CM6

April 26, 2021

1 [CM6]

[3]:
```
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 time

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)
```

1.0.1 Loading dataset

[4]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

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

[5]:
```
data_mnist = np.load('/content/gdrive/My Drive/Colab Notebooks/Mnist/
↪fashion_mnist_dataset_train.npy', allow_pickle=True)
```

So, there are 60,000 Training Samples and 10,000 Test Samples.
Each example is a 28x28 grayscale image, associated with a label from 6 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 255, inclusive.

The first column of the Training Samples consists of Class Labels and represents the article of Clothing.

[6]:
```
my_dict = data_mnist[()]
features = my_dict.get('features')
target = my_dict.get('target')
target = target -1
```

[7]:
```
np.unique(target)
```

[7]:
```
array([0., 1., 2., 3., 4.])
```

1.0.2 Splitting into train, test and validation set

[8]:
```
x_train, x_test1, y_train, y_test1 = train_test_split(features, target,␣
↪test_size=0.20, 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)
```

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

[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)
```

```
(48000, 28, 28, 1)
(6000, 28, 28, 1)
(6000, 28, 28, 1)
(6000, 5)
(48000, 5)
(6000, 5)
```

2 Models

Plots used to observe performance of the models

Optimization Learning Curves: Learning curves calculated on the metric by which the parameters of the model are being optimized. We will use loss vs epoch curve for this purpose.

Performance Learning Curves: Learning curves calculated on the metric by which the model will be evaluated and selected. We will use accuracy vs epoch curve for this purpose.

2.1 1. CNN Model

2.1.1 Hyper-parameters of CNN

1. Kernel/Filter Size : - A filter is a matrix of weights with which we convolve on the input. - The filter on convolution, provides a measure for how close a patch of input resembles a feature. - Smaller filters collect as much local information as possible, bigger filters represent more global, high-level and representative information. - If you think that a big amount of pixels are necessary for the network to recognize the object you will use large filters **(11x11 or 9x9)**. If you think what differentiates objects are some small and local features you should use small filters **(3x3 or 5x5)**. - **In general we use filters with odd sizes.**

2. Number of Layers : - It must be chosen wisely as a very high number may introduce problems like over-fitting and vanishing and exploding gradient problems and a lower number may cause a model to have high bias and low potential model. - Depends a lot on the size of data used for training.

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

4. Activation function : - The activation function is a node that is put at the end of or in between Neural Networks. - They help to decide if the neuron would fire or not. - **The activation function is the non linear transformation that we do over the input signal. This transformed**

output is then sent to the next layer of neurons as input. - The popular choices in this are ReLU, Sigmoid & Tanh(only for shallow networks), and LeakyReLU.

5. Number of Epochs : - The number of epochs is the number of times the entire training data is shown to the model. - It plays an important role in how well does the model fit on the train data. **High number of epochs may over-fit** to the data and may have generalization problems on the test and validation set, also they could cause vanishing and exploding gradient problems. - **Lower number of epochs may limit the potential** of the model.

- To add a Dense layer on top of the CNN layer, we have to change the 4D output of CNN to 2D using a **Flatten layer**.

[12]:
```
start = time.time()
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]:
```
es5 = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=10)
```

[14]:
```
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])
end = time.time()
```

```
Epoch 1/25
375/375 [==============================] - 34s 7ms/step - loss: 0.7996 -
accuracy: 0.6805 - val_loss: 1.5217 - val_accuracy: 0.3240
Epoch 2/25
375/375 [==============================] - 2s 7ms/step - loss: 0.5604 -
accuracy: 0.7769 - val_loss: 0.5731 - val_accuracy: 0.7735
Epoch 3/25
375/375 [==============================] - 2s 7ms/step - loss: 0.5081 -
```

```
accuracy: 0.8027 - val_loss: 0.5007 - val_accuracy: 0.7993
Epoch 4/25
375/375 [==============================] - 2s 7ms/step - loss: 0.4807 -
accuracy: 0.8124 - val_loss: 0.4837 - val_accuracy: 0.8082
Epoch 5/25
375/375 [==============================] - 2s 7ms/step - loss: 0.4580 -
accuracy: 0.8227 - val_loss: 0.4731 - val_accuracy: 0.8033
Epoch 6/25
375/375 [==============================] - 2s 7ms/step - loss: 0.4378 -
accuracy: 0.8295 - val_loss: 0.4602 - val_accuracy: 0.8195
Epoch 7/25
375/375 [==============================] - 2s 7ms/step - loss: 0.4228 -
accuracy: 0.8367 - val_loss: 0.4523 - val_accuracy: 0.8212
Epoch 8/25
375/375 [==============================] - 2s 7ms/step - loss: 0.4126 -
accuracy: 0.8397 - val_loss: 0.4505 - val_accuracy: 0.8263
Epoch 9/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3996 -
accuracy: 0.8442 - val_loss: 0.4350 - val_accuracy: 0.8297
Epoch 10/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3970 -
accuracy: 0.8443 - val_loss: 0.4257 - val_accuracy: 0.8298
Epoch 11/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3874 -
accuracy: 0.8491 - val_loss: 0.4311 - val_accuracy: 0.8247
Epoch 12/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3728 -
accuracy: 0.8541 - val_loss: 0.4221 - val_accuracy: 0.8337
Epoch 13/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3627 -
accuracy: 0.8605 - val_loss: 0.4172 - val_accuracy: 0.8385
Epoch 14/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3605 -
accuracy: 0.8607 - val_loss: 0.4032 - val_accuracy: 0.8408
Epoch 15/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3451 -
accuracy: 0.8685 - val_loss: 0.4205 - val_accuracy: 0.8348
Epoch 16/25
375/375 [==============================] - 3s 7ms/step - loss: 0.3475 -
accuracy: 0.8673 - val_loss: 0.4160 - val_accuracy: 0.8347
Epoch 17/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3437 -
accuracy: 0.8671 - val_loss: 0.3990 - val_accuracy: 0.8433
Epoch 18/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3380 -
accuracy: 0.8701 - val_loss: 0.4049 - val_accuracy: 0.8462
Epoch 19/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3246 -
```

```
accuracy: 0.8745 - val_loss: 0.4094 - val_accuracy: 0.8385
Epoch 20/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3197 -
accuracy: 0.8752 - val_loss: 0.4249 - val_accuracy: 0.8317
Epoch 21/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3213 -
accuracy: 0.8762 - val_loss: 0.3940 - val_accuracy: 0.8517
Epoch 22/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3140 -
accuracy: 0.8763 - val_loss: 0.4018 - val_accuracy: 0.8468
Epoch 23/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3121 -
accuracy: 0.8816 - val_loss: 0.3975 - val_accuracy: 0.8460
Epoch 24/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3036 -
accuracy: 0.8850 - val_loss: 0.3884 - val_accuracy: 0.8532
Epoch 25/25
375/375 [==============================] - 2s 7ms/step - loss: 0.3046 -
accuracy: 0.8825 - val_loss: 0.3809 - val_accuracy: 0.8550
```

[15]:
```
end - start
```

[15]: 98.72615313529968

[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]: 0.8588333333333333

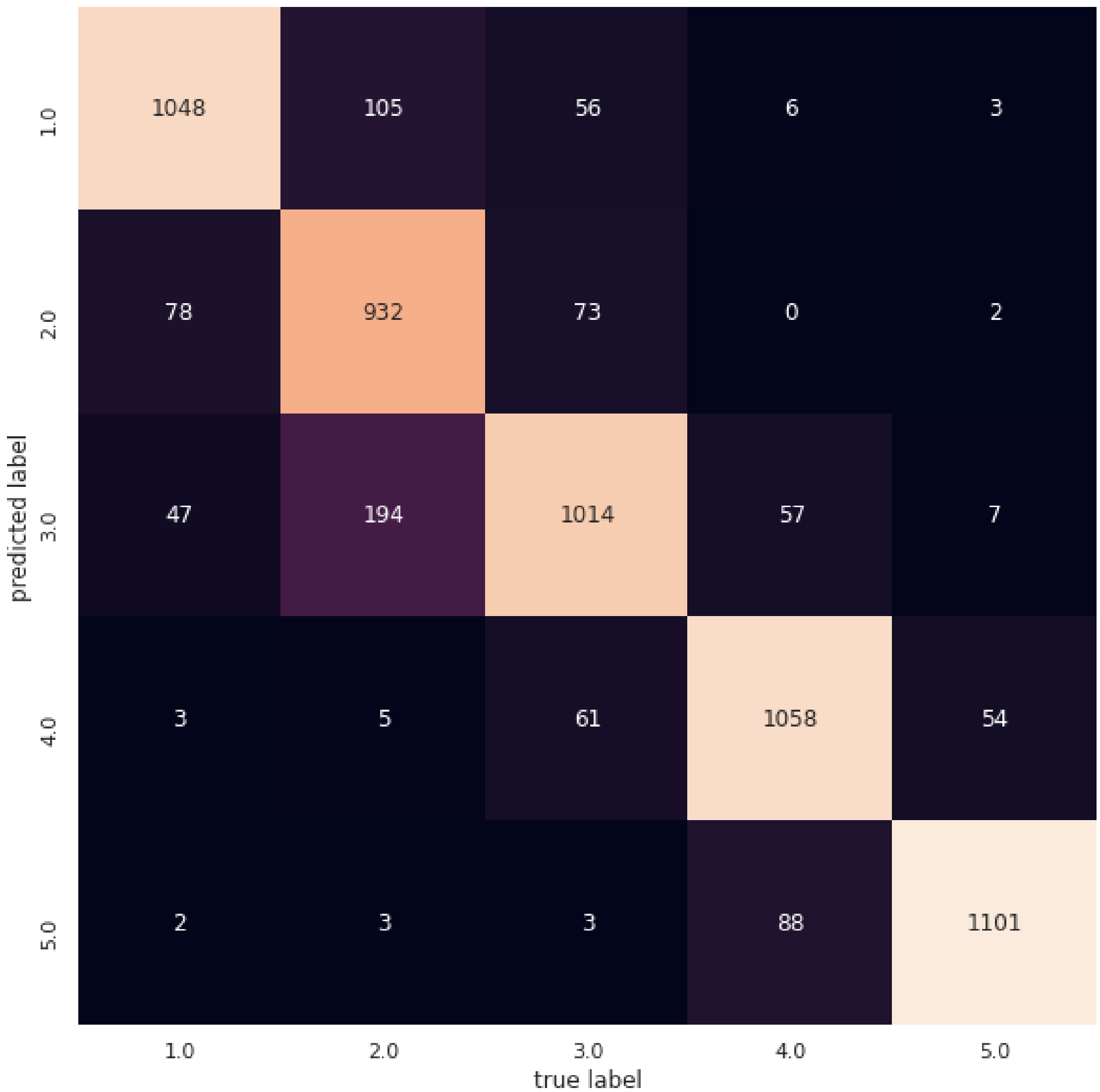
[17]:
```
predict_labels_cnn5 = []
for pred in y_pred_cnn5:
    predict_labels_cnn5.append(np.argmax(pred))
```

2.1.2 Confusion Matrix

The confusion indicates the the CNN succesfully identified most of the labels labels, however there is some confusion to separate labels like 2 & 3 and 4 & 5.

[18]:
```
mat_cnn = confusion_matrix(y_test, predict_labels_cnn5)
plt.figure(figsize=(10, 10))
sns.set()
sns.heatmap(mat_cnn.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=np.unique(y_test+1),
            yticklabels=np.unique(y_test+1))
plt.xlabel('true label')
plt.ylabel('predicted label')
```

```
plt.show()
```



Precision : Precision gives how precisely our model was able to identify labels. The model was aboe to precisly calssify labels 0 and 3 compared to others.

Recall : In an imbalanced classification problem with more than two classes, recall is calculated as the sum of true positives across all classes divided by the sum of true positives and false negatives across all classes. The model predicted correct labels 94% correct labels for class 4 compared to others.

f1-score : The F1 score can be interpreted as a weighted average of the precision and recall values, where an F1 score reaches its best value at 1 and worst value at 0. In our case, it is almost best.

Support : Support shows number of occurrences of each classes in predicted test data.

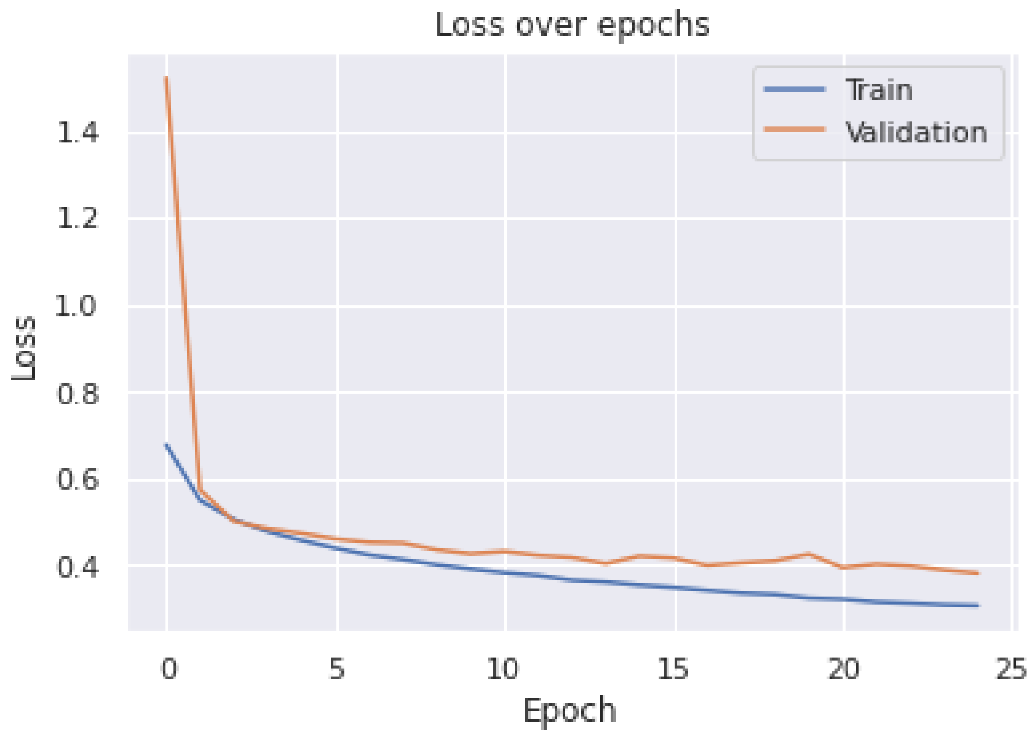
[19]:
```
print(classification_report(predict_labels_cnn5,y_test))
```

```
              precision    recall  f1-score   support

           0       0.89      0.86      0.87      1218
           1       0.75      0.86      0.80      1085
           2       0.84      0.77      0.80      1319
           3       0.88      0.90      0.89      1181
           4       0.94      0.92      0.93      1197

    accuracy                           0.86      6000
   macro avg       0.86      0.86      0.86      6000
weighted avg       0.86      0.86      0.86      6000
```

Optimization Learning Curve

We have tried various combination of parameters, and the above results preresentated here are the best obtained by the combination of parameters we tried. The curve shows that the CNN is good fit of for the data and the data has good learning rate.

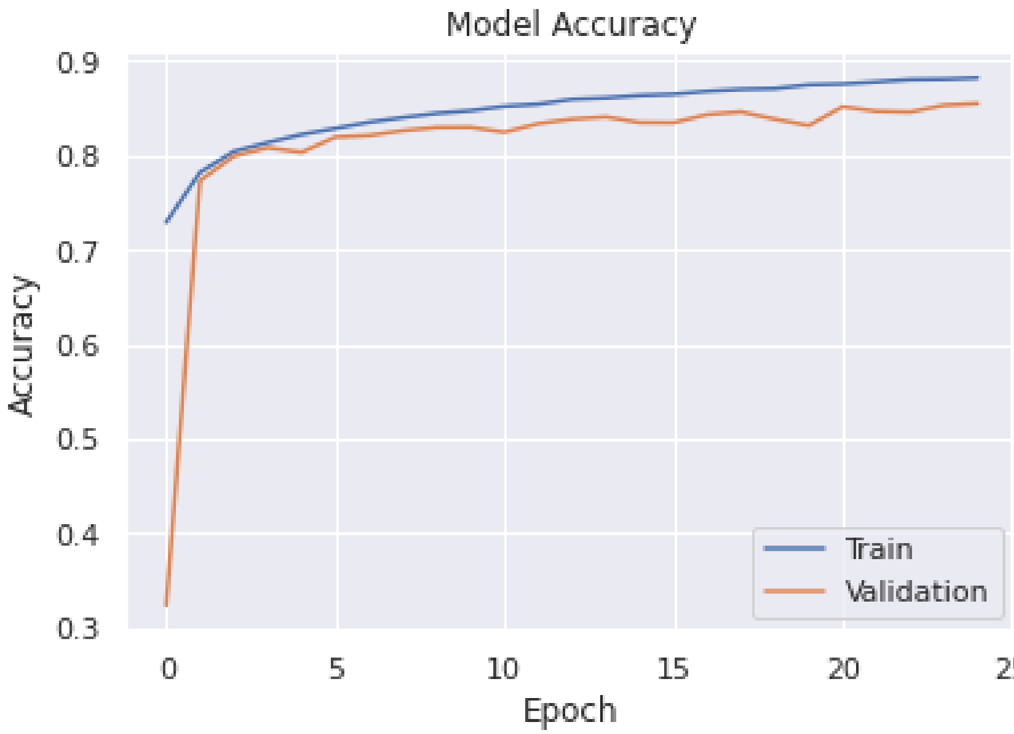
[20]:
```
plt.plot(History_cnn5.history['loss'])
plt.plot(History_cnn5.history['val_loss'])
plt.title('Loss over epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='best')
plt.show()
```



Performance Learning Curve

The gap between training and validation accuracy indicats of overfitting of model. From the graph below it we can observe that the DNN model is a little overfit.

[21]:
```
plt.plot(History_cnn5.history['accuracy'])
plt.plot(History_cnn5.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='best')
plt.show()
```



2.2 2. Resnet Model

[23]:
```
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
```

[24]:
```
# 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
```

[25]:
```
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
```

[26]:
```
start2 = time.time()
model_resnet = resnet_v1(input_shape = (28,28,1), depth = 2)
```

[27]:
```
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)
end2 = time.time()
```

```
Epoch 1/25
375/375 [==============================] - 2s 4ms/step - loss: 1.0832 -
accuracy: 0.6284 - val_loss: 0.7272 - val_accuracy: 0.7132
Epoch 2/25
375/375 [==============================] - 1s 4ms/step - loss: 0.7204 -
accuracy: 0.7138 - val_loss: 0.7264 - val_accuracy: 0.7125
Epoch 3/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6878 -
accuracy: 0.7270 - val_loss: 0.7073 - val_accuracy: 0.7203
Epoch 4/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6694 -
accuracy: 0.7327 - val_loss: 0.6906 - val_accuracy: 0.7225
Epoch 5/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6548 -
accuracy: 0.7360 - val_loss: 0.6705 - val_accuracy: 0.7305
Epoch 6/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6318 -
accuracy: 0.7444 - val_loss: 0.6655 - val_accuracy: 0.7325
Epoch 7/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6207 -
accuracy: 0.7525 - val_loss: 0.6505 - val_accuracy: 0.7307
Epoch 8/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6189 -
accuracy: 0.7481 - val_loss: 0.6579 - val_accuracy: 0.7290
```

```
Epoch 9/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6131 -
accuracy: 0.7551 - val_loss: 0.6423 - val_accuracy: 0.7468
Epoch 10/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6129 -
accuracy: 0.7542 - val_loss: 0.6642 - val_accuracy: 0.7225
Epoch 11/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6120 -
accuracy: 0.7546 - val_loss: 0.6414 - val_accuracy: 0.7467
Epoch 12/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6056 -
accuracy: 0.7552 - val_loss: 0.6344 - val_accuracy: 0.7480
Epoch 13/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6025 -
accuracy: 0.7555 - val_loss: 0.6407 - val_accuracy: 0.7472
Epoch 14/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6043 -
accuracy: 0.7580 - val_loss: 0.6441 - val_accuracy: 0.7357
Epoch 15/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6039 -
accuracy: 0.7585 - val_loss: 0.6526 - val_accuracy: 0.7442
Epoch 16/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6025 -
accuracy: 0.7570 - val_loss: 0.6375 - val_accuracy: 0.7437
Epoch 17/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6047 -
accuracy: 0.7579 - val_loss: 0.6490 - val_accuracy: 0.7442
Epoch 18/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6020 -
accuracy: 0.7595 - val_loss: 0.6389 - val_accuracy: 0.7500
Epoch 19/25
375/375 [==============================] - 1s 4ms/step - loss: 0.5943 -
accuracy: 0.7608 - val_loss: 0.6401 - val_accuracy: 0.7483
Epoch 20/25
375/375 [==============================] - 1s 4ms/step - loss: 0.5951 -
accuracy: 0.7592 - val_loss: 0.6441 - val_accuracy: 0.7432
Epoch 21/25
375/375 [==============================] - 1s 4ms/step - loss: 0.5992 -
accuracy: 0.7571 - val_loss: 0.6421 - val_accuracy: 0.7475
Epoch 22/25
375/375 [==============================] - 1s 4ms/step - loss: 0.5895 -
accuracy: 0.7646 - val_loss: 0.6354 - val_accuracy: 0.7503
Epoch 23/25
375/375 [==============================] - 1s 4ms/step - loss: 0.6018 -
accuracy: 0.7588 - val_loss: 0.6386 - val_accuracy: 0.7467
Epoch 24/25
375/375 [==============================] - 1s 4ms/step - loss: 0.5990 -
accuracy: 0.7588 - val_loss: 0.6469 - val_accuracy: 0.7413
```

```
Epoch 25/25
375/375 [==============================] - 1s 4ms/step - loss: 0.5968 -
accuracy: 0.7594 - val_loss: 0.6367 - val_accuracy: 0.7472
```

[28]:
```
end2 - start2
```

[28]: 37.033555030822754

[29]:
```
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
```

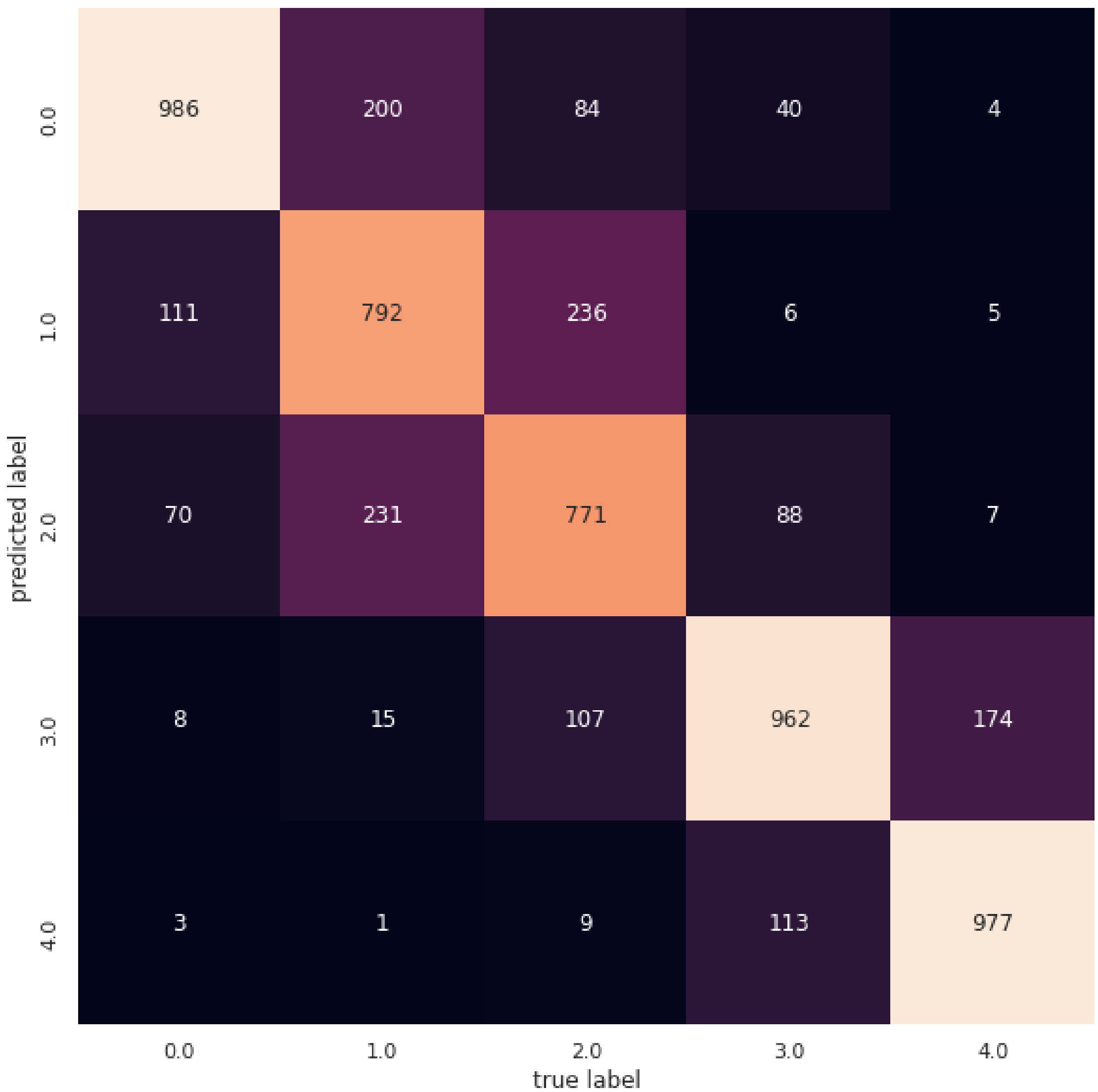
[29]: 0.748

[30]:
```
predict_labels_resnet = []
for pred in y_pred_resnet:
    predict_labels_resnet.append(np.argmax(pred))
```

2.2.1 Confusion Matrix

The confusion indicates that the CNN with ResNet succesfully identified most of the labels labels, however there is some confusion to separate labels like 1,2 & 3 and 4 & 5.

[31]:
```
mat_resnet = confusion_matrix(y_test, predict_labels_resnet)
plt.figure(figsize=(10, 10))
sns.set()
sns.heatmap(mat_resnet.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=np.unique(y_test),
            yticklabels=np.unique(y_test))
plt.xlabel('true label')
plt.ylabel('predicted label')
plt.show()
```



Precision : Precision gives how precisely our model was able to identify labels.The model was able to precisly calssify labels 0 and 4 compared to others.

Recall : In an imbalanced classification problem with more than two classes, recall is calculated as the sum of true positives across all classes divided by the sum of true positives and false negatives across all classes. The model predicted correct labels 84% correct labels for class 0 and 4 compared to others.

f1-score : The F1 score can be interpreted as a weighted average of the precision and recall values, where an F1 score reaches its best value at 1 and worst value at 0. In our case, it is at moderate level.

Support : Support shows number of occurrences of each classes in predicted test data.

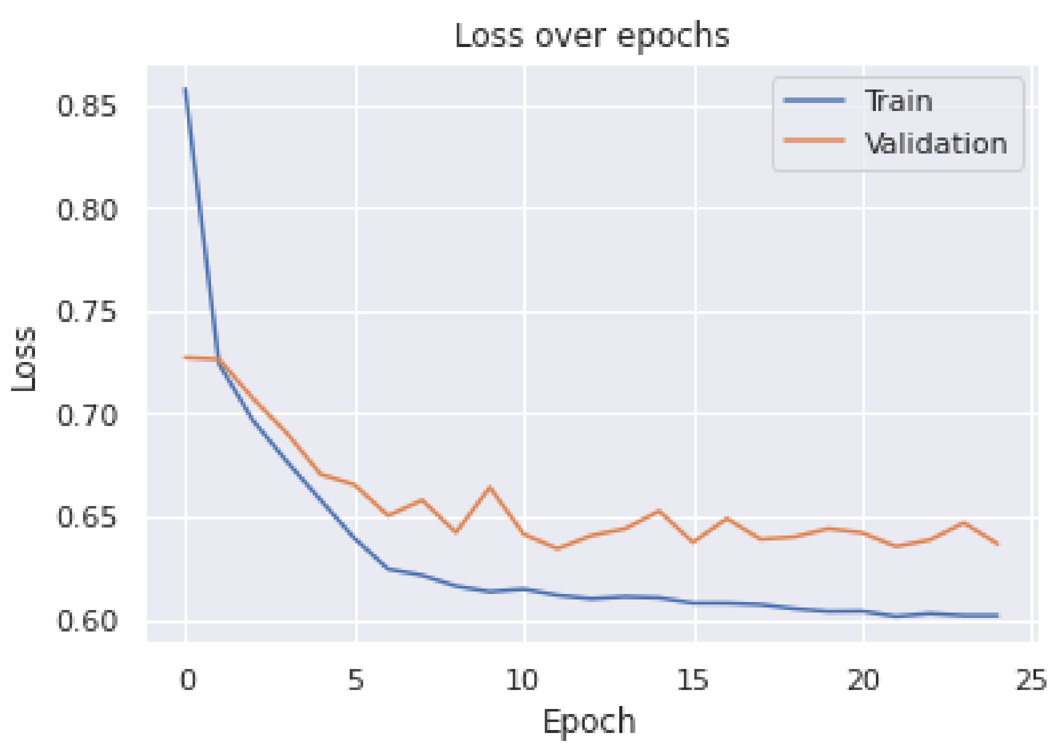
[32]:
```
print(classification_report(predict_labels_resnet,y_test))
```

```
              precision    recall  f1-score   support

           0       0.84      0.75      0.79      1314
           1       0.64      0.69      0.66      1150
           2       0.64      0.66      0.65      1167
           3       0.80      0.76      0.78      1266
           4       0.84      0.89      0.86      1103

    accuracy                           0.75      6000
   macro avg       0.75      0.75      0.75      6000
weighted avg       0.75      0.75      0.75      6000
```

Optimization Learning Curve

We have tried various combination of parameters, and the above results preresentated here are the best obtained by the combination of parameters we tried. The curve shows that the CNN is moderatly good fit of for the data and the data has good learning rate.

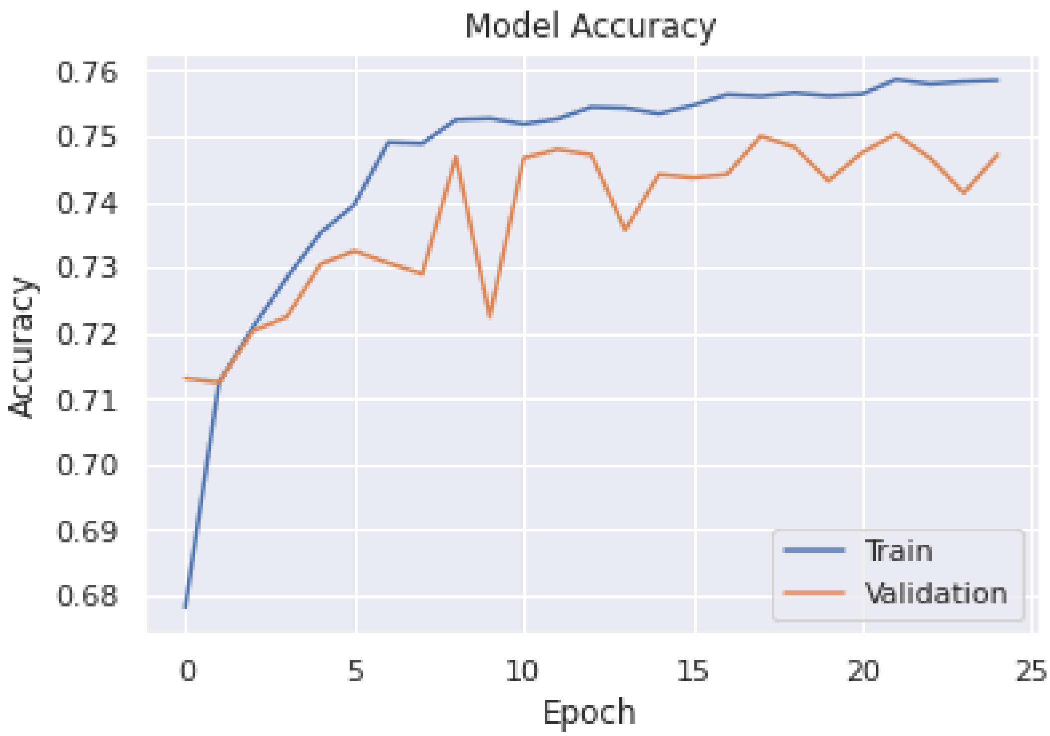
[33]:
```
plt.plot(History_resnet.history['loss'])
plt.plot(History_resnet.history['val_loss'])
plt.title('Loss over epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='best')
plt.show()
```



Performance Learning Curve

The gap between training and validation accuracy indicats of overfitting of model. From the grpah below it we can observe that the DNN model is a overfit.

[34]:
```
plt.plot(History_resnet.history['accuracy'])
plt.plot(History_resnet.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='best')
plt.show()
```



2.3 Simple CNN vs ResNet

- From learning curves, it can be said that simple CNN is better than Resnet, because dataset is small and Resnet performs well on bigger datasets.
- Saying that, we observed 85.88 and 74.8 accuracy on CNN and Resnet respectively.
- Additionally, Time taken by both models is 98.73 and 37.03 seconds.
- From the confusion matrix, it's clear that CNN correctly classifies more labels than Resnet.

2.4 References:

- https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5
- https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33