CM4

April 25, 2021

1 [CM4] MNIST Dataset

1.0.1 Importing Libraries

[1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, Input, BatchNormalization,␣
 ↪Activation, MaxPooling2D
from keras.utils import to_categorical
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report
from keras.models import Model
from keras.regularizers import l2
from keras.optimizers import Adam
import torch
from keras.utils.vis_utils import plot_model
from keras.callbacks import EarlyStopping

import warnings
warnings.filterwarnings("ignore")

## SET ALL SEED
import os
os.environ['PYTHONHASHSEED']=str(0)
import random
random.seed(0)
np.random.seed(0)
tf.random.set_seed(0)
```

1.0.2 Loading dataset

[2]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

[3]:
```
data_mnist = np.load('/content/gdrive/My Drive/Colab Notebooks/Mnist/
 ↪fashion_mnist_dataset_train.npy', allow_pickle=True)
```

[4]:
```
my_dict = data_mnist[()]
features = my_dict.get('features')
target = my_dict.get('target')
target = target -1
```

Here, we have decreased all target values by 1 for further calculations. However, we will be predicting correct values by adding 1 at the end.

[5]:
```
np.unique(target)
```

[5]:
```
array([0., 1., 2., 3., 4.])
```

1.1 Pre-processing Steps

1.1.1 Splitting into train, test and validation set

[6]:
```
x_train, x_test1, y_train, y_test1 = train_test_split(features, target,␣
 ↪test_size=0.10, random_state=0)
x_test, x_val, y_test, y_val = train_test_split(x_test1, y_test1, test_size=0.
 ↪50, random_state=0)
```

[7]:
```
print(x_train.shape)
print(x_test.shape)
print(x_val.shape)
print(y_val.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(54000, 28, 28)
(3000, 28, 28)
(3000, 28, 28)
(3000,)
(54000,)
(3000,)
```

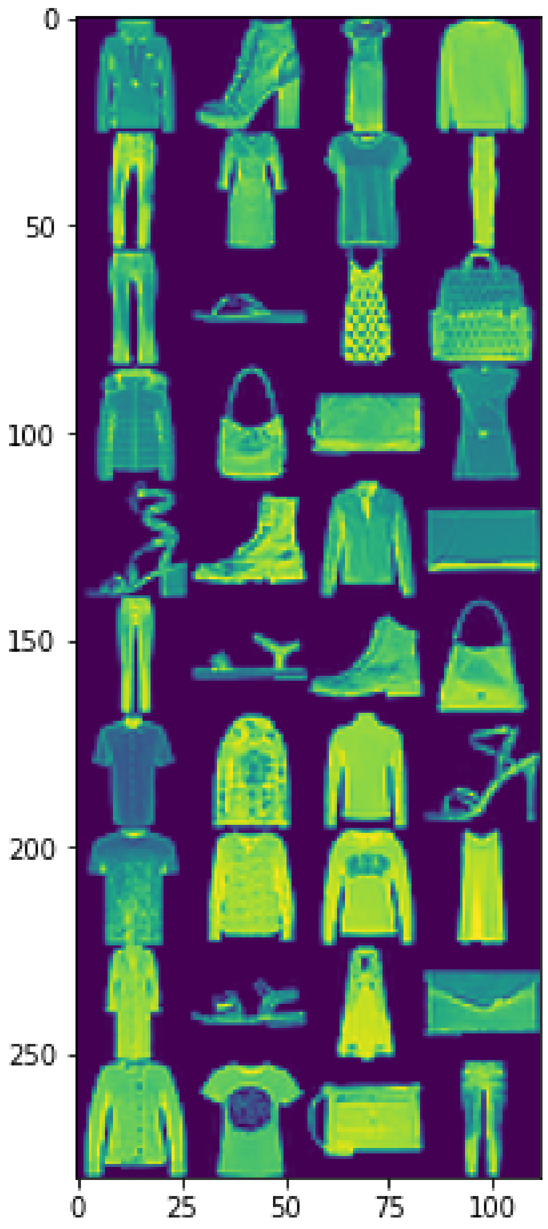
So, there are 60,000 Samples, from which we have taken 90% samples as Training data (54,000) and further divided 6000 samples in Validation and Test Samples (3000 each). Each example is a 28x28 grayscale image, associated with a label from 5 classes. - Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. - This pixel-value is an integer between 0 and 1, inclusive.

1.1.2 Visualization of Mnist data

[8]:
```
images = x_train[0:40]

visual_array = np.zeros([10*28, 4*28])
for col in range(10):
    for row in range(4):
        index = col + row*10
        visual_array[col*28:(col+1)*28, row*28:(row+1)*28] = images[index]
```

[9]:
```
fig= plt.figure(figsize=(20,8))
plt.imshow(visual_array)
plt.show()
```



1.1.3 Input/Output shape of data

- You always have to feed a 4D array of shape (batch_size, height, width, depth) to the CNN.
- Output data from CNN is also a 4D array of shape (batch_size, height, width, depth).

In below steps, we have changed shape of training, testing and validation inputs for later uses.

[10]:
```
  x_train_new = np.expand_dims(x_train, -1)
x_test_new = np.expand_dims(x_test, -1)
```

```
x_val_new = np.expand_dims(x_val, -1)

y_train_new = to_categorical(y_train)
y_test_new = to_categorical(y_test)
y_val_new = to_categorical(y_val)
```

[11]:
```
print(x_train_new.shape)
print(x_test_new.shape)
print(x_val_new.shape)
print(y_val_new.shape)
print(y_train_new.shape)
print(y_test_new.shape)
```

```
(54000, 28, 28, 1)
(3000, 28, 28, 1)
(3000, 28, 28, 1)
(3000, 5)
(54000, 5)
(3000, 5)
```

- Inputs expanded with 1 dimension and outputs have been converted to categorical forms.

2 Models

2.1 1. CNN Model

- The model has 3 hidden layer, where the number of neurons in first hidden layer is CNN and has 32 neurons, next layer has 64 neurons. The output layer has 3 nodes as there are 3 classes.
- Activation functions play a key role in neural networks, so it is essential to understand the advantages and disadvantages to achieve better performance.
- The activation function used for hiddens layer is **sigmoid** because it can be used as a good classifier. Another advantage of this function is that it produces a value in the range of (0,1) when encountered with (- infinite, + infinite) as in the linear function. So the activation value does not vanish.
- The number of channels of the input matrix and the number of channels in each filter must match in order to be able to perform element-wise multiplication.
- So the main difference between first and second convolutions is that the # of channels in the input matrix in the first convolution is 3 so we will use 32 filters where each filter has 3 channels (depth of kernel matrix).
- For the second convolution, the input matrix has 32 channels (feature maps), so each filter for this convolution must have 32 channels as well. For example: each of the 64 filters will have the 32(3x3 shape).
- The result of a convolution step for a single filter of 32(3x3) shape will be a single channel of WxH (Width, Height) shape. After applying all 64 filters (where each of them has shape:

32@3x3) we will get 64 channels, where each channel is a result of the convolution of a single filter.

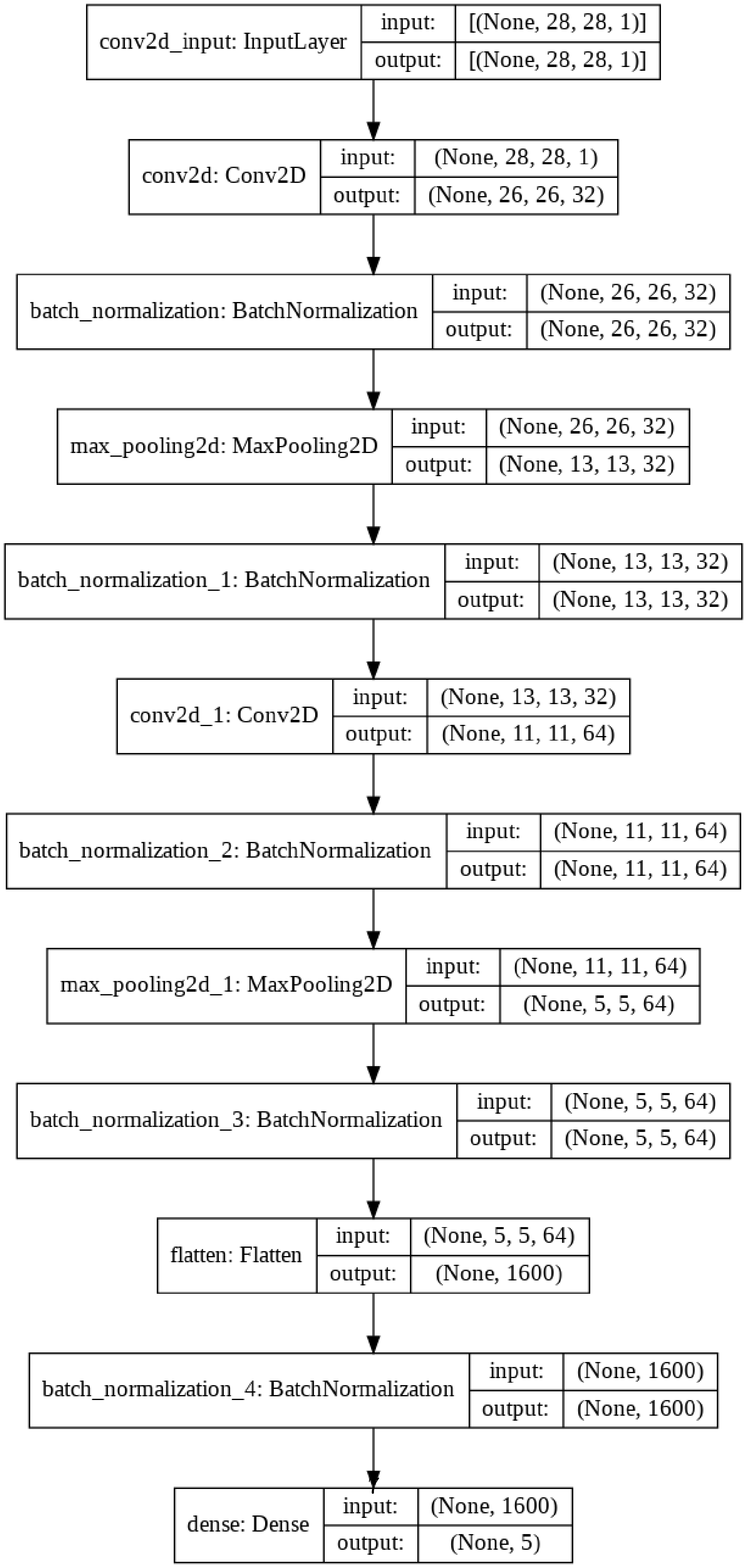
- The last layer has **softmax** activation function becasue it is used to normalize the outputs, converting them from weighted sum values into probabilities that sum to one. Each value in the output of the softmax function is interpreted as the probability of membership for each class.
- Apart from this, we have implemented max_pooling and batch_normalization for more accurate results.

[12]:
```
model_cnn5 = Sequential()

model_cnn5.add(Conv2D(32, kernel_size=3, activation='sigmoid',␣
 ↪input_shape=(28,28,1)))
model_cnn5.add(BatchNormalization())
model_cnn5.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn5.add(BatchNormalization())
model_cnn5.add(Conv2D(64, kernel_size=3, activation='sigmoid'))
model_cnn5.add(BatchNormalization())
model_cnn5.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn5.add(BatchNormalization())
model_cnn5.add(Flatten())
model_cnn5.add(BatchNormalization())
model_cnn5.add(Dense(5, activation='softmax'))
```

[13]:
```
plot_model(model_cnn5, show_shapes=True, show_layer_names=True)
```

[13]:


conv2d_input: InputLayer
input: [(None, 28, 28, 1)]
output: [(None, 28, 28, 1)]
conv2d: Conv2D
input: (None, 28, 28, 1)
output: (None, 26, 26, 32)
batch_normalization: BatchNormalization
input: (None, 26, 26, 32)
output: (None, 26, 26, 32)
max_pooling2d: MaxPooling2D
input: (None, 26, 26, 32)
output: (None, 13, 13, 32)
batch_normalization_1: BatchNormalization
input: (None, 13, 13, 32)
output: (None, 13, 13, 32)
conv2d_1: Conv2D
input: (None, 13, 13, 32)
output: (None, 11, 11, 64)
batch_normalization_2: BatchNormalization
input: (None, 11, 11, 64)
output: (None, 11, 11, 64)
max_pooling2d_1: MaxPooling2D
input: (None, 11, 11, 64)
output: (None, 5, 5, 64)
batch_normalization_3: BatchNormalization
input: (None, 5, 5, 64)
output: (None, 5, 5, 64)
flatten: Flatten
input: (None, 5, 5, 64)
output: (None, 1600)
batch_normalization_4: BatchNormalization
input: (None, 1600)
output: (None, 1600)
dense: Dense
input: (None, 1600)
output: (None, 5)

```
[14]: es5 = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=10)
```

- We used batch_size 128, as it is an idea size for this much training data.
- We have implemented best optimizer-activation function pair.
- No of epochs used with early stopping are 25.
- Loss function used here is, categorical cross-entropy.

```
[15]: batch_size = 128
      no_epochs = 25
      verbosity = 1

      model_cnn5.compile(optimizer='sgd', loss='categorical_crossentropy',␣
       ↪metrics=['accuracy'])
      History_cnn5 = model_cnn5.fit(x_train_new, y_train_new,␣
       ↪validation_data=(x_val_new, y_val_new), batch_size=batch_size,␣
       ↪epochs=no_epochs, verbose=verbosity, callbacks=[es5])
```

```
Epoch 1/25
422/422 [==============================] - 36s 7ms/step - loss: 0.7833 -
accuracy: 0.6885 - val_loss: 1.2141 - val_accuracy: 0.3957
Epoch 2/25
422/422 [==============================] - 2s 6ms/step - loss: 0.5565 -
accuracy: 0.7814 - val_loss: 0.6429 - val_accuracy: 0.7170
Epoch 3/25
422/422 [==============================] - 2s 6ms/step - loss: 0.5067 -
accuracy: 0.8025 - val_loss: 0.4977 - val_accuracy: 0.8010
Epoch 4/25
422/422 [==============================] - 2s 6ms/step - loss: 0.4778 -
accuracy: 0.8144 - val_loss: 0.4606 - val_accuracy: 0.8117
Epoch 5/25
422/422 [==============================] - 2s 6ms/step - loss: 0.4535 -
accuracy: 0.8221 - val_loss: 0.4581 - val_accuracy: 0.8157
Epoch 6/25
422/422 [==============================] - 2s 6ms/step - loss: 0.4352 -
accuracy: 0.8299 - val_loss: 0.4530 - val_accuracy: 0.8117
Epoch 7/25
422/422 [==============================] - 2s 6ms/step - loss: 0.4196 -
accuracy: 0.8398 - val_loss: 0.4271 - val_accuracy: 0.8250
Epoch 8/25
422/422 [==============================] - 2s 6ms/step - loss: 0.4041 -
accuracy: 0.8424 - val_loss: 0.4268 - val_accuracy: 0.8290
Epoch 9/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3980 -
accuracy: 0.8445 - val_loss: 0.4269 - val_accuracy: 0.8263
Epoch 10/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3862 -
```

```
accuracy: 0.8510 - val_loss: 0.4107 - val_accuracy: 0.8340
Epoch 11/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3782 -
accuracy: 0.8536 - val_loss: 0.4273 - val_accuracy: 0.8250
Epoch 12/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3719 -
accuracy: 0.8554 - val_loss: 0.4241 - val_accuracy: 0.8290
Epoch 13/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3587 -
accuracy: 0.8637 - val_loss: 0.3834 - val_accuracy: 0.8430
Epoch 14/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3558 -
accuracy: 0.8629 - val_loss: 0.3784 - val_accuracy: 0.8477
Epoch 15/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3491 -
accuracy: 0.8644 - val_loss: 0.3858 - val_accuracy: 0.8477
Epoch 16/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3469 -
accuracy: 0.8653 - val_loss: 0.4020 - val_accuracy: 0.8433
Epoch 17/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3312 -
accuracy: 0.8705 - val_loss: 0.3933 - val_accuracy: 0.8357
Epoch 18/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3305 -
accuracy: 0.8717 - val_loss: 0.3821 - val_accuracy: 0.8463
Epoch 19/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3259 -
accuracy: 0.8730 - val_loss: 0.3698 - val_accuracy: 0.8507
Epoch 20/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3234 -
accuracy: 0.8750 - val_loss: 0.3601 - val_accuracy: 0.8577
Epoch 21/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3202 -
accuracy: 0.8759 - val_loss: 0.3778 - val_accuracy: 0.8500
Epoch 22/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3122 -
accuracy: 0.8804 - val_loss: 0.3680 - val_accuracy: 0.8550
Epoch 23/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3041 -
accuracy: 0.8815 - val_loss: 0.3597 - val_accuracy: 0.8543
Epoch 24/25
422/422 [==============================] - 2s 6ms/step - loss: 0.3081 -
accuracy: 0.8803 - val_loss: 0.3740 - val_accuracy: 0.8467
Epoch 25/25
422/422 [==============================] - 2s 6ms/step - loss: 0.2965 -
accuracy: 0.8864 - val_loss: 0.3564 - val_accuracy: 0.8513
```

2.1.1 Accuracy on test set with best hyper-parameters

[16]:
```
y_pred_cnn5 = model_cnn5.predict_classes(x_test_new, verbose=0)
y_pred_cnn5 = to_categorical(y_pred_cnn5)
accuracy_cnn5 = accuracy_score(y_test_new, y_pred_cnn5)
accuracy_cnn5
```

[16]:
```
0.8513333333333334
```

- As we see from the above results, we got 88.64 accuracy on training data, and 85.13 on validation set and finally 85.13 accuracy on test/unknown data.

2.2 2. Resnet Model

- ResNet, short for **Residual Networks** is a classic neural network used as a backbone for many computer vision tasks.
- The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+ layers successfully. Prior to ResNet training very deep neural networks was difficult due to the problem of vanishing gradients.
- However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem — as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient extremely small.
- As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly.
- We can see this in an example explained below :

[17]:
```
def resnet_v1(input_shape, depth, num_classes = 5):
    if (depth - 2) % 6 != 0:
        raise ValueError('depth should be 6n + 2 (eg 20, 32, 44 in [a])') #18␣
↪layers

    num_filters = 16
    num_res_blocks = int((depth - 2) / 6) #3 blocks

    inputs = Input(shape = input_shape)
    x = resnet_layer(inputs = inputs)

    for stack in range(3):
        for res_block in range(num_res_blocks):
            strides = 1
            if stack > 0 and res_block == 0:
                strides = 2
            y = resnet_layer(inputs = x, num_filters = num_filters, strides =␣
↪strides)
            y = resnet_layer(inputs = y, num_filters = num_filters, activation␣
↪= None)
            if stack > 0 and res_block == 0:
                x = resnet_layer(inputs = x, num_filters = num_filters,␣
↪kernel_size = 1, strides = strides, activation = None, batch_normalization =␣
↪False)
            x = keras.layers.add([x, y])
            x = Activation('relu')(x)
        num_filters *= 2

    y = Flatten()(x)
    outputs = Dense(num_classes, activation ='softmax', kernel_initializer␣
↪='he_normal')(y)

    model = Model(inputs = inputs, outputs = outputs)
    return model
```

[18]:

```
# Basic ResNet Building Block
conv_first = False
def resnet_layer(inputs,
                                num_filters = 16,
                                kernel_size = 3,
                                strides = 1,
                                activation ='relu',
                                batch_normalization = True):

        conv = Conv2D(num_filters,
                                kernel_size = kernel_size,
                                strides = strides,
                                padding ='same',
                                kernel_initializer ='he_normal',
                                kernel_regularizer = l2(1e-4))

        x = inputs
        if conv_first:
                x = conv(x)
                if batch_normalization:
                        x = BatchNormalization()(x)
                if activation is not None:
                        x = Activation(activation)(x)
        else:
                if batch_normalization:
                        x = BatchNormalization()(x)
                if activation is not None:
                        x = Activation(activation)(x)
                x = conv(x)
        return x
```

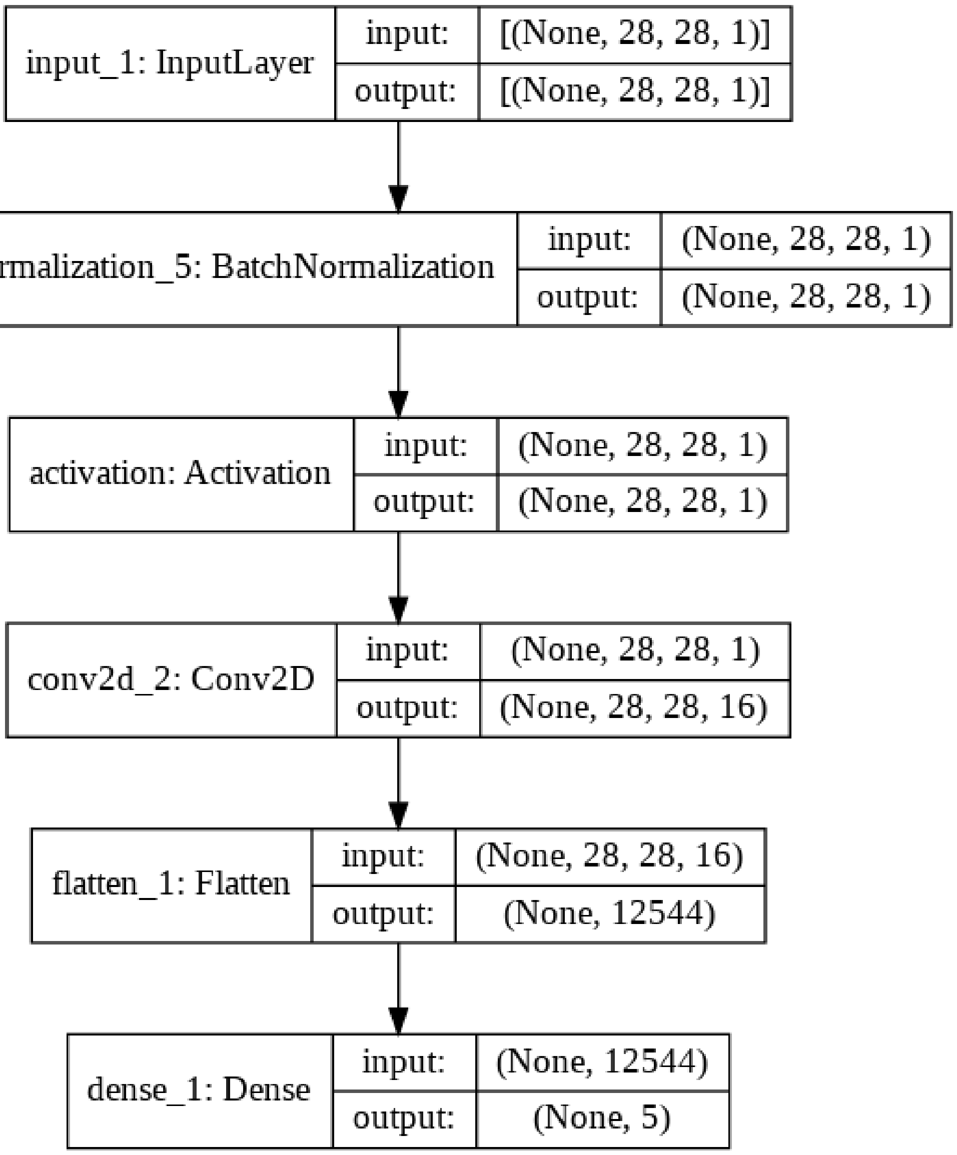
[19]:

```
def lr_schedule(epoch):
    lr = 1e-3
    if epoch > 180:
        lr *= 0.5e-3
    elif epoch > 160:
        lr *= 1e-3
    elif epoch > 120:
        lr *= 1e-2
    elif epoch > 80:
        lr *= 1e-1
    print('Learning rate: ', lr)
    return lr
```

[20]:
```
model_resnet = resnet_v1(input_shape = (28,28,1), depth = 2)
```

[21]:
```
plot_model(model_resnet, show_shapes=True, show_layer_names=True)
```

[21]:



From the above plot, we can observe following points :

- Model has 2 layers, our input size changed from (28,28,1) to (28,28,16).
- In the next step, we flattened our data from 3-dimension to 1-dimension, and lastly reduced to 5 (ouput size).

- We used batch_size 128, as it is an ideal size for this much training data.
- We have implemented adam optimizer- relu activation function pair.
- No of epochs used with early stopping are 25.
- Loss function used here is, categorical cross-entropy.
- Learning rate used is 1e-3.

[22]:
```
batch_size = 128
no_epochs = 25
verbosity = 1
model_resnet.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
History_resnet = model_resnet.fit(x_train_new, y_train_new,␣
 ↪validation_data=(x_val_new, y_val_new), epochs=no_epochs, verbose=verbosity,␣
 ↪batch_size=batch_size)
```

```
Epoch 1/25
422/422 [==============================] - 2s 4ms/step - loss: 1.0588 -
accuracy: 0.6368 - val_loss: 0.7300 - val_accuracy: 0.7023
Epoch 2/25
422/422 [==============================] - 1s 3ms/step - loss: 0.7247 -
accuracy: 0.7124 - val_loss: 0.7039 - val_accuracy: 0.7193
Epoch 3/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6880 -
accuracy: 0.7258 - val_loss: 0.7241 - val_accuracy: 0.7037
Epoch 4/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6794 -
accuracy: 0.7261 - val_loss: 0.6612 - val_accuracy: 0.7333
Epoch 5/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6569 -
accuracy: 0.7360 - val_loss: 0.6522 - val_accuracy: 0.7267
Epoch 6/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6319 -
accuracy: 0.7461 - val_loss: 0.6261 - val_accuracy: 0.7447
Epoch 7/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6208 -
accuracy: 0.7500 - val_loss: 0.6278 - val_accuracy: 0.7510
Epoch 8/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6131 -
accuracy: 0.7543 - val_loss: 0.6221 - val_accuracy: 0.7583
Epoch 9/25
```

```
422/422 [==============================] - 1s 3ms/step - loss: 0.6160 -
accuracy: 0.7549 - val_loss: 0.6245 - val_accuracy: 0.7490
Epoch 10/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6123 -
accuracy: 0.7548 - val_loss: 0.6238 - val_accuracy: 0.7487
Epoch 11/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6100 -
accuracy: 0.7583 - val_loss: 0.6366 - val_accuracy: 0.7467
Epoch 12/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6079 -
accuracy: 0.7545 - val_loss: 0.6215 - val_accuracy: 0.7570
Epoch 13/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6047 -
accuracy: 0.7551 - val_loss: 0.6181 - val_accuracy: 0.7487
Epoch 14/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6067 -
accuracy: 0.7567 - val_loss: 0.6203 - val_accuracy: 0.7600
Epoch 15/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6083 -
accuracy: 0.7575 - val_loss: 0.6259 - val_accuracy: 0.7460
Epoch 16/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6074 -
accuracy: 0.7575 - val_loss: 0.6228 - val_accuracy: 0.7497
Epoch 17/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6001 -
accuracy: 0.7591 - val_loss: 0.6189 - val_accuracy: 0.7560
Epoch 18/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6015 -
accuracy: 0.7607 - val_loss: 0.6259 - val_accuracy: 0.7457
Epoch 19/25
422/422 [==============================] - 2s 4ms/step - loss: 0.5971 -
accuracy: 0.7613 - val_loss: 0.6251 - val_accuracy: 0.7510
Epoch 20/25
422/422 [==============================] - 1s 4ms/step - loss: 0.6026 -
accuracy: 0.7572 - val_loss: 0.6208 - val_accuracy: 0.7497
Epoch 21/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6015 -
accuracy: 0.7589 - val_loss: 0.6177 - val_accuracy: 0.7523
Epoch 22/25
422/422 [==============================] - 1s 3ms/step - loss: 0.5975 -
accuracy: 0.7611 - val_loss: 0.6349 - val_accuracy: 0.7483
Epoch 23/25
422/422 [==============================] - 1s 3ms/step - loss: 0.5975 -
accuracy: 0.7608 - val_loss: 0.6218 - val_accuracy: 0.7460
Epoch 24/25
422/422 [==============================] - 1s 3ms/step - loss: 0.6007 -
accuracy: 0.7606 - val_loss: 0.6244 - val_accuracy: 0.7530
Epoch 25/25
```

```
422/422 [==============================] - 1s 3ms/step - loss: 0.5955 -
accuracy: 0.7601 - val_loss: 0.6177 - val_accuracy: 0.7463
```

[23]:
```
y_pred_resnet = model_resnet.predict(x_test_new, verbose=0)
y_pred_resnet = np.argmax(y_pred_resnet,axis=1)
y_pred_resnet = to_categorical(y_pred_resnet)
accuracy_resnet = accuracy_score(y_test_new, y_pred_resnet)
accuracy_resnet
```

[23]:
```
0.7346666666666667
```

On training set, model got 76.01 accuracy, whereas on validation and test/unknown data it got 74.63 and 73.47 accuracy score respectively.

2.3 Can the addition of the Resnet architecture learn a better classifier than a simple CNN?

Its a generally accepted principle that deeper networks are capable of learning more complex functions and representations of the input which should lead to better performance. However, many researchers observed that adding more layers eventually had a negative effect on the final performance.

This phenomenon is referred as the **degradation problem** - alluding to the fact that although better parameter initialization techniques and batch normalization allow for deeper networks to converge, they often converge at a higher error rate than their shallower counterparts. In the limit, simply stacking more layers degrades the model's ultimate performance.

Loss for CNN model is **0.29**
Loss for Resnet architecture is **0.59**

We conclude that, in this case Resnet architecture is not a better fit than simple CNN.

2.4 References

- https://www.kaggle.com/jagdish2386/fashion-mnist-dnn
- https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7
- https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b
- https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-706ac4284c8a
- https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33