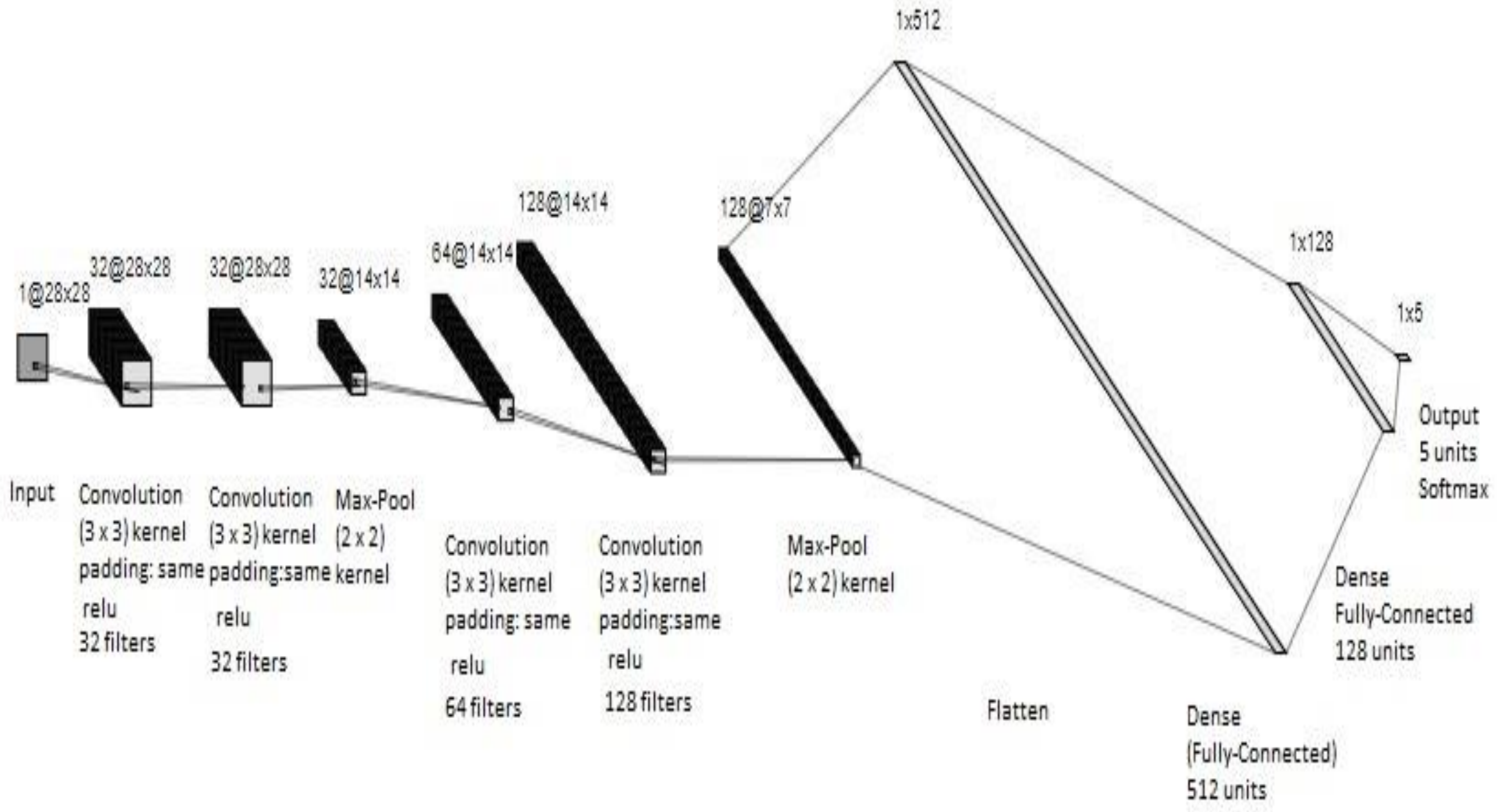
- units: the number of classes (in our case 5, one for each possible outcome)
- activation: softmax; for this final layer it is used softmax activation (standard for multiclass classification). **Softmax** makes the output sum up to 1 so the output can be interpreted as probabilities.

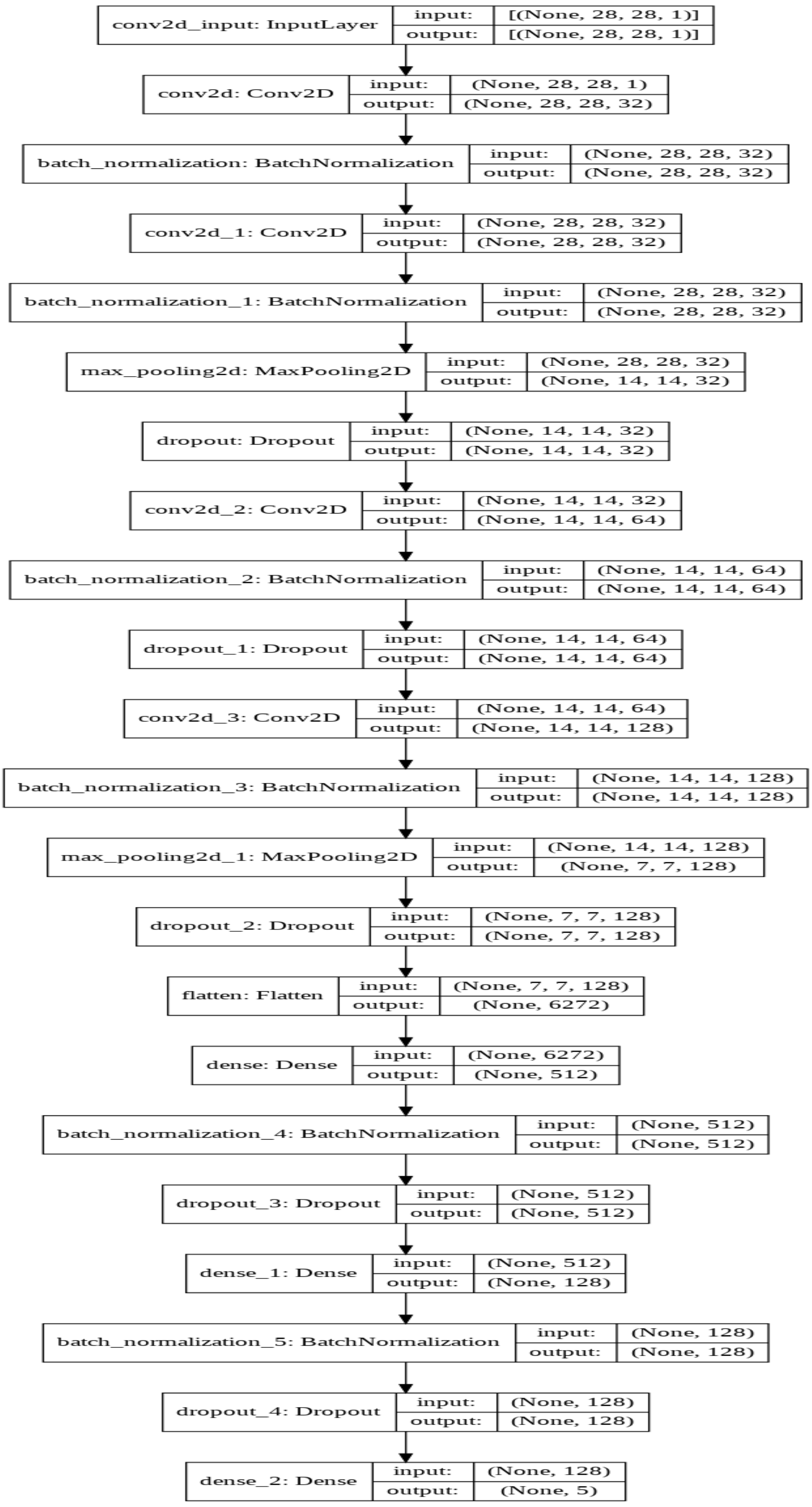
Then we compile the model, specifying as well the following parameters:

- loss - cross-entropy;
- optimizer - Adam;
- metrics - accuracy.

Optimizer is the algorithm used by the model to update weights of every layer after every iteration. SGD, RMSProp and Adam are the popular choices. SGD works well for shallow networks but cannot escape saddle points and local minima, whereas RMSProp could be a better choice in such cases. AdaDelta/AdaGrad for sparse data, whereas Adam is a general favourite and could be used to achieve faster convergence. Hence, we use Adam for our model.

Model Visualization




conv2d_input: InputLayer | input: [(None, 28, 28, 1)] | output: [(None, 28, 28, 1)]
conv2d: Conv2D | input: (None, 28, 28, 1) | output: (None, 28, 28, 32)
batch_normalization: BatchNormalization | input: (None, 28, 28, 32) | output: (None, 28, 28, 32)
conv2d_1: Conv2D | input: (None, 28, 28, 32) | output: (None, 28, 28, 32)
batch_normalization_1: BatchNormalization | input: (None, 28, 28, 32) | output: (None, 28, 28, 32)
max_pooling2d: MaxPooling2D | input: (None, 28, 28, 32) | output: (None, 14, 14, 32)
dropout: Dropout | input: (None, 14, 14, 32) | output: (None, 14, 14, 32)
conv2d_2: Conv2D | input: (None, 14, 14, 32) | output: (None, 14, 14, 64)
batch_normalization_2: BatchNormalization | input: (None, 14, 14, 64) | output: (None, 14, 14, 64)
dropout_1: Dropout | input: (None, 14, 14, 64) | output: (None, 14, 14, 64)
conv2d_3: Conv2D | input: (None, 14, 14, 64) | output: (None, 14, 14, 128)
batch_normalization_3: BatchNormalization | input: (None, 14, 14, 128) | output: (None, 14, 14, 128)
max_pooling2d_1: MaxPooling2D | input: (None, 14, 14, 128) | output: (None, 7, 7, 128)
dropout_2: Dropout | input: (None, 7, 7, 128) | output: (None, 7, 7, 128)
flatten: Flatten | input: (None, 7, 7, 128) | output: (None, 6272)
dense: Dense | input: (None, 6272) | output: (None, 512)
batch_normalization_4: BatchNormalization | input: (None, 512) | output: (None, 512)
dropout_3: Dropout | input: (None, 512) | output: (None, 512)
dense_1: Dense | input: (None, 512) | output: (None, 128)
batch_normalization_5: BatchNormalization | input: (None, 128) | output: (None, 128)
dropout_4: Dropout | input: (None, 128) | output: (None, 128)
dense_2: Dense | input: (None, 128) | output: (None, 5)

[CM5] Implementation of your Design Choices

Loading the dataset

```
dataset = np.load('fashion_mnist_dataset_train.npy', allow_pickle=True).item()
```

```
X = dataset['features']
y = dataset['target']
```

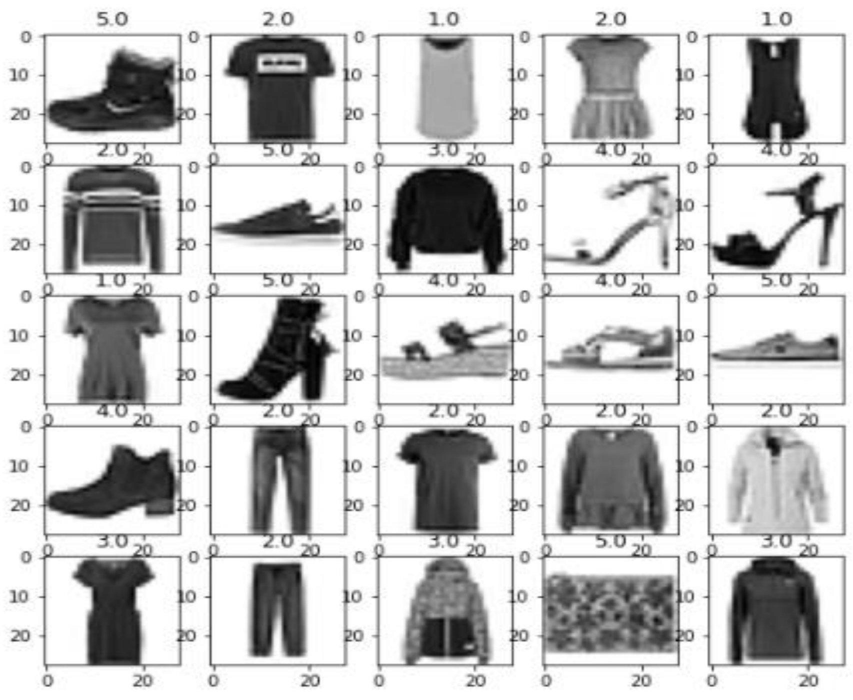
```
X.shape, y.shape
```

```
((60000, 28, 28), (60000,))
```

Visualizing the Fashion MNIST dataset.

```
def display_picture(images, labels):
        f, ax = plt.subplots(5,5, figsize=(8,8))
        for i in range(25):
            ax[i//5, i%5].imshow(images[i], cmap=plt.cm.binary)
            ax[i//5, i%5].set_title(labels[i])
        plt.show()
```

```
display_picture(X, y)
```



Pre-processing the dataset.

The pixel values (“features”) in the dataset are already normalized to unit dimensions so the values lie between 0 to 1 (0 for white and 1 for black). The class labels (“target”) are ranging from 1-5, so they need to be updated to 0-4, and also one hot encoded transforming the integer into a 5 element binary vector with a 1 for the index of the class value, for the final classification layer to predict their respective probabilities.

```
from tensorflow.keras.utils import to_categorical
y = y-1
y = to_categorical(y, 5)
y.shape
```

```
(60000, 5)
```

Splitting the dataset into Train, Validation and Test set (80% - 10% - 10%) with random seed as 0.

```
from sklearn.model_selection import train_test_split

# train - val - test split (80%-10%-10% , random seed =0)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size=0.5, random_state=0)
```

Reshaping the image dimensions to fit the CNN model. The channel dimension of 1 (Greyscale) needs to be added to the image dimensions of 28 x 28.

```
X_train = tf.expand_dims(X_train, axis=-1)
X_val = tf.expand_dims(X_val, axis=-1)
X_test = tf.expand_dims(X_test, axis=-1)
```

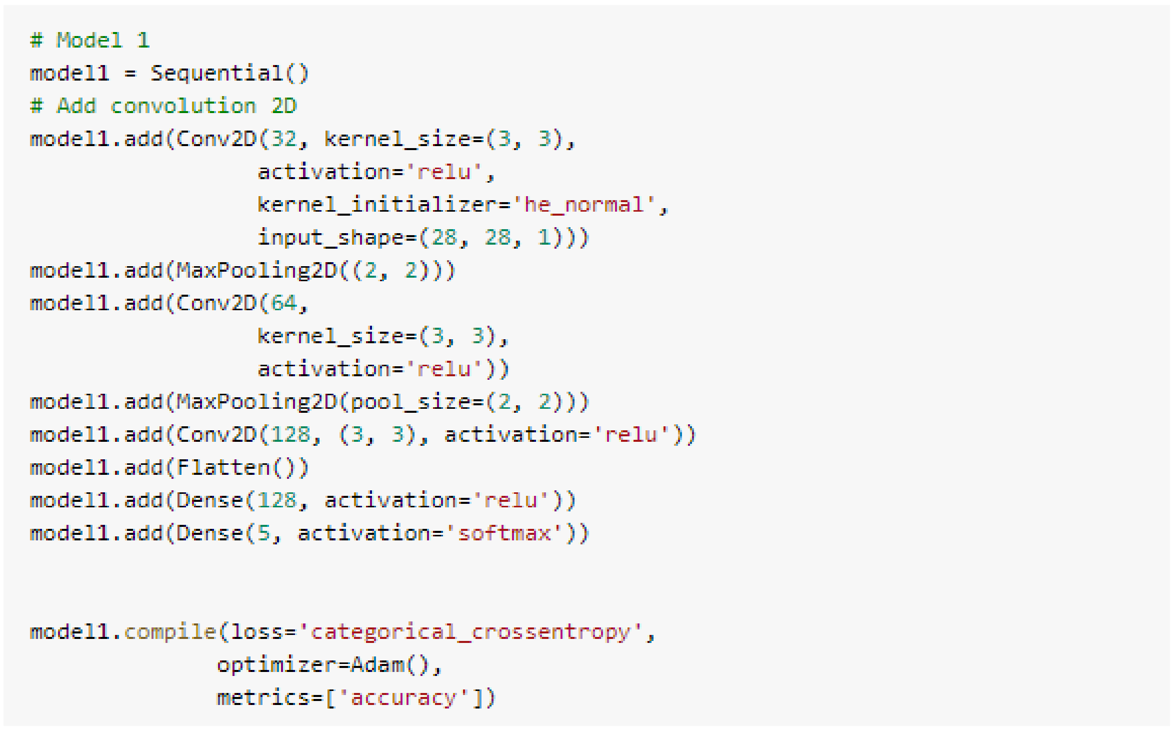
```
X_train.shape, X_val.shape, X_test.shape
```

```
(TensorShape([48000, 28, 28, 1]),
 TensorShape([6000, 28, 28, 1]),
 TensorShape([6000, 28, 28, 1]))
```

Defining and compiling the CNN models

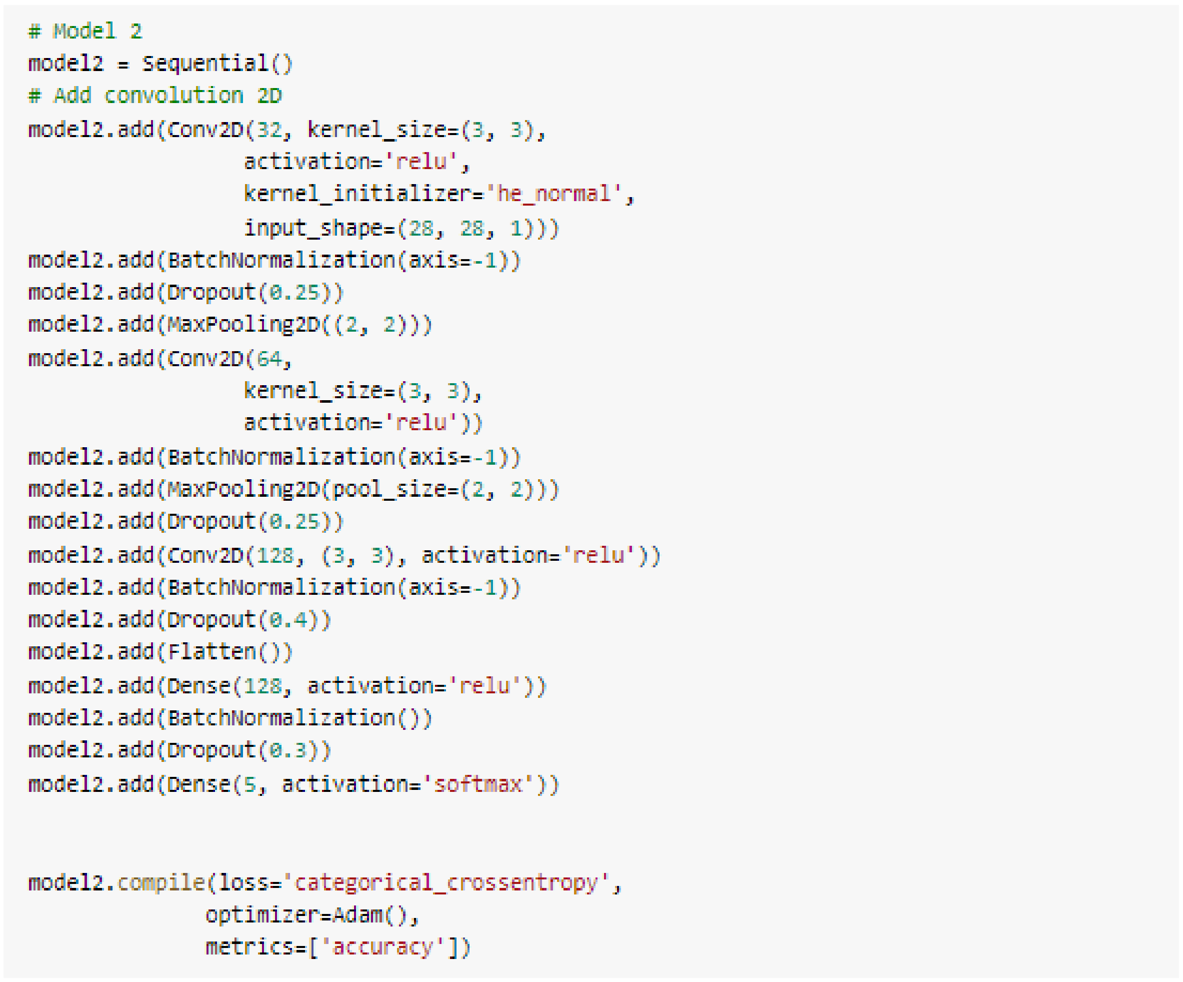
```
IMG_ROWS = 28
IMG_COLS = 28
CHANNEL = 1
NUM_CLASSES = 5
TEST_SIZE = 0.2
RANDOM_STATE = 0

#Model
EPOCHS = 50
BATCH_SIZE = 128
```

Model 1: 3 convolution layer



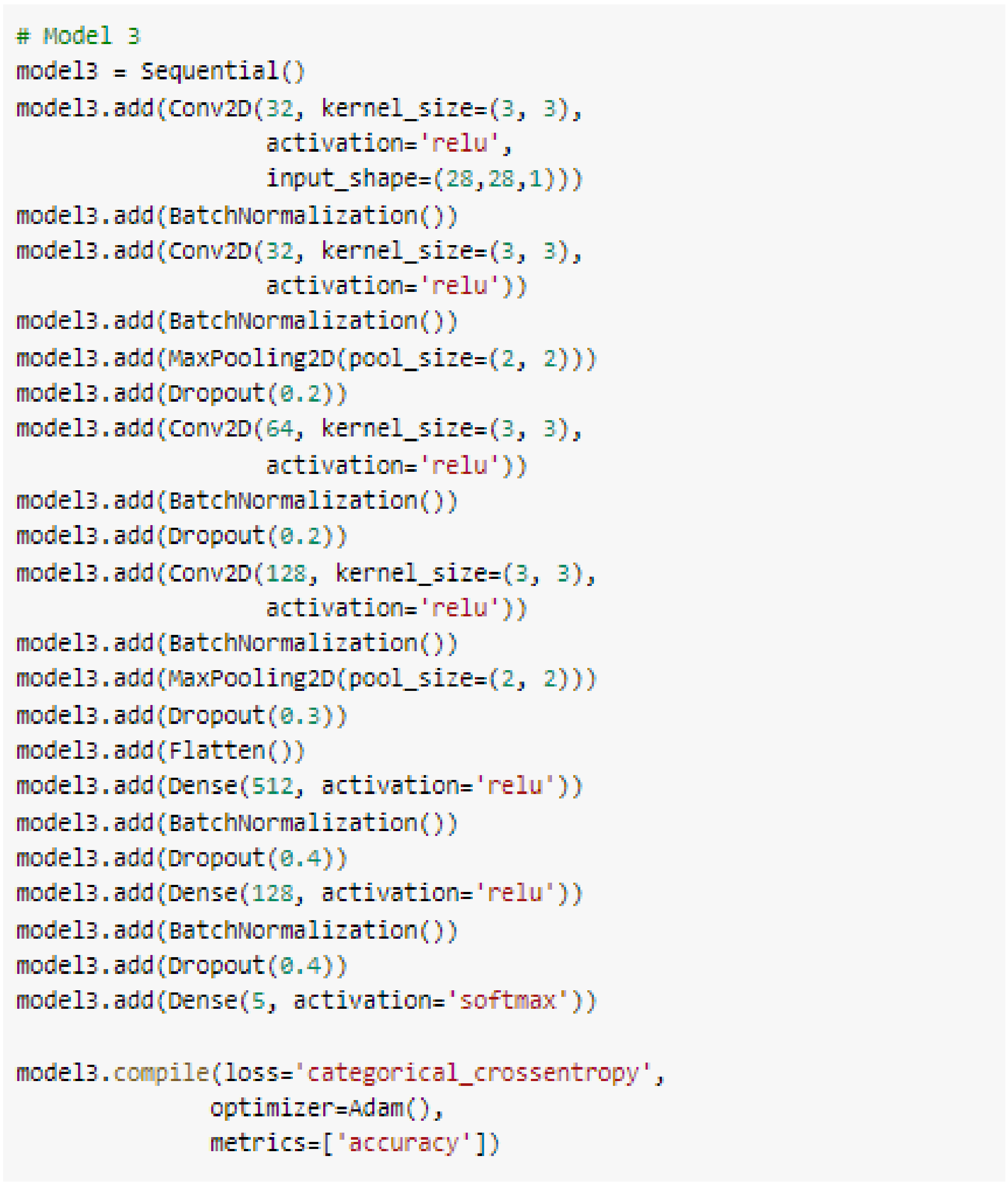
```
# Model 1
model1 = Sequential()
# Add convolution 2D
model1.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 kernel_initializer='he_normal',
                 input_shape=(28, 28, 1)))
model1.add(MaxPooling2D((2, 2)))
model1.add(Conv2D(64,
                 kernel_size=(3, 3),
                 activation='relu'))
model1.add(MaxPooling2D(pool_size=(2, 2)))
model1.add(Conv2D(128, (3, 3), activation='relu'))
model1.add(Flatten())
model1.add(Dense(128, activation='relu'))
model1.add(Dense(5, activation='softmax'))


model1.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
```

Model 2: 3 convolution layer with Dropout and Batch Normalization



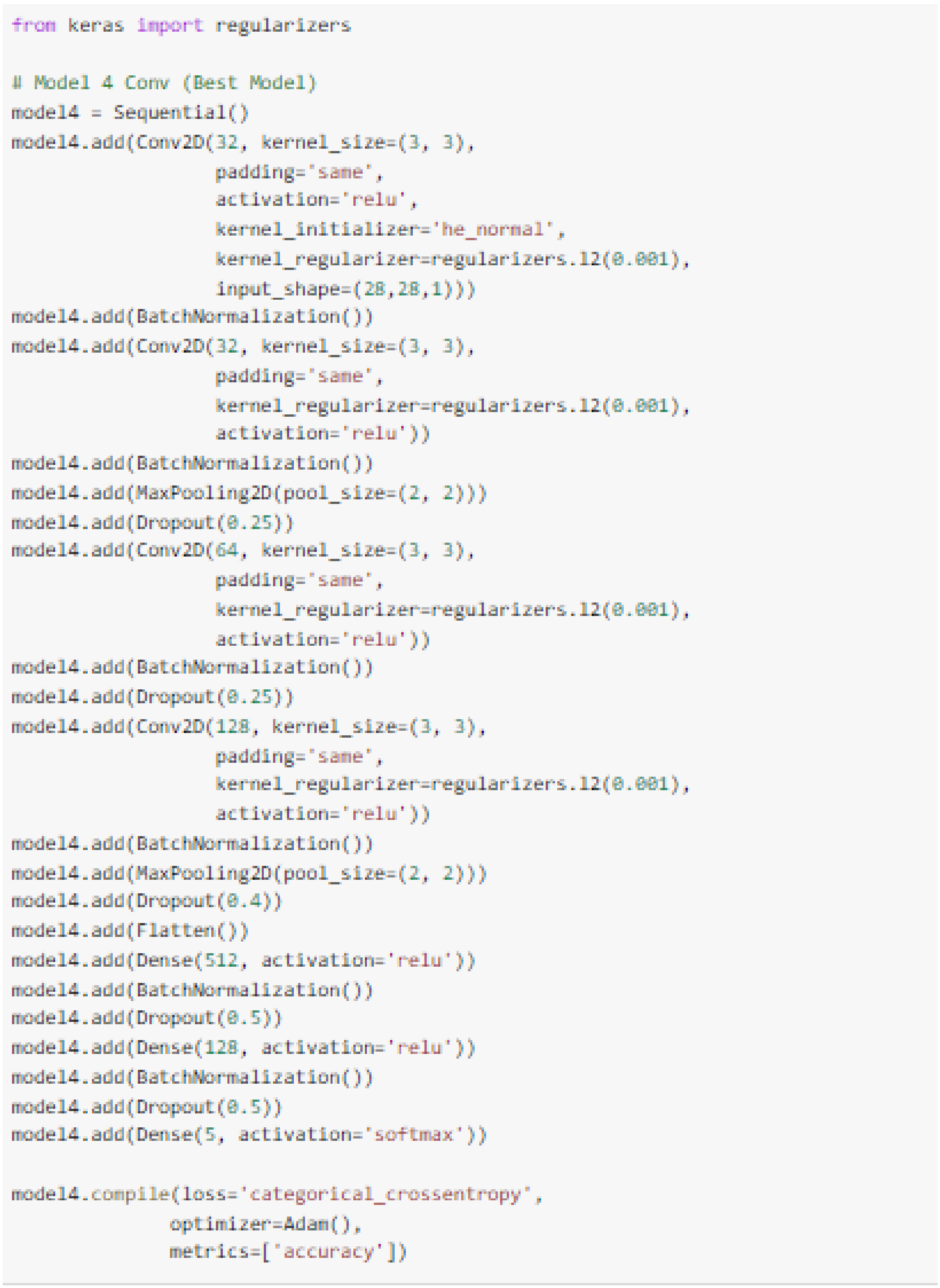
```
# Model 2
model2 = Sequential()
# Add convolution 2D
model2.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 kernel_initializer='he_normal',
                 input_shape=(28, 28, 1)))
model2.add(BatchNormalization(axis=-1))
model2.add(Dropout(0.25))
model2.add(MaxPooling2D((2, 2)))
model2.add(Conv2D(64,
                 kernel_size=(3, 3),
                 activation='relu'))
model2.add(BatchNormalization(axis=-1))
model2.add(MaxPooling2D(pool_size=(2, 2)))
model2.add(Dropout(0.25))
model2.add(Conv2D(128, (3, 3), activation='relu'))
model2.add(BatchNormalization(axis=-1))
model2.add(Dropout(0.4))
model2.add(Flatten())
model2.add(Dense(128, activation='relu'))
model2.add(BatchNormalization())
model2.add(Dropout(0.3))
model2.add(Dense(5, activation='softmax'))


model2.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
```

Model 3: 4 convolution layer with Batch normalization and Dropout



```
# Model 3
model3 = Sequential()
model3.add(Conv2D(32, kernel_size=(3, 3),
                  activation='relu',
                  input_shape=(28,28,1)))
model3.add(BatchNormalization())
model3.add(Conv2D(32, kernel_size=(3, 3),
                  activation='relu'))
model3.add(BatchNormalization())
model3.add(MaxPooling2D(pool_size=(2, 2)))
model3.add(Dropout(0.2))
model3.add(Conv2D(64, kernel_size=(3, 3),
                  activation='relu'))
model3.add(BatchNormalization())
model3.add(Dropout(0.2))
model3.add(Conv2D(128, kernel_size=(3, 3),
                  activation='relu'))
model3.add(BatchNormalization())
model3.add(MaxPooling2D(pool_size=(2, 2)))
model3.add(Dropout(0.3))
model3.add(Flatten())
model3.add(Dense(512, activation='relu'))
model3.add(BatchNormalization())
model3.add(Dropout(0.4))
model3.add(Dense(128, activation='relu'))
model3.add(BatchNormalization())
model3.add(Dropout(0.4))
model3.add(Dense(5, activation='softmax'))

model3.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
```

Model 4: 4 convolution layer with Batch Normalization, Dropout, same padding, and L2 regularization (Best model)



```
from keras import regularizers

# Model 4 Conv (Best Model)
model4 = Sequential()
model4.add(Conv2D(32, kernel_size=(3, 3),
                  padding='same',
                  activation='relu',
                  kernel_initializer='he_normal',
                  kernel_regularizer=regularizers.l2(0.001),
                  input_shape=(28,28,1)))
model4.add(BatchNormalization())
model4.add(Conv2D(32, kernel_size=(3, 3),
                  padding='same',
                  kernel_regularizer=regularizers.l2(0.001),
                  activation='relu'))
model4.add(BatchNormalization())
model4.add(MaxPooling2D(pool_size=(2, 2)))
model4.add(Dropout(0.25))
model4.add(Conv2D(64, kernel_size=(3, 3),
                  padding='same',
                  kernel_regularizer=regularizers.l2(0.001),
                  activation='relu'))
model4.add(BatchNormalization())
model4.add(Dropout(0.25))
model4.add(Conv2D(128, kernel_size=(3, 3),
                  padding='same',
                  kernel_regularizer=regularizers.l2(0.001),
                  activation='relu'))
model4.add(BatchNormalization())
model4.add(MaxPooling2D(pool_size=(2, 2)))
model4.add(Dropout(0.4))
model4.add(Flatten())
model4.add(Dense(512, activation='relu'))
model4.add(BatchNormalization())
model4.add(Dropout(0.5))
model4.add(Dense(128, activation='relu'))
model4.add(BatchNormalization())
model4.add(Dropout(0.5))
model4.add(Dense(5, activation='softmax'))

model4.compile(loss='categorical_crossentropy',
               optimizer=Adam(),
               metrics=['accuracy'])
```

Transfer learning using pre-trained Resnet50 and VGG16 models

Resizing the input image shape to fit the pre-trained model requirement

```
# Resize the images 32*32 as required by ResNet50
# pad zeros to make 28*28*1 array to 32*32*3 array
X_train = np.pad(X_train,((0,0),(2,2),(2,2),(1,1)), mode='constant', constant_values=0)
X_val = np.pad(X_val,((0,0),(2,2),(2,2),(1,1)), mode='constant', constant_values=0)
X_test = np.pad(X_test,((0,0),(2,2),(2,2),(1,1)), mode='constant', constant_values=0)

X_train.shape, X_val.shape, X_test.shape
```

```
((48000, 32, 32, 3), (6000, 32, 32, 3), (6000, 32, 32, 3))
```

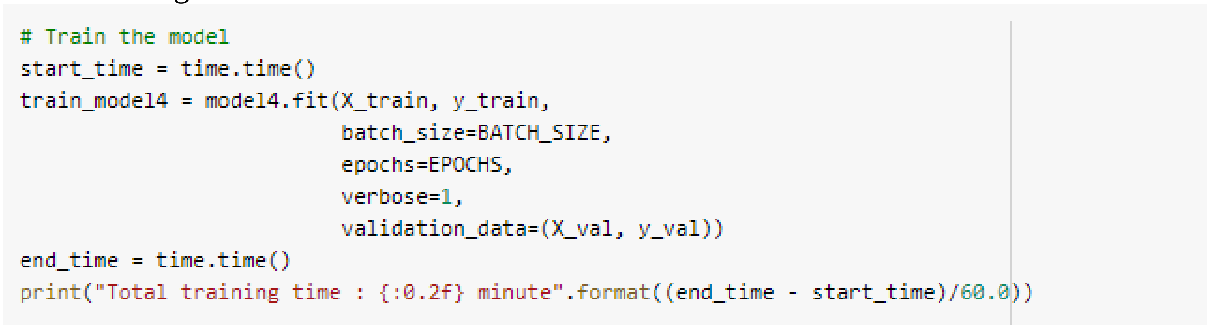
Resnet50

```
IMAGE_SIZE = [32, 32,3]
NUM_CLASSES = 5

# add preprocessing layer to the front of Resnet
resnet = ResNet50(input_shape=IMAGE_SIZE, weights='imagenet', include_top=False)
# don't train existing weights(Freeze the base model)
for layer in resnet.layers:
  layer.trainable = False

# adding flatten and output layer to the resnet architecture
x = Flatten()(resnet.output)
prediction = Dense(NUM_CLASSES, activation='softmax')(x)

# create a model object
model = Model(inputs=resnet.input, outputs=prediction)

# Compile the model
model.compile(loss='categorical_crossentropy',
             optimizer='adam',
             metrics=['accuracy'])
```

VGG16

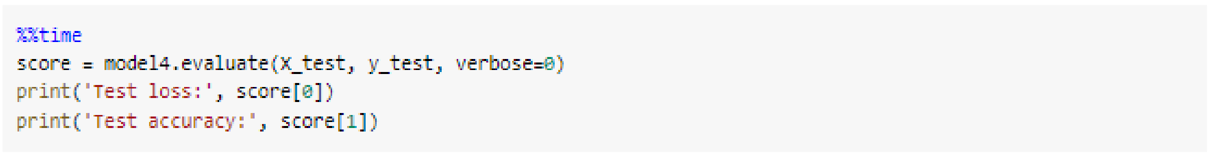
```
IMAGE_SIZE = [32, 32,3]
NUM_CLASSES = 5

# add preprocessing layer to the front of VGG
vgg_net = VGG16(input_shape=IMAGE_SIZE, weights='imagenet', include_top=False)

# don't train existing weights
for layer in vgg_net.layers:
  layer.trainable = False

# adding flatten and output layer to the resnet architecture
x = Flatten()(vgg_net.output)
prediction = Dense(NUM_CLASSES, activation='softmax')(x)

# create a model object
model = Model(inputs=vgg_net.input, outputs=prediction)

# Compile the model
model.compile(loss='categorical_crossentropy',
             optimizer='adam',
             metrics=['accuracy'])
```

Training the model to fit the processed Fashion MNIST dataset with Batch-Size of 128, and Epochs 50. The start time and the end time is noted to calculate the total time taken for model training.

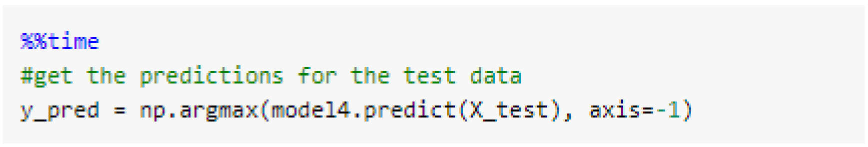


```
# Train the model
start_time = time.time()
train_model4 = model4.fit(X_train, y_train,
                          batch_size=BATCH_SIZE,
                          epochs=EPOCHS,
                          verbose=1,
                          validation_data=(X_val, y_val))
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0))
```

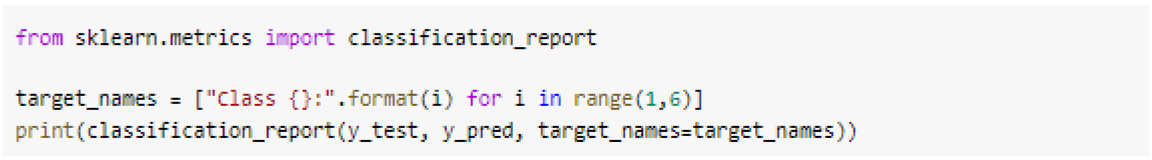
Performance of the trained model (best model) on the Test set



```
%%time
score = model4.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

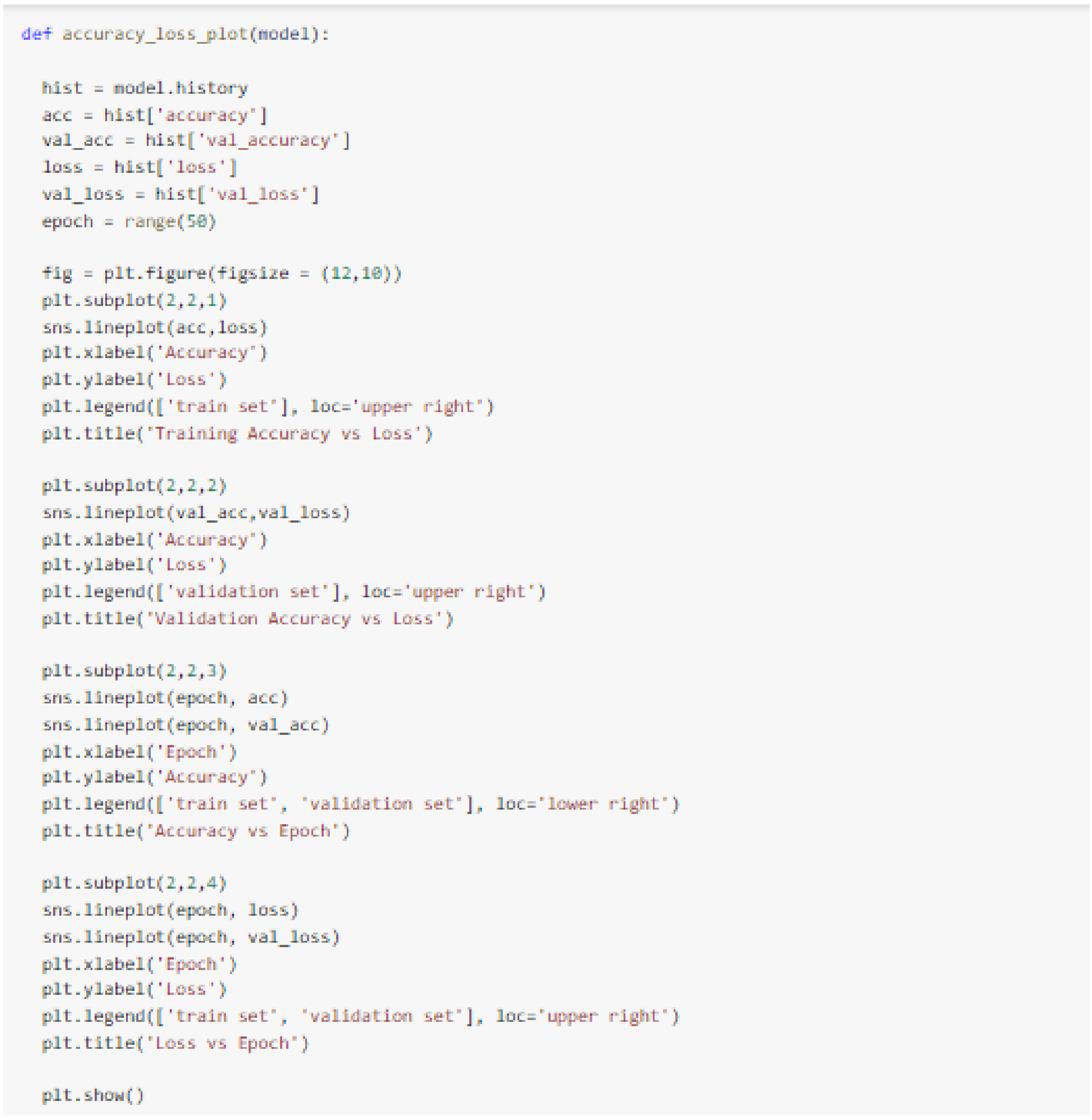


```
%%time
#get the predictions for the test data
y_pred = np.argmax(model4.predict(X_test), axis=-1)
```



```
from sklearn.metrics import classification_report

target_names = ["Class {}:".format(i) for i in range(1,6)]
print(classification_report(y_test, y_pred, target_names=target_names))
```

Method for Accuracy –Loss plots for the trained model



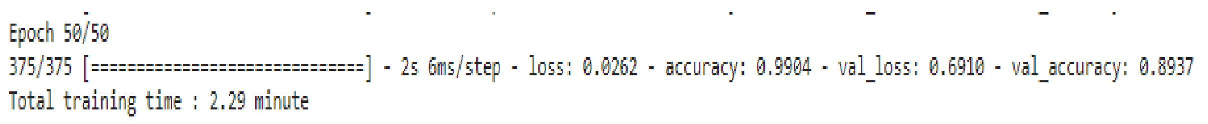
```python
def accuracy_loss_plot(model):

  hist = model.history
  acc = hist['accuracy']
  val_acc = hist['val_accuracy']
  loss = hist['loss']
  val_loss = hist['val_loss']
  epoch = range(50)

  fig = plt.figure(figsize = (12,10))
  plt.subplot(2,2,1)
  sns.lineplot(acc,loss)
  plt.xlabel('Accuracy')
  plt.ylabel('Loss')
  plt.legend(['train set'], loc='upper right')
  plt.title('Training Accuracy vs Loss')

  plt.subplot(2,2,2)
  sns.lineplot(val_acc,val_loss)
  plt.xlabel('Accuracy')
  plt.ylabel('Loss')
  plt.legend(['validation set'], loc='upper right')
  plt.title('Validation Accuracy vs Loss')

  plt.subplot(2,2,3)
  sns.lineplot(epoch, acc)
  sns.lineplot(epoch, val_acc)
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
  plt.legend(['train set', 'validation set'], loc='lower right')
  plt.title('Accuracy vs Epoch')

  plt.subplot(2,2,4)
  sns.lineplot(epoch, loss)
  sns.lineplot(epoch, val_loss)
  plt.xlabel('Epoch')
  plt.ylabel('Loss')
  plt.legend(['train set', 'validation set'], loc='upper right')
  plt.title('Loss vs Epoch')

  plt.show()
```

[CM6] Results Analysis

Run-time performance for training and testing

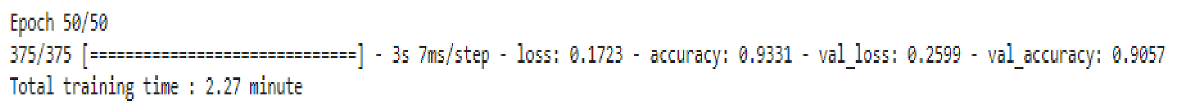
- Training time for all the models for Batch-Size 128 and 50 epochs

Model 1:



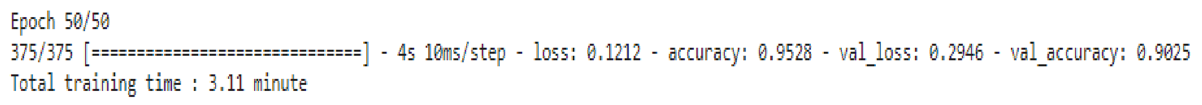
```
Epoch 50/50
375/375 [==============================] - 2s 6ms/step - loss: 0.0262 - accuracy: 0.9904 - val_loss: 0.6910 - val_accuracy: 0.8937
Total training time : 2.29 minute
```

Model 2:



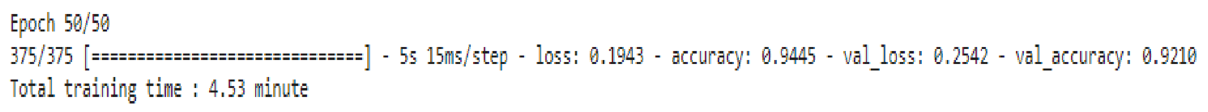
```
Epoch 50/50
375/375 [==============================] - 3s 7ms/step - loss: 0.1723 - accuracy: 0.9331 - val_loss: 0.2599 - val_accuracy: 0.9057
Total training time : 2.27 minute
```

Model 3:



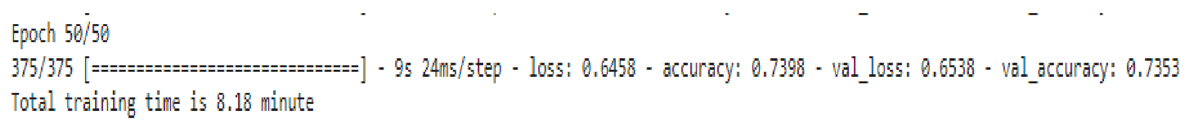
```
Epoch 50/50
375/375 [==============================] - 4s 10ms/step - loss: 0.1212 - accuracy: 0.9528 - val_loss: 0.2946 - val_accuracy: 0.9025
Total training time : 3.11 minute
```

Model 4:



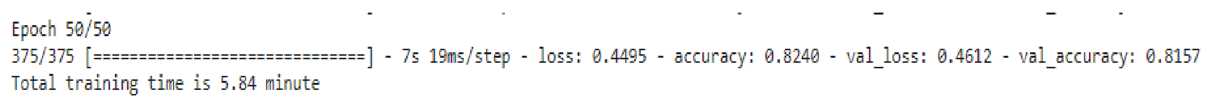
```
Epoch 50/50
375/375 [==============================] - 5s 15ms/step - loss: 0.1943 - accuracy: 0.9445 - val_loss: 0.2542 - val_accuracy: 0.9210
Total training time : 4.53 minute
```

Resnet50:



```
Epoch 50/50
375/375 [==============================] - 9s 24ms/step - loss: 0.6458 - accuracy: 0.7398 - val_loss: 0.6538 - val_accuracy: 0.7353
Total training time is 8.18 minute
```

VGG16:



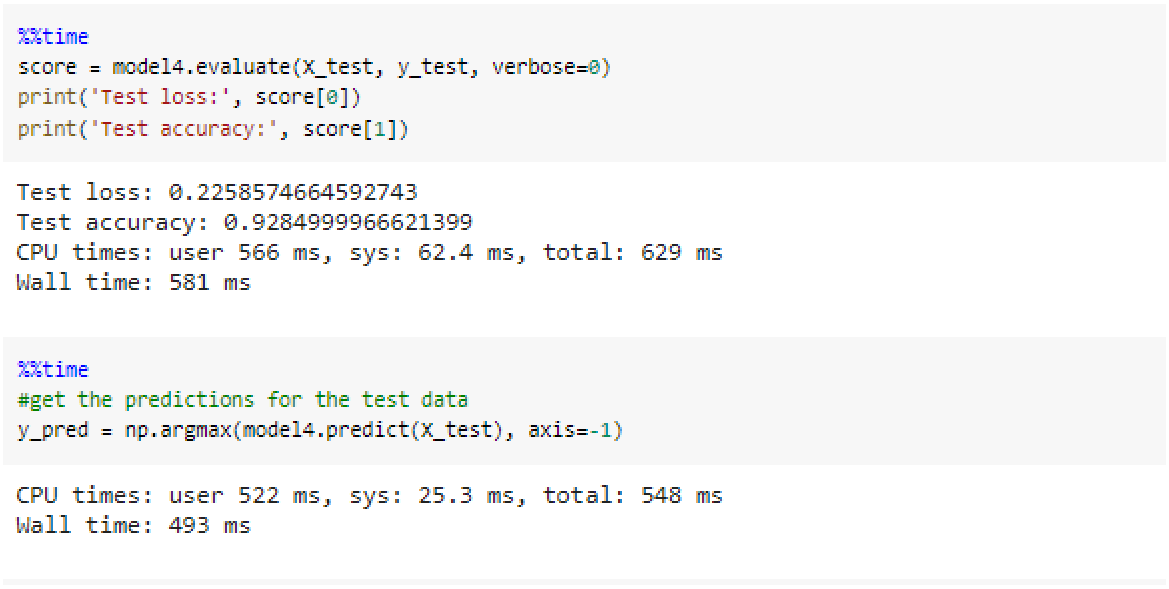
```
Epoch 50/50
375/375 [==============================] - 7s 19ms/step - loss: 0.4495 - accuracy: 0.8240 - val_loss: 0.4612 - val_accuracy: 0.8157
Total training time is 5.84 minute
```

The training cost/time for CNN models mainly depends on the model complexity (depth, parameters etc.) and the training sample size.

From the above data, we observe that:

- As the model complexity increases (number of layers, dropouts, normalization, regularization etc.), the total training time increases as well.
- The training time increases as the number of trainable parameters increase.
- Resnet is seen to take the longest training time as it is very deep and has a lot of parameters to be processed.

- Testing time of Model 4



```
%%time
score = model4.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.2258574664592743
Test accuracy: 0.9284999966621399
CPU times: user 566 ms, sys: 62.4 ms, total: 629 ms
Wall time: 581 ms
```

```
%%time
#get the predictions for the test data
y_pred = np.argmax(model4.predict(X_test), axis=-1)
```

```
CPU times: user 522 ms, sys: 25.3 ms, total: 548 ms
Wall time: 493 ms
```

The model takes 581 ms to evaluate on the test dataset, and 493 ms to predict the classes of the test dataset images.

Comparison of the different algorithms and parameters

Model	**Layers**	**Optimizer**	**Train Accuracy**	**Val Accuracy**	**Train Loss**	**Val Loss**	**Test Accuracy**	**Training Time**
Model 1	3	Adam	0.9953	0.8937	0.0141	0.6910	0.9017	2.29 minute
Model 2	3	Adam	0.9527	0.9057	0.1224	0.2599	0.9148	2.27 minute
Model 3	4	Adam	0.9605	0.9025	0.0901	0.2946	0.9061	3.11 minute
Model 4	**4**	**Adam**	**0.9674**	**0.9210**	**0.1309**	**0.2542**	**0.9285**	**4.53 minute**
Resnet 50		Adam	0.7398	0.7350	0.6461	0.6525	0.7443	8.28 minute
VGG 16		Adam	0.8244	0.8176	0.4486	0.4596	0.8182	5.74 minute

Table 1: Model performance for different algorithms and parameters

From Table 1, we observe that:

- Model 1 with 3 convolution layers, though has a high training accuracy of 99%, is not able to generalize and overfits a lot.
- Model 2 with 3 convolution layers and additional Dropout and Batch Normalization layers has managed to reduce the overfitting, but the accuracy has reduced aswell.
- Model 3 with 4 convolution layers and additional Dropout and Batch Normalization layers shows good training accuracy but still overfits.
- Model 4 with 4 convolution layers with "same" padding, L2 regularization, additional Dropout and Batch Normalization layer has the best training accuracy without significant overfitting, and hence is able to generalize better. The model can be further improved with methods like learning rate optimization, early stopping, hyperparameter tuning using gradient boosting or random search.
- The pre-trained models, Resnet50 and VGG16, though do not overfit, have less training accuracy, and also consume more time for training.

Comparing different parameters on Model 4 (Best Model)

Optimizer	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy
SGD	0.9101	0.8982	0.2492	0.2828	0.9008
Adam	**0.9674**	**0.9210**	**0.1309**	**0.2542**	**0.9285**

Table 2: Model 4 Performance of Optimizers (Epochs -50, Batch Size- 128)

From Table 2, upon trying different optimizers (SGD and Adam), we observe that though model with SGD optimizer overfits less, but has lower training accuracy compared to Adam. Thus, Adam is a better optimizer for the model.

Batch Size	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy	Training Time
16	0.9407	0.9163	0.2387	0.3057	0.9238	13.88 minute
32	0.9428	0.9142	0.2043	0.2928	0.9117	8.36 minute
64	0.9581	0.9128	0.1691	0.2910	0.9218	5.91 minute
128	**0.9674**	**0.9210**	**0.1309**	**0.2542**	**0.9285**	**4.53 minute**

Table 3: Model 4 Batch Size Experimentation (Epochs -50, Optimizer- Adam)

Batch size is a number of samples processed before the model is updated. From Table 3, we observe that the training time increases as the batch size reduces. Using larger batch sizes would allow the model to parallelize computations to a greater extent, in turn increasing the speed of model training significantly. We get the best accuracy for Batch Size of 128.

Learning Rate	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy	Training Time
0.01	0.8488	0.8359	0.5689	0.5465	0.8349	5.22 minute
0.02	0.8772	0.8657	0.3587	0.3907	0.8650	4.69 minute
0.001 (default)	**0.9674**	**0.9210**	**0.1309**	**0.2542**	**0.9285**	**4.53 minute**
0.002	0.9592	0.9207	0.1738	0.2776	0.9265	4.73 minute

Table 4: Model 4 Learning Rate Experimentation

The learning rate is a hyperparameter that determines the amount of change in the model in response to the estimated error each time the model weights are updated. A small learning rate value may result in a long training process that could get stuck, whereas a value too large may lead to learning a sub-optimal set of weights too fast or an unstable training process. From Table 4, we observe that the model has a low training accuracy for higher learning rate. Also, as the learning rate decreases, the training time decreases. We observe the best training accuracy for the default training rate (0.001) for Adam optimizer.

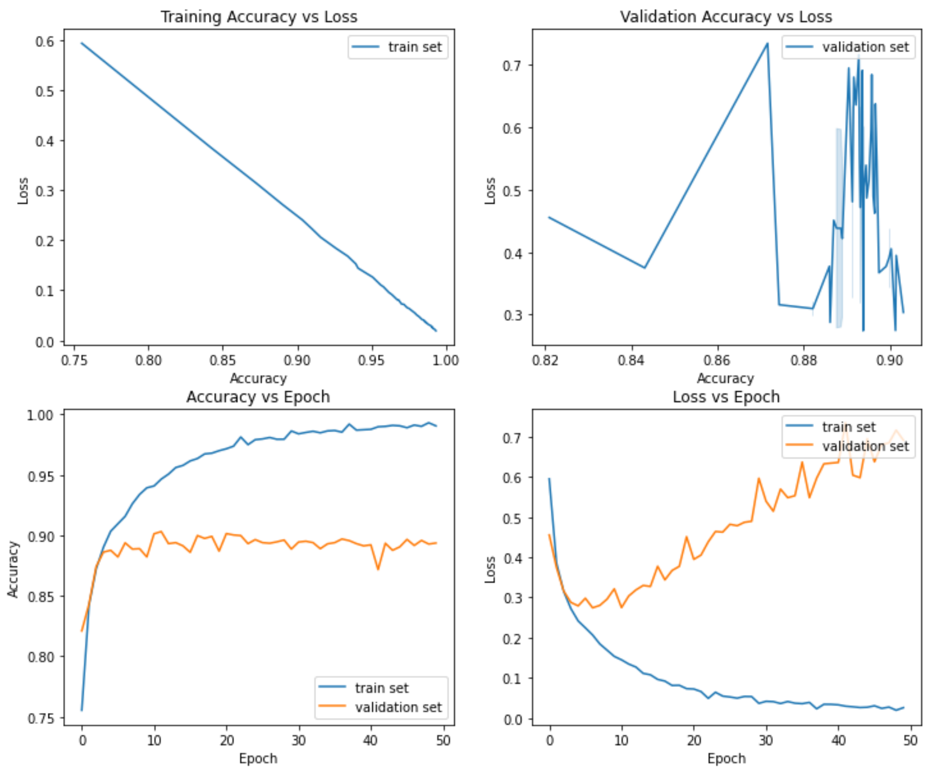
Epochs	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy	Training Time
10	0.8948	0.8719	0.2996	0.3580	0.8802	1.00 minute
30	0.9477	0.9163	0.1679	0.2522	0.9188	2.89 minute
50	**0.9674**	**0.9210**	**0.1309**	**0.2542**	**0.9285**	**4.53 minute**
70	0.9777	0.9255	0.1089	0.2628	0.9277	6.71 minute

Table 5: Model 4 Epoch Experimentation (Batch Size -128, Optimizer- Adam)

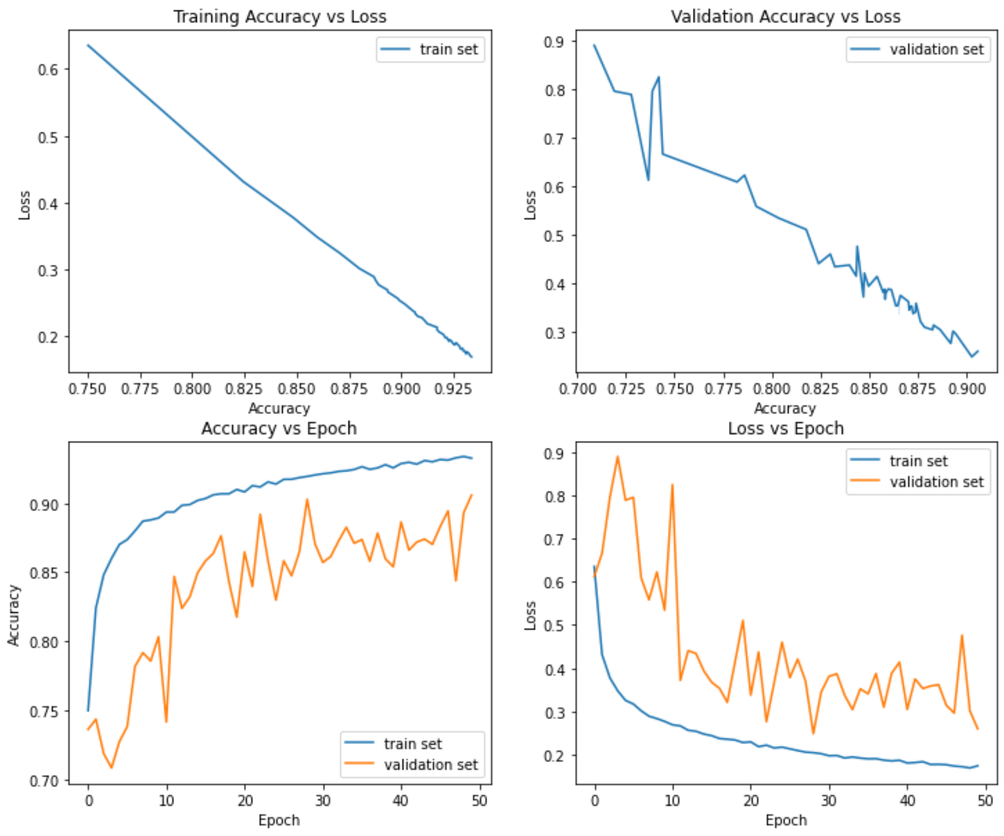
The number of epochs is the number of complete passes through the training dataset. From Table 5, we notice that the accuracy of the model increases with more epochs. For 50 and 70 epochs, the model has similar accuracy but different training time. The training time increases as the number of epochs increase.

Accuracy – Loss plots

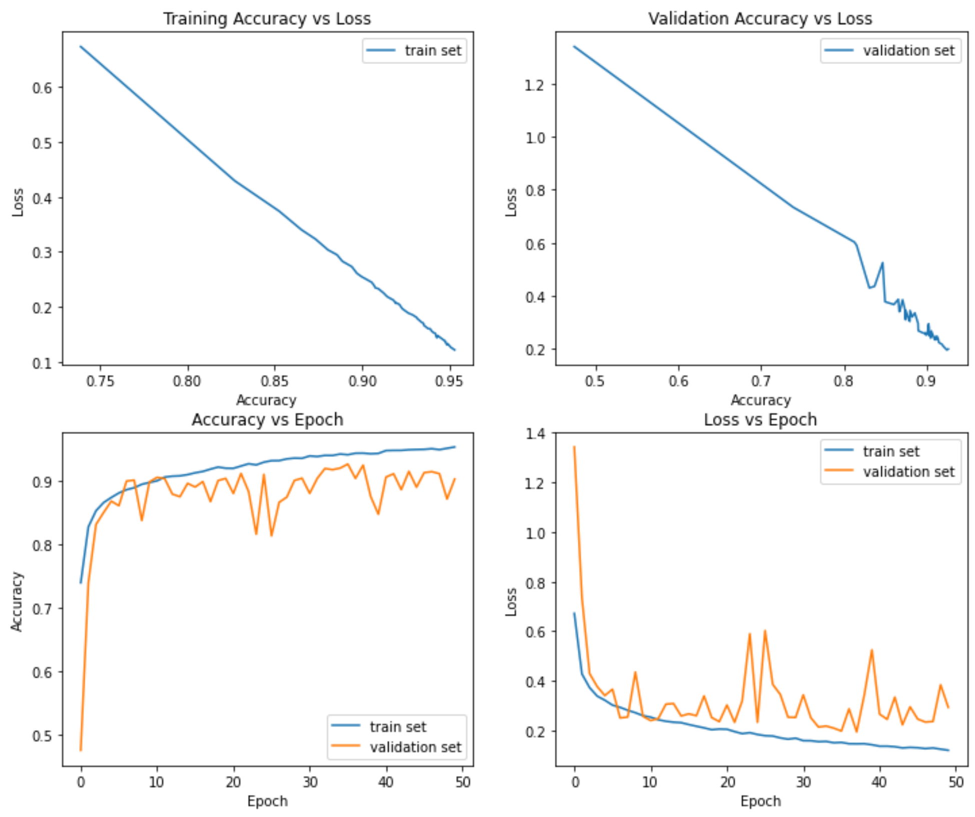
Model 1:



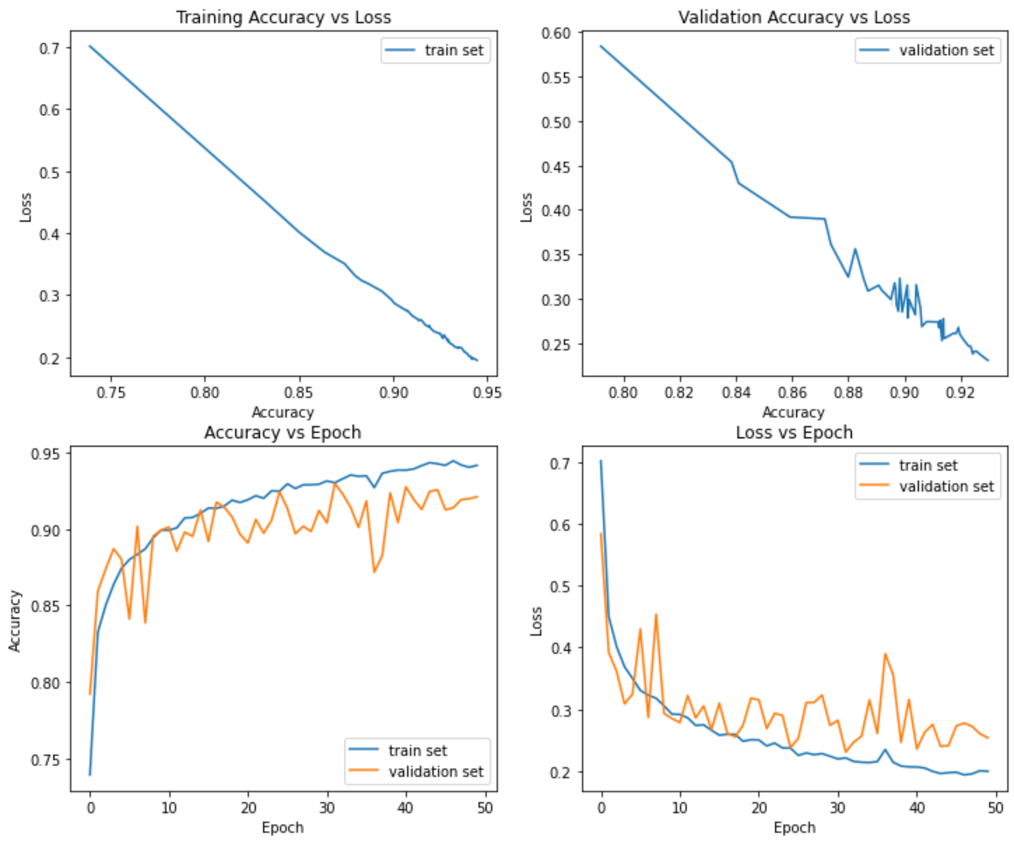
Model 2:



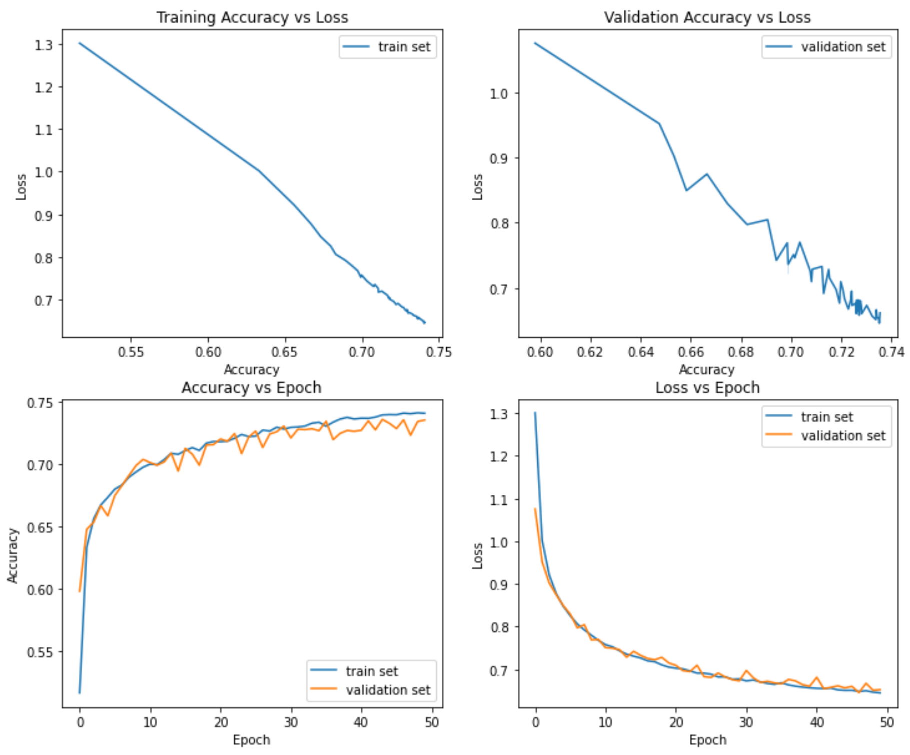
Model 3:



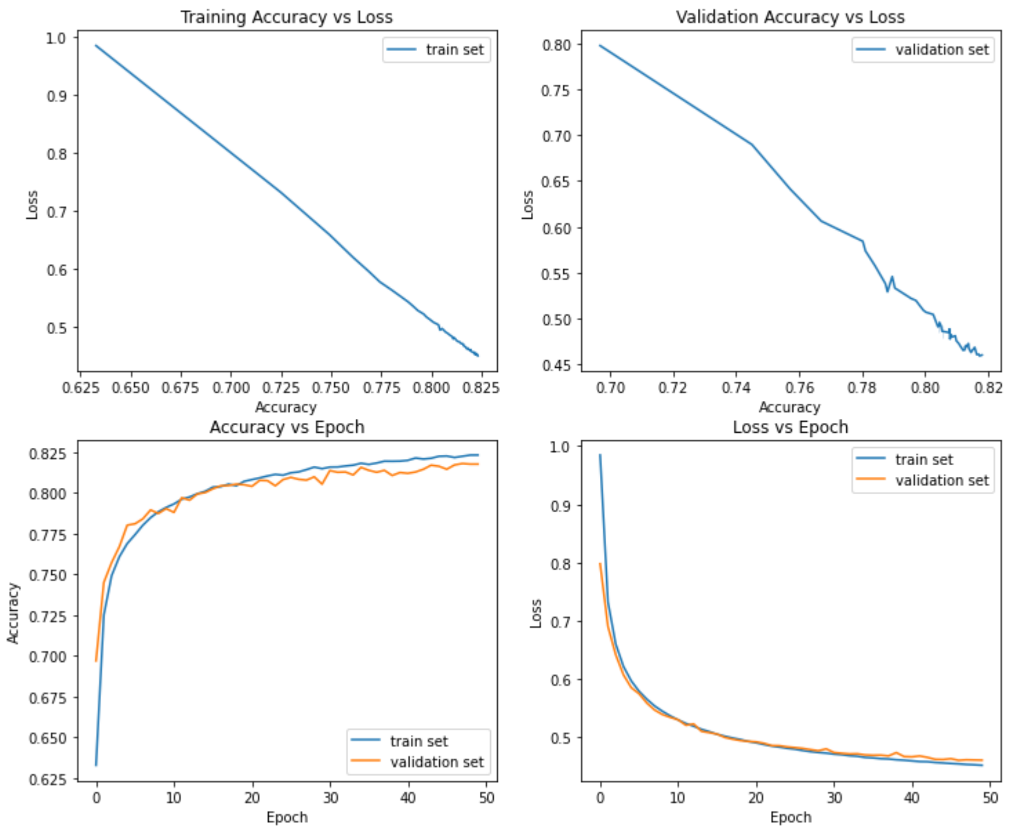
Model 4:



Resnet 50:



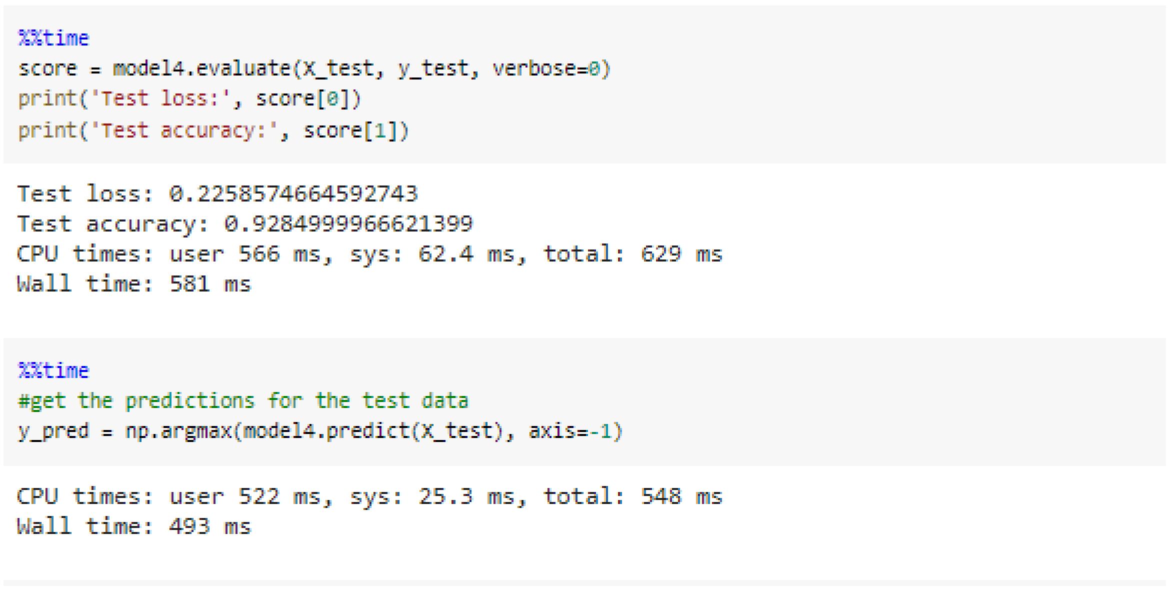
VGG 16:



From the Accuracy – Loss plots, we observe:

- For all the Models, the Loss decreases as the accuracy increases for both the training and validation dataset.
- Also for all the Models, the Accuracy increases with increase in Epochs, and the Loss decreases with increase in epochs for both the training and test dataset.
- Model 1 has a high accuracy but is clearly overfitting. Whereas Model 2 has reduced overfitting, but the accuracy has also reduced.
- Model 3 has good accuracy, but is still slightly overfitting, when compared to Model 4 which has similarly good accuracy, and reduced overfitting. The 'generalization gap' is low.
- The Resnet50 and the VGG16 model do not overfit and show a smooth curve, but their training accuracy is comparatively lower than the CNN sequential models.
- The fluctuations in the Validation Accuracy and Loss plots can be attributed to small size of the validation dataset.

Performance of the Model on the test set



```
%%time
score = model4.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.2258574664592743
Test accuracy: 0.9284999966621399
CPU times: user 566 ms, sys: 62.4 ms, total: 629 ms
Wall time: 581 ms
```

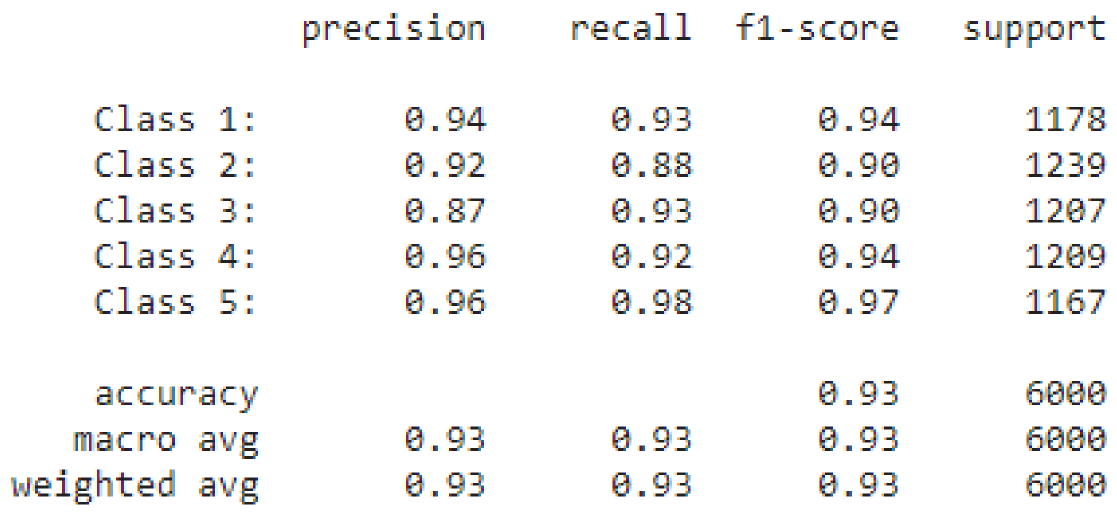
```
%%time
#get the predictions for the test data
y_pred = np.argmax(model4.predict(X_test), axis=-1)
```

```
CPU times: user 522 ms, sys: 25.3 ms, total: 548 ms
Wall time: 493 ms
```

Evaluating the performance of the best Model 4 on the Test dataset (10% of whole dataset), we get an accuracy of 92%, which is good for a classifier. The model is able to generalize i.e. classify unseen data well

In addition to classification accuracy, the metrics that are commonly required for a neural network model on a classification problem are:

- Precision
- Recall
- F1 Score



```
              precision    recall  f1-score   support

    Class 1:       0.94      0.93      0.94      1178
    Class 2:       0.92      0.88      0.90      1239
    Class 3:       0.87      0.93      0.90      1207
    Class 4:       0.96      0.92      0.94      1209
    Class 5:       0.96      0.98      0.97      1167

    accuracy                           0.93      6000
   macro avg       0.93      0.93      0.93      6000
weighted avg       0.93      0.93      0.93      6000
```

From the above model classification report, we observe:

- Model has the highest precision for Class 5 & 4, and the lowest for Class 3. Precision tells how precise/accurate the model is, out of the predicted positive, how many are actual positive. We see the Model is able to correctly identify most of the images of class 4 and 5, and misclassifies many Class 3 images.
- Model has highest recall for Class 5, and the lowest for Class 2. Recall tells how many of the Actual Positives the model capture through classifying it as Positive (True Positive).
- Model has the highest f1-score for Class 5 and the lowest for Class 2 & 3. F1-score seek a balance between Precision and Recall. We see the model has better balance between precision-recall for Class 5 than all other classes.

References:

https://www.kaggle.com/gpreda/cnn-with-tensorflow-keras-for-fashion-mnist

https://www.pyimagesearch.com/2019/02/11/fashion-mnist-with-keras-and-deep-learning/

https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-fashion-mnist-clothing-classification/

https://keras.io/guides/transfer_learning/

https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9

https://machinelearningmastery.com/how-to-calculate-precision-recall-f1-and-more-for-deep-learning-models/

https://www.tensorflow.org/api_docs/python/tf/keras

https://datascience.stackexchange.com/questions/12851/how-do-you-visualize-neural-network-architectures

https://alexlenail.me/NN-SVG/LeNet.html

https://machinelearningmastery.com/how-to-use-transfer-learning-when-developing-convolutional-neural-network-models/

https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/