1.1 - CM1 : Design and Implementation Choices of your Model (15 points)

COVID 19 Dataset can be achieved through Classification Problem, wherein we can predict the Outcome1 column which is a Categorical variable.

Data Type of Features : Datetime, Continuous, Categorical
Data Type of Target Variable : Categorical

We can achieve the solution for this problem by using fully connected DNN and RNN.

Deep Neural Networks (DNN) is a computation graph where some nodes are input nodes, some output nodes and the rest are termed as hidden nodes. For simplicity, we can imagine a canonical deep neural network consisting of multiple layers where the layers are stacked vertically one on top of the other. Each layer consists of one or more nodes. The nodes simulate “neurons” which are computing elements in our brain. Inputs are applied on the input layer (imagine this as the bottom most layer for simplicity) and the outputs are produced at the output layer. As the network has a graph structure, a strictly vertical stack is a special case but we imagine our deep neural network (DNN) as a vertically layered architecture for the purpose of answering this question. The DNN can be specialized and architected in various ways. A fully connected deep neural network consists of typically more than 1 hidden layer and each node in a given layer (layer i) is connected to every other node in the layer immediately above it (that is nodes in layer i+1).

A Recurrent Neural Network (RNN) is used to process sequences and can be termed as a DNN in the “temporal” sense as opposed to purely being spatially vertical. Here, the hidden activations from time step t is used as the context to compute the activations of time step t+1. Hence we can view this as a hidden layer h(t+1) being stacked on top of h(t) and in this sense it is a deep network. If the sequence length is T, the depth of this network in terms of hidden layers executed is T.

We have done the implementation using both DNN and RNN. But The main model with high efficiency is DNN. So let us first explore it further.

Model Details

1. Network Architecture used : DNN with Dense Layers
2. Number of Layers : 1 input, 3 hidden and 1 output.
3. Optimizer used: SGD
4. Activation function used: relu and softmax
5. Regularization method used: Dropout

DNN with Dense Layers:

The given COVID data consist of categorical, continuous and Datetime features. CNN may not work efficiently. The term deep neural nets refers to any neural network with several hidden layers. Convolutional neural nets are a specific type of deep neural net which are especially useful for image recognition. Specifically, convolutional neural nets use convolutional and pooling layers, which reflect the translation-invariant nature of most images. Thus we are using a fully connected DNN for this specific data.

Number of Layers : 5 layers

Generally 2 hidden layers will enable the network to model any arbitrary function.Using too few neurons in the hidden layers will result in something called underfitting. Underfitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set.
Using too many neurons in the hidden layers can result in several problems. First, too many neurons in the hidden layers may result in overfitting. Overfitting occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers

So we choose 5 layers in our architecture with 1 input, 3 hidden and 1 output layers.

Optimizer used: SGD

An optimizer is one of the two arguments required for compiling a Keras model.Keras provides the SGD class that implements the stochastic gradient descent optimizer with a learning rate and momentum. An instance of the class must be created and configured, then specified to the “optimizer” argument when calling the fit() function on the model.

SGD's simplicity makes it a good choice for shallow networks. However, it also means that SGD converges significantly more slowly than other, more advanced algorithms that are also available in `keras`.

We used SGD Optimizer with lr=0.01 and momentum=0.8 where Momentum helps accelerate SGD in the correct directionThe amount that the weights are updated during training is referred to as the

step size or the “learning rate”. Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.

Activation function used: relu and softmax

A Rectified Linear Unit (A unit employing the rectifier is also called a rectified linear unit ReLU) has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to the input. The operation of ReLU is closer to the way our biological neurons work.

The softmax function is often used in the final layer of a neural network-based classifier. Such networks are commonly trained under a log loss (or cross-entropy) regime, giving a non-linear variant of multinomial logistic regression.

We used Relu for the top layers and softmax activation function for the final layer.

Regularization method used: Dropout

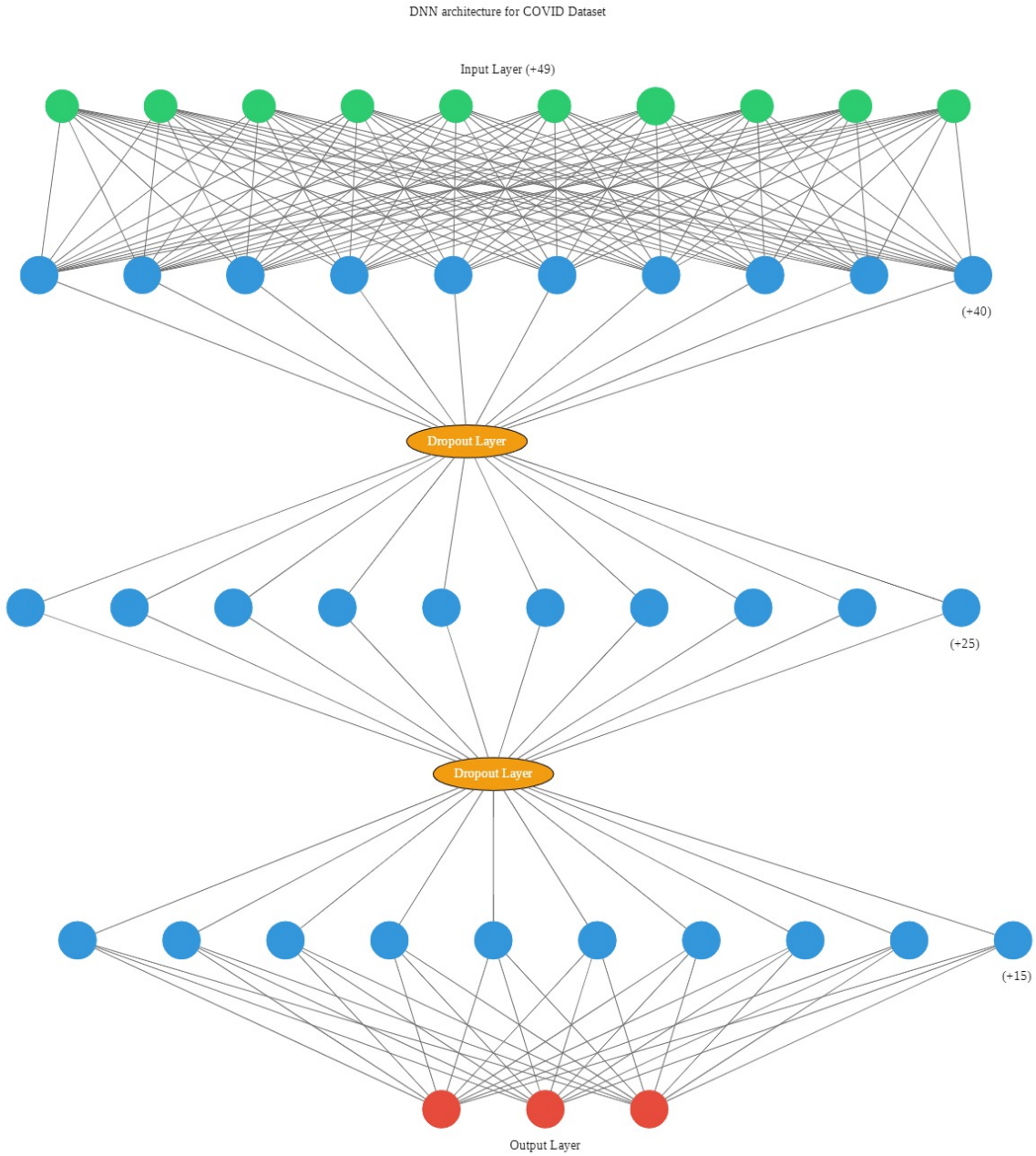
Dropout is implemented per-layer in a neural network.The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged.It can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers such as the long short-term memory network layer.

Dropout in our model is implemented on 1 input and 1 hidden layer. It should not used on the output layer.

Visualization of Architecture:

- Network is made up of an input layer consisting of 59 neurons, an output layer consisting of 3 neuron and 3 hidden layers, the first hidden layer consists of 60 neurons, the second hidden layer consist of 40 neurons while the last hidden layer consists of 20 neurons.
- The activation function employed for each hidden layer is Rectified Linear Unit (ReLU) which outputs zero when the value of the input is less than or equal to zero and outputs the value of the input when the value of the input is greater than zero.
- The activation function used for the output layer is the Softmax function which converts the vector of numbers into vector of probabilities, where each probability defines the likelihood of a number in the vector with respect to the other numbers.
- The optimizer employed for this network is the stochastic gradient decent optimizer which updates the networks parameters for each training sample until the loss function converges.
- The regularization method used in this network is the dropout method where 20% of the neurons for the first and second hidden layers are randomly dropped during training Is other to prevent over fitting
- Generally, the number of parameters present in this model is 6923.

The diagram for the network is shown below


DNN architecture for COVID Dataset
Input Layer (+49)
(+40)
Dropout Layer
(+25)
Dropout Layer
(+15)
Output Layer

Does a combination of fully-connected and RNN/LSTM perform better ?

Although fully connected networks make no assumptions about the input they tend to perform less and aren't good for feature extraction. Plus they have a higher number of weights to train that results in high training time while on the other hand RNNs are trained to identify and extract the best features from the data for the problem at hand with relatively fewer parameters to train.

Even if we had enough "computing power" and we weren't at all interested in **efficiency** (i.e. solving the same task quicker with less parameters), there is still the issue that Fully Connected Neural Networks tend to **overfit** very easily. In RNNs: Because they can **exploit the sequential nature** of the data, they can solve problems that wouldn't be possible with FC networks. Through the context of its previous states it can extract information and enhance its performance. This gives it the ability to identify trends and seasonalities in the data, which are impossible to identify in a FC network. Even if we had an arbitrarily-large FC network and we passed it the full sequence in every time step (so that it doesn't miss out on any of the context), that would mean that the RNN requires less information (i.e. fewer features) about that given time step (because they can obtain this information from another time step). This makes less dimensions for the data so less *room* to overfit.

From the above Analysis of both fully connected network and LSTM along with a combination of RNN, We can say that a combination of both fully connected and LSTM worked better.

CM2 (5 points)

5 pages submitted

submit content here

1.2 - CM2 : Implementation Choices of Model (5 points)

Importing Libraries

In [2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import datetime
from sklearn.utils import shuffle
from sklearn.preprocessing import MinMaxScaler
from keras.layers import Dropout
from keras.constraints import maxnorm
from sklearn.metrics import confusion_matrix
import itertools
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
from keras.callbacks import EarlyStopping
from tensorflow.keras import callbacks
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation , Dense, Flatten
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.metrics import categorical_crossentropy
from ann_visualizer.visualize import ann_viz;
import time
import os
from datetime import timedelta

import logging
logging.basicConfig(format="%(levelname)s - %(asctime)s: %(message)s", datefmt='%H:%M:%S', level=logging.INFO)
```

- The three main libraries used for the implementation of the model are:
  1. Sklearn
  2. Tensorflow
  3. Keras

Data Preprocessing:

Performed Data Cleaning and Dealt with missing data

There are three features having dates. Let us first convert the dates to an integer format.

In [4]:
```
def convert(arg):
  val = pd.to_datetime(arg,format="%Y-%m-%dT",errors = "coerce")
  return val
```

In [5]:
```
# Coverting the datatype for Dates to pandas datatime datatype
df["Accurate_Episode_Date"] = convert(df["Accurate_Episode_Date"])
df["Case_Reported_Date"] = convert(df["Case_Reported_Date"])
df["Test_Reported_Date"] = convert(df["Test_Reported_Date"])
df["Specimen_Date"] = convert(df["Specimen_Date"])
```

In [6]:
```
# Filling the NaT in Specimen_Date and Test_Reported_Date with adjacent values in Case_Reported_Date given that thier E

df['Test_Reported_Date'] = df.apply(lambda row: row['Case_Reported_Date'] if pd.isnull(row['Test_Reported_Date']) else
df['Specimen_Date'] = df.apply(lambda row: row['Case_Reported_Date'] if pd.isnull(row['Specimen_Date']) else row['Speci
```

In [7]:
```
# Converting Dates to integer which result to time in nanoseconds

df["Outbreak_Related"].fillna('No' ,  inplace=True)
df['Accurate_Episode_Date']= pd.to_numeric(df['Accurate_Episode_Date'], downcast='integer')
df['Case_Reported_Date']= pd.to_numeric(df['Case_Reported_Date'], downcast='integer')
df['Test_Reported_Date']= pd.to_numeric(df['Test_Reported_Date'], downcast='integer')
df['Specimen_Date']= pd.to_numeric(df['Specimen_Date'], downcast='integer')
```

In [8]:
```
# Converting Dates from nanoseconds to days

df['Accurate_Episode_Date'] = df.apply(lambda row: row['Accurate_Episode_Date']/8.64e+13,axis=1)
df['Case_Reported_Date'] = df.apply(lambda row: row['Case_Reported_Date']/8.64e+13,axis=1)
df['Test_Reported_Date'] = df.apply(lambda row: row['Test_Reported_Date']/8.64e+13,axis=1)
df['Specimen_Date'] = df.apply(lambda row: row['Specimen_Date']/8.64e+13,axis=1)
```

In [9]:
```
# Checking the mode value in the train value
df["Age_Group"].mode()

df["Age_Group"].fillna("80s",inplace=True)

drop_index = df[df['Client_Gender'] == 'GENDER DIVERSE' ].index
df.drop(drop_index , inplace=True)
df["Outbreak_Related"] = df["Outbreak_Related"].replace({"No":0, "Yes":1})
```

In [10]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 14859 entries, 0 to 14859
Data columns (total 12 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Accurate_Episode_Date   14859 non-null  float64
 1   Case_Reported_Date      14859 non-null  float64
 2   Test_Reported_Date      14859 non-null  float64
 3   Specimen_Date           14859 non-null  float64
 4   Age_Group               14859 non-null  object
 5   Client_Gender           14859 non-null  object
 6   Case_AcquisitionInfo    14859 non-null  object
 7   Reporting_PHU_City      14859 non-null  object
 8   Outbreak_Related        14859 non-null  int64
 9   Reporting_PHU_Latitude  14859 non-null  float64
 10  Reporting_PHU_Longitude 14859 non-null  float64
 11  Outcome1                14859 non-null  object
```

Preparing the data for Training, Validating and Testing

```
[14]: ## Extracting the dependent and the independent variable
X = df.drop(["Outcome1"], axis = 1)
y = df["Outcome1"]

# Label encoding target variable
encoder = LabelEncoder()
encoder.fit(y)
encoded_Y = encoder.transform(y)
```

```
[15]: X_encoded = pd.get_dummies(X)
```

```
[16]: # split the data into training and testing sets
from sklearn.model_selection import train_test_split
X_train,X_test,y_train, y_test = train_test_split(X_encoded, encoded_Y, random_state = 270 ,test_size = 0.2)

X_train_value,X_val,y_train_value,y_val = train_test_split(X_train, y_train, random_state = 270 ,test_size = 0.2)

X_train_value = np.array(X_train_value)
y_train_value = np.array(y_train_value)
y_train_value , X_train_value = shuffle(y_train_value , X_train_value)

X_val = np.array(X_val)
y_val = np.array(y_val)
y_val , X_val = shuffle(y_val , X_val)
```

```
[17]: scaler  = MinMaxScaler(feature_range=(0,1))
scaled_X_val = scaler.fit_transform(X_val)
scaled_X_train = scaler.fit_transform(X_train_value)
```

I considered taking the train, val and test split for the model implementation where:
Train data:64%
Validation data:16%
Test data:20%

Model Defining

```
8]: model = Sequential([
                    Dense(units = 50 ,input_shape=(59,), activation='relu',kernel_constraint=maxnorm(3)),
                    Dropout(0.2),
                    Dense(units = 35 , activation='relu',kernel_constraint=maxnorm(3)),
                    Dropout(0.2),
                    Dense(units = 25 , activation='relu',kernel_constraint=maxnorm(3)),
                    Dense(units = 3 , activation = 'softmax')
])
```

The model is built using keras Sequential function and this model takes in input of shape (59,)
and has 3 hidden layers. Here a dropout is applied after the first hidden layer and after the second hidden layer

In [21]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 50)                3000
_________________________________________________________________
dropout (Dropout)            (None, 50)                0
_________________________________________________________________
dense_1 (Dense)              (None, 35)                1785
_________________________________________________________________
dropout_1 (Dropout)          (None, 35)                0
_________________________________________________________________
dense_2 (Dense)              (None, 25)                900
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 78
=================================================================
Total params: 5,763
Trainable params: 5,763
Non-trainable params: 0
_________________________________________________________________
```

- On the image below is shown the summary of the model, here we can see that first hidden layer has 3600 parameters while the second hidden layer contains 2440 parameters, and the last hidden layer contains 820 parameters and the output layer contains 63 parameters.

Model Fitting

In [23]:
```
%%time
model.compile(optimizer=SGD(lr=0.01, momentum=0.8), loss = 'sparse_categorical_crossentropy' , metrics=['accuracy'])
history = model.fit(x= scaled_X_train, y = y_train_value, validation_data=(scaled_X_val , y_val), batch_size=30,
                    callbacks=callbacks,epochs = 50 , shuffle= True, verbose = 2)
```

TESTING

In [26]:
```
%%time
X_val = np.array(X_val)
y_val = np.array(y_val)
X_val , y_val = shuffle(X_val , y_val)

scaled_X_val = scaler.fit_transform(X_val)
predictions = model.predict(x = scaled_X_val,batch_size=10,verbose=0)
rounded_predictions = np.argmax(predictions, axis = -1)
accuracy_score(y_val, rounded_predictions)
```

```
CPU times: user 331 ms, sys: 38.4 ms, total: 370 ms
Wall time: 465 ms
```

Out[26]:
```
0.8910849453322119
```

In [27]:
```
%%time
X_test = np.array(X_test)
y_test = np.array(y_test)
X_test , y_test = shuffle(X_test , y_test)

scaled_X_test = scaler.fit_transform(X_test)

predictions = model.predict(x = scaled_X_test,batch_size=10,verbose=0)
rounded_predictions = np.argmax(predictions, axis = -1)
accuracy_score(y_test, rounded_predictions)
```

```
CPU times: user 337 ms, sys: 45.9 ms, total: 383 ms
Wall time: 329 ms
```

Out[27]:
```
0.9067967698519516
```

The final accuracy of the model is 90.67%

CM3 (20 points)

8 pages submitted

submit content here

CM3: 1.3: Results Analysis (20 points)

---

1. **Run-time performance for training and testing.**

Running time performance for training

```
Epoch  49  run time is:  1.8497970000000095
Total trained time is:  95.78078500000001
CPU times: user 1min 30s, sys: 5.05 s, total: 1min 35s
Wall time: 1min 30s
```

The time taken to train the model is 1 min 30 secs

Running time performance for testing

```
CPU times: user 331 ms, sys: 38.4 ms, total: 370 ms
Wall time: 465 ms
```

The time taken to test the model is 6 sec

Since there are 3 hidden layers and 2 dropout layers, and the model consists of 50 epochs, it takes a basic time of around 1 to 2 mins for every time we train the model.
There is relatively very less amount of data present in test data, the run time to test and evaluate the model is comparatively low.

2. Comparison of the different algorithms and parameters you tried.

There are 2 different approaches tested and compared along with the basic model.

1. DNN with different parameters
2. RNN with LSTM

DNN with different parameters:

- Here we tried to change the number of hidden layers, optimizer and also the activation function as below.

Change in parameters

[88]:
```
model = Sequential([
                    Dense(units = 20 ,input_shape=(59,), activation='elu',kernel_constraint=maxnorm(3)),
                    Dropout(0.2),
                    Dense(units = 20 , activation='elu',kernel_constraint=maxnorm(3)),
                    Dropout(0.2),
                    Dense(units = 20 , activation='elu',kernel_constraint=maxnorm(3)),
                    Dropout(0.2),
                    Dense(units = 20 , activation='elu',kernel_constraint=maxnorm(3)),
                    Dropout(0.2),
                    Dense(units = 10 , activation='elu',kernel_constraint=maxnorm(3)),
                    Dense(units = 3 , activation = 'softmax')
])
```

The change made is by increasing the number of layers which would give the model summary as below.

]:
```
model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_25 (Dense)             (None, 20)                1200
_________________________________________________________________
dropout_14 (Dropout)         (None, 20)                0
_________________________________________________________________
dense_26 (Dense)             (None, 20)                420
_________________________________________________________________
dropout_15 (Dropout)         (None, 20)                0
_________________________________________________________________
dense_27 (Dense)             (None, 20)                420
_________________________________________________________________
dropout_16 (Dropout)         (None, 20)                0
_________________________________________________________________
dense_28 (Dense)             (None, 20)                420
_________________________________________________________________
dropout_17 (Dropout)         (None, 20)                0
_________________________________________________________________
dense_29 (Dense)             (None, 10)                210
_________________________________________________________________
dense_30 (Dense)             (None, 3)                 33
=================================================================
Total params: 2,703
Trainable params: 2,703
Non-trainable params: 0
_________________________________________________________________
```

Compiling the model:

- Optimizer used: Adam

```
In [98]: %%time
model.compile(optimizer="adam", loss = 'sparse_categorical_crossentropy' , metrics=['accuracy'])
history = model.fit(x= scaled_X_train, y = y_train_value, validation_data=(scaled_X_val , y_val), batch_size=30,
                    callbacks=callbacks,epochs = 50 , shuffle= True, verbose = 2)
```

Epoch 16 run time is: 2.5514639999999144

Testing the model

TESTING

```
In [94]: %%time
X_val = np.array(X_val)
y_val = np.array(y_val)
X_val , y_val = shuffle(X_val , y_val)

scaled_X_val = scaler.fit_transform(X_val)
predictions = model.predict(x = scaled_X_val,batch_size=10,verbose=0)
rounded_predictions = np.argmax(predictions, axis = -1)
accuracy_score(y_val, rounded_predictions)
```

```
CPU times: user 421 ms, sys: 55.7 ms, total: 477 ms
Wall time: 416 ms
```

```
Out[94]: 0.8528174936921783
```

```
In [95]: %%time
X_test = np.array(X_test)
y_test = np.array(y_test)
X_test , y_test = shuffle(X_test , y_test)

scaled_X_test = scaler.fit_transform(X_test)

predictions = model.predict(x = scaled_X_test,batch_size=10,verbose=0)
rounded_predictions = np.argmax(predictions, axis = -1)
accuracy_score(y_test, rounded_predictions)
```

```
CPU times: user 497 ms, sys: 63 ms, total: 560 ms
Wall time: 676 ms
```

```
Out[95]: 0.8687752355316285
```

Comparison

- Accuracy for the model with less number of layers and SGD optimizer along with relu activation function is : **90.6**
- Accuracy for the model with more number of layers and Adam optimizer along with elu activation function is : **86.8**
- From the observation, the model with more hidden layers in a fully connected network would be less efficient compared to a less numbered hidden layer network.
- Here the model faced underfitting in changing the parameters.
- Coming to the optimizers, SGD worked well for the given data compared with Adam. SGD could work better for shallow networks.

RNN with LSTM

- Here we tried to implement a RNN network using the LSTM approach.
- No of layers: 1 input 4 hidden 1 output.
- Activation function : tanh and softmax

Designing the model:

```
%%time
LAYERS = [45, 45, 45,1]           # number of test examples (2D),full=X_test.shape[0]
N = 59                 # number of features
BATCH = 60                       # batch size
EPOCH = 50                           # recurrent dropout rate

# Build the Model
model = Sequential()
model.add(LSTM(units = LAYERS[0] ,input_shape=(N,1),return_sequences=True))
model.add(LSTM(units = LAYERS[1] ,return_sequences=True))
model.add(BatchNormalization())
model.add(LSTM(units=50, activation='tanh',return_sequences=True))
model.add(Dense(units=LAYERS[3], activation='softmax'))
```

Compiling the model

Optimizer used: SGD with learning rate - 0.01 and momentum = 0.8

```
9]: model.compile(optimizer=SGD(lr=0.01, momentum=0.8), loss = 'sparse_categorical_crossentropy' , metrics=['accuracy'])
```

Fitting the model on train data:

```
In [65]: history=model.fit(reshape_X_train, reshape_y_train,
                           epochs=10,callbacks=callbacks,
                           batch_size=80,
                           shuffle=True,verbose=1,)
```

```
Epoch 10/10
149/149 [==============================] - 19s 125ms/step - loss: 4.0775 - accuracy: 0.3341
```

Accuracy obtained = 33.4%

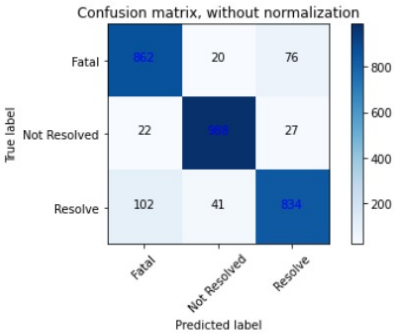
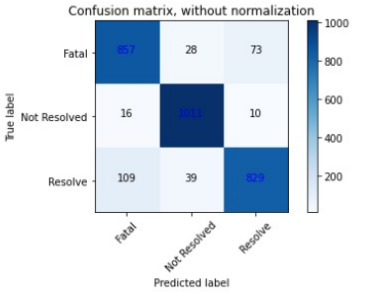
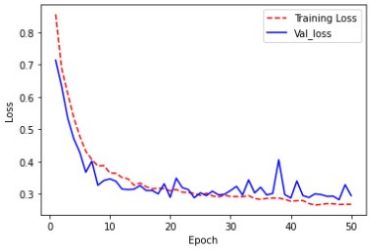
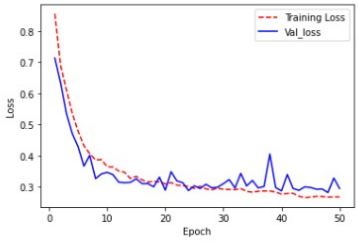
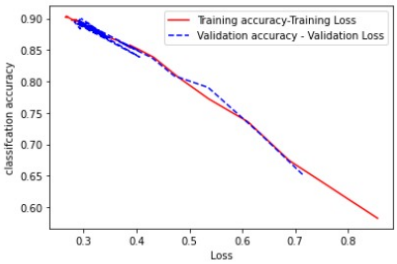
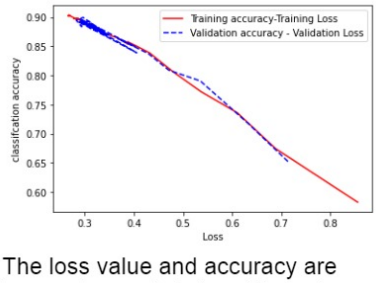
Comparison

- RNN/LSTM worked less efficiently compared to DNN
- RNN/LSTM Accuray is 33% and DNN model accuracy is 90%
- When deeply observed, the data is not completely based on time series to get the complete functionality of LSTM.
- RNN could classify the problems well but there is a huge variation with the given data.
- The time taken to execute is also high and which in turn is a less efficient feature to consider for the given data set.

3. **You can use any plots to explain the performance of your approach. But at the very least produce two plots, one of training epoch vs. loss and one of classification accuracy vs. loss on both your training and validation sets.**

   There are totally 3 models implemented:

   1. **DNN with 2 hidden layers**
   2. **DNN with 5 hidden layers**

Metrics	DNN with 2 hidden layers	DNN with 5 hidden layers
Confusion matrices	 The accuracy from this confusion matrix is 91%	 The accuracy from this confusion matrix is 90%
Loss vs Epoch	 The loss value decreased with increase in epochs.	 The loss value decreased with increase in epochs.
Loss vs Accuracy	 The loss value and accuracy are inversely correlated and there is no much difference with the validation and training values	 The loss value and accuracy are inversely correlated and there is no much difference with the validation and training values

The performance of the RNN/LSTM is very low and the model got under fitted. Hence we selected DNN fully connected network as the main model.

Comparing Evalution Metrics

For DNN with less number of layers

Metrics(Precision,Recall, F-Score) for model 1:

```
In [27]: #classification Report for model 1
         targets = ['1','2','3','4','5']
         print(classification_report(Y_true, Y_pred_classes, target_names = targets))
```

	precision	recall	f1-score	support
1	0.92	0.93	0.93	2334
2	0.92	0.89	0.90	2421
3	0.87	0.88	0.88	2456
4	0.94	0.92	0.93	2425
5	0.94	0.98	0.96	2364
accuracy			0.92	12000
macro avg	0.92	0.92	0.92	12000
weighted avg	0.92	0.92	0.92	12000

For DNN with more number of Layers

Metrics(Precision,Recall, F-Score) for model 2:

```
In [38]: #classification Report for model 2
         targets = ['1','2','3','4','5']
         print(classification_report(Y_true1, Y_pred_classes1, target_names = targets))
```

	precision	recall	f1-score	support
1	0.91	0.92	0.92	2334
2	0.90	0.87	0.89	2421
3	0.86	0.88	0.87	2456
4	0.92	0.92	0.92	2425
5	0.96	0.95	0.95	2364
accuracy			0.91	12000
macro avg	0.91	0.91	0.91	12000
weighted avg	0.91	0.91	0.91	12000

From the above classification matrix, we can say that the model #1 is well fitted however it might be little underfitted for the label 3 compared to other labels.

From the above classification matrix, we can say that the model #2 is well fitted good however it might be little underfitted for the label 3 compared to other labels.

But, when both the models are compared, we see that model #1 is fit good for the given data and model #2 is slightly underfit in comparison.

When changing the activation function, optimizers and the number of layers for the given data in different models there is a final model which is standing better compared to the remaining.
Its DNN fully connected network with SGD optimizer.
The final accuracy for that model is 90.67%

CM4 (15 points)

5 pages submitted

submit content here

CONVOLUTION NEURAL NETWORKS:

Convolutional Neural Networks come under the domain of Deep Learning in Machine Learning. Image classification involves the process of feature extraction from the image to observe some interesting patterns in the dataset. CNN models are used predominantly in the image data space. The practical benefit of CNN is that having fewer parameters majorly improves the time it takes to learn as well as reduces the amount of data required to train the model. So, instead of a fully connected network of weights from each pixel, a CNN has just enough weights to look at a small patch of the image. Another major benefit of using CNNs over NNs is that the input images need not be flattened to 1D, as CNNs are capable of working with image data in 2D. This helps in retaining the "spatial" properties of images.

When implementing CNN, we make use of the filters which are of various kinds, designed according to their purpose. These filters help in exploiting the spatial locality of a particular image by enforcing a local connectivity pattern between neurons.

Convolution means a pointwise multiplication performed on two functions, to generate a third function. In image classification using CNN, one of these functions is our image pixels matrix and another is the filter. The resulting matrix is called an "Activation Map" or "Feature Map".

DATASET INFORMATION:

Fashion MNIST is a dataset comprising of 60000 samples of data,with 2 columns namely Features and the Target. Features contain the pixel information of the input images and the Target is the label value ranging 1-5 for each image.

Each sample is a 28x28 grayscale image with a label associated with one of the 5 labels (1-5). It is a more challenging classification problem than MNIST and top results are achieved by deep learning convolutional neural networks

Image Samples Details:

- Each image sample is 28 pixels in height and 28 pixels in width, such that each image has a total of 784 pixels.
- Every pixel has unique pixel-value such that the lightness or darkness of that pixel is indicated using the pixel value.
- These pixel-values in this dataset are normalized to have their values between [0-1].
- The number of samples for each class is 12000 which can be visualized from the given dataset.

For the given dataset, we see that all the labels are unique and there are no missing values as well.

DESIGN OF THE IMPLEMENTATION MODEL:

Here, we configured the CNN to process input images of size (28, 28, 1), which is the format of the FashionMNIST images. I do this by passing the argument input_shape=(28, 28, 1) to the first layer.

The Conv2D layers in the model are used for performing the convolution operation, for extracting the features from the input images by sliding a convolution filter over the input to produce a feature map/activation map.

To combat overfitting, we used the Dropout layers, a powerful regularization technique. Dropout is the method used to reduce overfitting. It forces the model to learn multiple independent representations of the same data by randomly disabling neurons in the learning phase.

ACTIVATION FUNCTIONS:

A Rectified Linear Unit (A unit employing the rectifier is also called a rectified linear unit ReLU) has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to the input. The operation of ReLU is closer to the way our biological neurons work.

The softmax function is often used in the final layer of a neural network-based classifier. Such networks are commonly trained under a log loss (or cross-entropy) regime, giving a non-linear variant of multinomial logistic regression. We used Relu for the top layers and softmax activation function for the final layer.

We have implemented our model with 4 Conv2D layers,3 Dropout layers and 2 Dense layers.

ARCHITECTURE:

- The first and the second Convolution layers have 32 filters with kernel size 5x5 which gives us 32 feature maps as output for each of these conv 2D layers.
- The next layer is the 1st drop out layer which will randomly disable 40% of the outputs.
- We again have the 3rd and 4th Convolution layers with 64 filters and kernel size 3x3 which gives us 64 feature maps as output for each of these conv 2D layers.
- The next layer is the 2nd drop out layer which will randomly disable 40% of the outputs from previous layers and gives an input to next layers.
- We used the ReLU activation function for our convolution layer with stride set to 1 in the first 2 conv2D layers and as 2 in last 2 conv2D layers. The filter movement over the pixels is decided using the stride value.
- We used the ‘same’ padding in our model so that zero values are added around the input such that the output has the same size as the input.

The next step is to feed this last output tensor into a stack of Dense layers, otherwise known as fully-connected layers. These densely connected classifiers process vectors, which are 1D, whereas the current output is a 3D tensor.

- We are using the Flatten() method to convert the 3D outputs to 1D, and then add 2 Dense layers on top.
- We did a 5-way classification (as there are 5 classes of fashion images), using a final layer with 5 outputs and a SoftMax activation function. SoftMax activation enables to calculate the output based on the probabilities. Each class is assigned a probability and the class with the maximum probability is the model's output for the input.

When compiling the model, we chose categorical_crossentropy as the loss function (which is relevant for multiclass, single-label classification problem) and RMSprop optimizer.

- The cross-entropy loss calculates the error rate between the predicted value and the original value. Categorical is used because there are 5 classes to predict from.
- The optimizer is responsible for updating the weights of the neurons via backpropagation. It calculates the derivative of the loss function with respect to each weight and subtracts it from the weight. That is how a neural network learns.

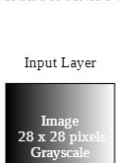
The dimensions of the feature maps change with every successive layer and can be observed from the model summary.

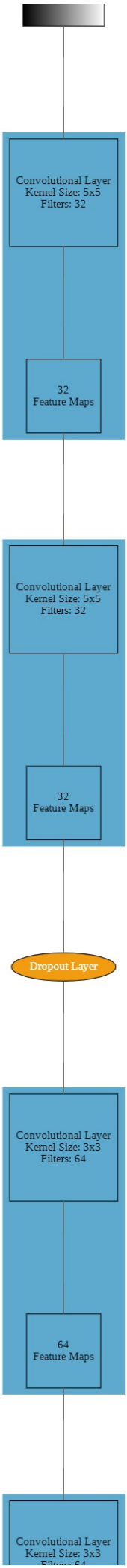
Here, we trained the model with batch size of 100 and 20 epochs on both training and validation data.

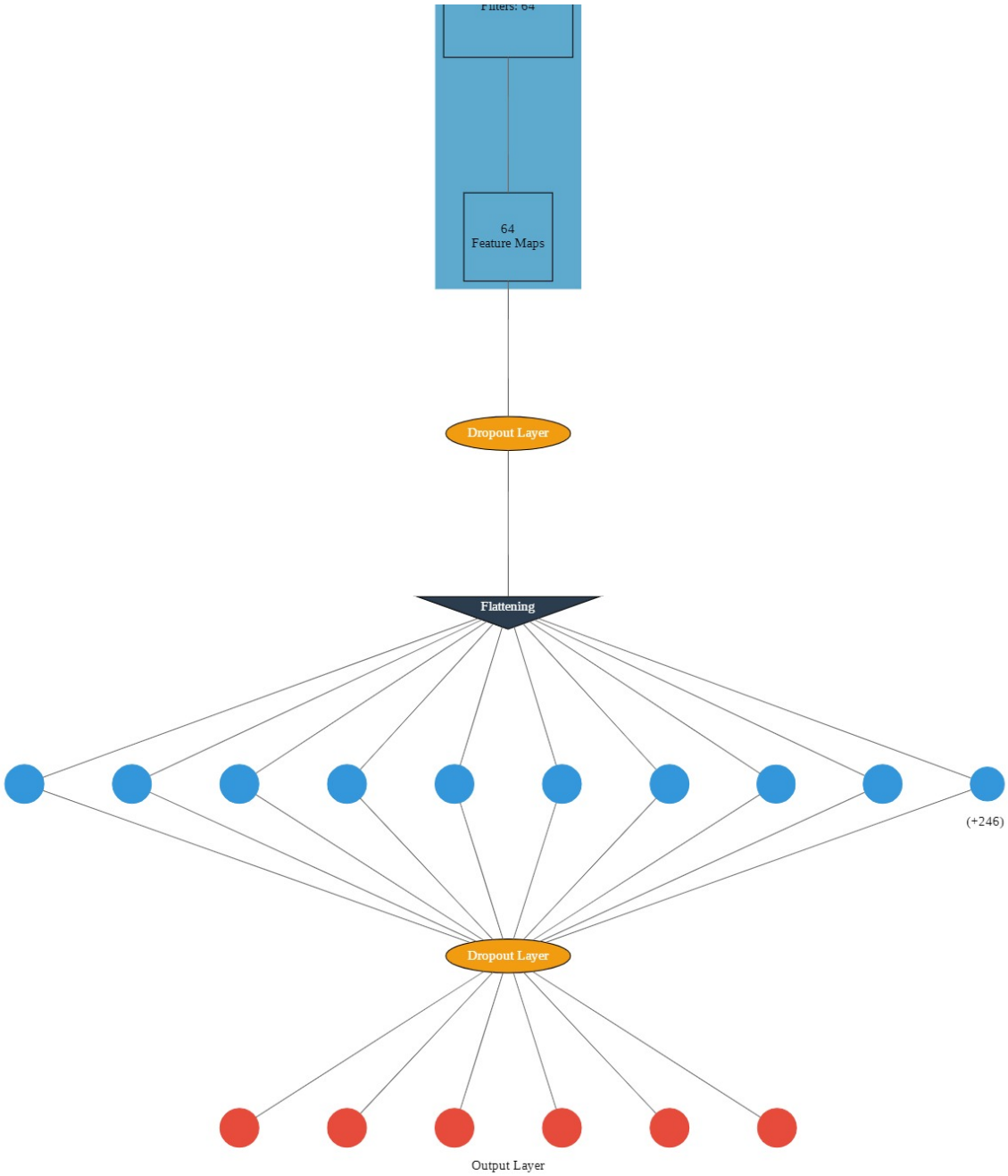
We evaluated this model on the unseen test data and we achieved an accuracy of 91.78%.

The various performance metrics like precision, recall etc., are calculated for the model and a confusion matrix is also plotted which tells us how many actual labels are predicted correctly and incorrectly.

The architecture of our implemented model is as shown:




Convolutional Layer
Kernel Size: 5x5
Filters: 32
32
Feature Maps
Convolutional Layer
Kernel Size: 5x5
Filters: 32
32
Feature Maps
Dropout Layer
Convolutional Layer
Kernel Size: 3x3
Filters: 64
64
Feature Maps
Convolutional Layer
Kernel Size: 3x3


Filters: 64
64
Feature Maps
Dropout Layer
Flattening
(+246)
Dropout Layer
Output Layer

It was accepted globally that deep neural networks can learn more complex functions and representations of the input data that could lead to a better performance. But it was soon discovered that adding more layers could have a negative impact on the ultimate performance of the model which is commonly quoted as degradation problem. This could be handled by adding residual blocks in which intermediate layers of a block learn a residual function with reference to the block input. Using ResNet on the plain CNN model might significantly enhance the performance of neural networks with more layers, but it would actually consume more time and may prove to additional computational cost and would obviously increase the complexity of the model architecture.

CM5 (5 points)

6 pages submitted

submit content here

Implementation of CNN on Fashion MNIST dataset

Importing libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
%matplotlib inline
import time
import tensorflow
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras.utils import to_categorical
from keras.optimizers import RMSprop
from keras.callbacks import ReduceLROnPlateau
import warnings
warnings.filterwarnings('ignore')
from sklearn.metrics import confusion_matrix, classification_report,plot_confusion_matrix
from sklearn.metrics import accuracy_score,precision_score,recall_score,f1_score
import itertools
```

The three main libraries used for the implementation of the model are

- sklearn
- Tensorflow
- Keras

Exploring the given Fashion MNIST dataset:

In [2]:
```
#Loading the given dataset
ds = np.load('fashion_mnist_dataset_train.npy',allow_pickle=True).item()
ds
```

Out[2]:
```
{'features': array([[[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]],

       [[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]],

       [[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
```

```
                [[0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 ...,
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.]],

                [[0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 ...,
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.]],

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

In [3]:
```
type(ds)
```

Out[3]:
```
dict
```

In [4]:
```
X=ds['features']
X
```

Out[4]:
```
array([[[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]],

       [[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]],

       [[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]],

       ...,
```

In [5]:
```
Y=ds['target']
Y
```

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

In [6]:
```
df = pd.DataFrame(list(zip(X, Y)), columns =['Features', 'Target'])
df
```

Out[6]:

	Features	Target
0	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	5.0
1	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.00392156862745098...	2.0
2	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	1.0
3	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.12...	2.0
4	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	1.0
...	...	...
59995	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	4.0
59996	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	2.0
59997	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	3.0
59998	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	1.0
59999	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...	4.0

In [7]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   Features  60000 non-null  object 
 1   Target    60000 non-null  float64
dtypes: float64(1), object(1)
memory usage: 937.6+ KB
```

In [8]:
```
df.shape
```

Out[8]:
```
(60000, 2)
```

In [9]:
```
#to check unique values
df['Target'].unique()
```

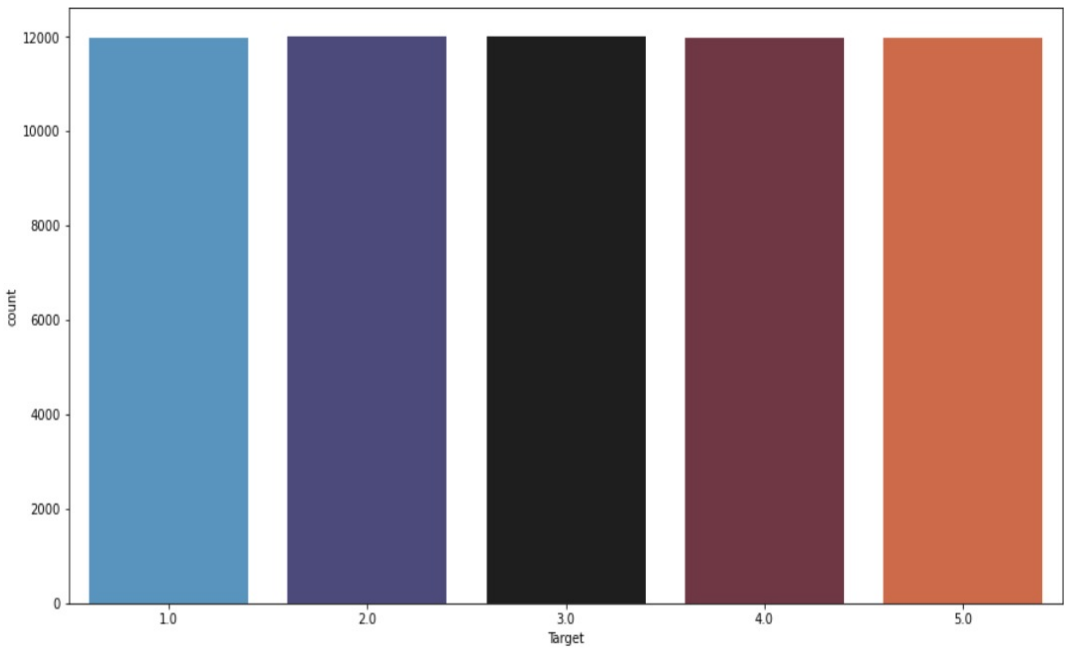
Out[9]:
```
array([5., 2., 1., 3., 4.])
```

In [10]:
```
#To check for missing values/images,if any
df.isnull().sum()
```

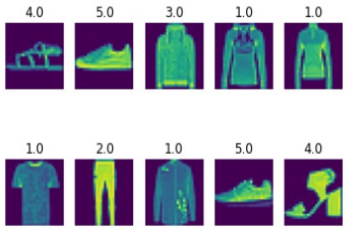
Out[10]:
```
Features    0
Target      0
dtype: int64
```

Visualizing the given data:

In [11]:
```
#finding no.of samples for each class
figure = plt.figure(figsize=(15,8))
sns.countplot(df['Target'],palette="icefire")
plt.show()
```



In [12]:
```
#Random plotting of some samples
sample = np.random.randint(0,high= 59999, size=10)
sample_img = X[sample, :]
sample_label = Y[sample]
plt.figure(figsize=(10, 10))
for i , img in enumerate(sample_img):
    plt.subplot(2,5,i+1)
    plt.axis("off")
    plt.title(sample_label[i])
    img = img.reshape(28,28)
    plt.imshow(img)
plt.show()
```



Preprocessing of Data:

In [13]:
```
# Each image's dimension is 28 x 28
rows, colm = 28, 28
img_shape = (rows, colm, 1)

# Reshaping the input images
X1 = X.reshape(X.shape[0], rows, colm, 1)
```

In [14]:
```
# performing one hot encoding of output variables
Y1 = to_categorical(Y)
```

We have reshaped the input image into 3D from 2D so that it can be fed to our CNN model.

We have performed the one hot encoding of the target variable here to change the output variables to be of binary format. So, that if a sample of class 1.0 exists it is given a 1 or 0 otherwise.

Splitting the given dataset into train, validation and test sets

In [15]:
```
from sklearn.model_selection import train_test_split
X_train1, X_test, Y_train1, Y_test = train_test_split(X1, Y1, test_size = 0.2, random_state = 0)
X_train, X_val, Y_train, Y_val = train_test_split(X_train1, Y_train1, test_size = 0.2, random_state = 0)
```

Here, we have split the dataset into 64% train set,16% validation set (which is 20% of train data) and 20% of unseen test data.

DEFINING THE MODEL:

```
In [16]: cnn_model = Sequential()

         cnn_model.add(Conv2D(32, kernel_size=(5, 5), activation='relu', strides=1, padding='same', input_shape=img_shape))
         #cnn_model.add(BatchNormalization())
         cnn_model.add(Conv2D(32, kernel_size=(5, 5), activation='relu', strides=1, padding='same'))
         #cnn_model.add(BatchNormalization())
         cnn_model.add(Dropout(0.4))

         cnn_model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', strides=2, padding='same'))
         #cnn_model.add(BatchNormalization())
         cnn_model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', strides=2, padding='same'))
         #cnn_model.add(BatchNormalization())
         cnn_model.add(Dropout(0.4))

         cnn_model.add(Flatten())

         cnn_model.add(Dense(256, activation='relu'))
         cnn_model.add(Dropout(0.4))

         cnn_model.add(Dense(6, activation='softmax'))
```

The model is built using Keras Sequential function and this model takes in input of shape (28,28,1) and has 5 hidden layers.

Here one dropout is applied after the first 2 hidden layers, another after the next 2 hidden layers and one before the output layer.

MODEL SUMMARY:

```
In [18]: cnn_model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        25632
_________________________________________________________________
dropout (Dropout)            (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
dropout_1 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 256)               803072
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 1542
=================================================================
Total params: 886,502
Trainable params: 886,502
Non-trainable params: 0
_________________________________________________________________
```

From the image above, we can see that first hidden layer has 832 parameters, the second hidden layer contains 25632 parameters, 3rd hidden layer contains 18496 parameters,4th hidden layer has 36928 parameters and the last hidden layer has 803072 parameters while the output layer contains 1542 parameters.

DEFINING THE OPTIMIZER AND COMPILING THE MODEL:

In [17]:
```
# Define the optimizer and compile the model
optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
cnn_model.compile(optimizer = optimizer , loss = "categorical_crossentropy", metrics=["accuracy"])

# Set a learning rate annealer
learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy', patience=3, verbose=1, factor=0.5, min_lr=0.00001)
```

MODEL FITTING:

In [20]:
```
#training the model 1
#cnn_train_model = cnn_model.fit(X_train, Y_train,batch_size=100,epochs=10,verbose=1,validation_data=(X_val, Y_val))
#starting time
start_time= time.time()
#training the model 1
cnn_train_model = cnn_model.fit(X_train, Y_train,batch_size=100,epochs=20,verbose=1,validation_data=(X_val, Y_val),
                                callbacks=learning_rate_reduction)
```

```
Epoch 1/20
384/384 [==============================] - 353s 904ms/step - loss: 0.9184 - accuracy: 0.6828 - val_loss: 0.6979 - val_accuracy:
0.6908
Epoch 2/20
384/384 [==============================] - 354s 922ms/step - loss: 0.4803 - accuracy: 0.8138 - val_loss: 0.3311 - val_accuracy:
0.8696
Epoch 3/20
384/384 [==============================] - 379s 988ms/step - loss: 0.4276 - accuracy: 0.8361 - val_loss: 0.3698 - val_accuracy:
0.8574
Epoch 4/20
384/384 [==============================] - 350s 911ms/step - loss: 0.3974 - accuracy: 0.8490 - val_loss: 0.3501 - val_accuracy:
0.8659
Epoch 5/20
384/384 [==============================] - 349s 910ms/step - loss: 0.3591 - accuracy: 0.8589 - val_loss: 0.2962 - val_accuracy:
0.8893
Epoch 6/20
384/384 [==============================] - 397s 1s/step - loss: 0.3482 - accuracy: 0.8674 - val_loss: 0.3094 - val_accuracy: 0.
8843
Epoch 7/20
```

As mentioned, the model is fitted with training data with 20 epochs and with a batch size of 100. The corresponding validation loss and accuracies can be also seen for each epoch from the above image.

EVALUATION ON TEST DATA:

Evaluation of the model 1 on test set:

In [23]:
```
#Evaluating the trained model 1 on test dataset
cnn_model_testresults = cnn_model.evaluate(X_test, Y_test, verbose=0)
cnn_model_testloss = cnn_model_testresults[0]*100
cnn_model_testaccuracy=cnn_model_testresults[1]*100
print('Accuracy of our model on test set is equal to '+ str(round(cnn_model_testaccuracy)) + ' %.')
print('Test loss:', cnn_model_testloss)
```

```
Accuracy of our model on test set is equal to 91.78 %.
Test loss: 22.02579826116562
```

We see that **the final accuracy of our model on the test set is 91.78%**

CM6 (20 points)

8 pages submitted

submit content here

RESULT ANALYSIS:

1. **Run-time performance for training and testing.**

MODEL IMPLEMENTED:

In [16]:
```
cnn_model = Sequential()

cnn_model.add(Conv2D(32, kernel_size=(5, 5), activation='relu', strides=1, padding='same', input_shape=img_shape))
#cnn_model.add(BatchNormalization())
cnn_model.add(Conv2D(32, kernel_size=(5, 5), activation='relu', strides=1, padding='same'))
#cnn_model.add(BatchNormalization())
cnn_model.add(Dropout(0.4))

cnn_model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', strides=2, padding='same'))
#cnn_model.add(BatchNormalization())
cnn_model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', strides=2, padding='same'))
#cnn_model.add(BatchNormalization())
cnn_model.add(Dropout(0.4))

cnn_model.add(Flatten())

cnn_model.add(Dense(256, activation='relu'))
cnn_model.add(Dropout(0.4))

cnn_model.add(Dense(6, activation='softmax'))
```

In [17]:
```
# Define the optimizer and compile the model
optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
cnn_model.compile(optimizer = optimizer , loss = "categorical_crossentropy", metrics=["accuracy"])

# Set a learning rate annealer
learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy', patience=3, verbose=1, factor=0.5, min_lr=0.00001)
```

TRAIN DATA:

In [20]:
```
#training the model 1
#cnn_train_model = cnn_model.fit(X_train, Y_train,batch_size=100,epochs=10,verbose=1,validation_data=(X_val, Y_val))
#starting time
start_time= time.time()
#training the model 1
cnn_train_model = cnn_model.fit(X_train, Y_train,batch_size=100,epochs=20,verbose=1,validation_data=(X_val, Y_val),
                                callbacks=learning_rate_reduction)
```

In [21]:
```
#end time
end_time = time.time()
#time taken to train the model on training data
cnn_time = end_time-start_time
print("Time taken for CNN to implement on training set : {:0.2f} minutes".format(cnn_time/60.0))
```

```
Time taken for CNN to implement on training set : 131.55 minutes
```

TEST DATA:

In [39]:
```
#starting time
start_time_1= time.time()
#training the model 1
cnn_train_model_1 = cnn_model.fit(X_test, Y_test,batch_size=100,epochs=20,verbose=1,validation_data=(X_val, Y_val),
                                  callbacks=learning_rate_reduction)
```

In [40]:
```
#end time
end_time_1 = time.time()
#time taken to train the model on training data
cnn_time_1 = end_time_1-start_time_1
print("Time taken for CNN to implement on test set : {:0.2f} minutes".format(cnn_time_1/60.0))
```

```
Time taken for CNN to implement on test set : 44.77 minutes
```

The time taken for the model to be implemented on train set is 131.55 min whereas on the test set it is 44.77 min.

We see that the model designed has 5 hidden layers out of which 4 are 2D conv layers and one is fully connected dense layer. As the number of samples on which the model is trained is comparatively very large compared to the number of unseen test data samples. Hence, the time for implementation is relatively more for training set than the test set.

2. Comparison of the different algorithms and parameters you tried.

We have tried following scenarios:

- CNN with different hyperparameters
- ResNet Implementation

CNN WITH DIFFERENT PARAMETERS:

In [28]:
```
cnn_model1 = Sequential()
cnn_model1.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=img_shape))
#cnn_model1.add(BatchNormalization())
cnn_model1.add(MaxPooling2D(pool_size=(2, 2)))
cnn_model1.add(Dropout(0.25))

cnn_model1.add(Flatten())

cnn_model1.add(Dense(128, activation='relu'))
#cnn_model1.add(BatchNormalization())

cnn_model1.add(Dense(6, activation='softmax'))
```

In [29]:
```
#Compiling the model 2
cnn_model1.compile(loss=keras.losses.categorical_crossentropy,optimizer=keras.optimizers.Adam(),metrics=['accuracy'])

# Set a learning rate annealer
learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy', patience=3, verbose=1, factor=0.5, min_lr=0.00001)
```

In [31]:
```
#starting time
start_time1= time.time()
#training the model 2
cnn_train_model1 = cnn_model1.fit(X_train, Y_train,batch_size=100,epochs=20,verbose=1,validation_data=(X_val, Y_val),
                                  callbacks=learning_rate_reduction)
```

In [32]:
```
#end time
end_time1 = time.time()
#time taken to train the model on training data
cnn_time1 = end_time1-start_time1
print("Time taken for CNN to implement on training set : {:0.2f} minutes".format(cnn_time1/60.0))
```

```
Time taken for CNN to implement on training set : 18.03 minutes
```

Evaluation of the model 2 on test set:

In [34]:
```
#Evaluating the trained model 2 on test dataset
cnn_model_testresults1 = cnn_model1.evaluate(X_test, Y_test, verbose=0)
cnn_model_testloss1 = cnn_model_testresults1[0]*100
cnn_model_testaccuracy1=cnn_model_testresults1[1]*100
print('Accuracy of our model on test set is equal to '+ str(round(cnn_model_testaccuracy1)) + ' %.')
print('Test loss:', cnn_model_testloss1)
```

```
Accuracy of our model on test set is equal to 91.03 %.
Test loss: 26.799315214157104
```

We have implemented another model #2 that has 2 hidden layers (one convolution layer and1 Max pooling layer) and 1 output layer. We have also tried changing the optimizer to Adam optimizer in this model. Also, a dropout of 25% occurs only after the max pooling layer of pooling window 2x2. The number of filters for the first conv 2D layer is 32 with a kernel size of 3x3. Here also, we have used ReLu as the activation function in hidden layers and SoftMax as the activation function in the output layer. The same number of epochs and batch size are considered for comparison with our finalized model.

From the observations, we can infer that,

- When the number of hidden layers increased (our finalized model), the accuracy has increased slightly compared to the model with 2 hidden layers.
- The accuracy of our model is 91.78% whereas the accuracy of model #2 is 91.0
- As the model has less layers, the time taken to implement the model #2 on train set is 18.03 min

RESNET MODEL IMPLEMENTATION:

In [ ]:
```
#Defining constants
epochs = 8
batch_size = 200
data_augmentation = False
img_size = 28

num_classes = 5
num_filters = 32
num_blocks = 4
num_sub_blocks = 2
use_max_pool = False
input_size = (img_size, img_size,1)
```

```
#Creating model based on ResNet published archietecture
inputs = Input(shape=input_size)
x = Conv2D(num_filters, padding='same',
           kernel_initializer='he_uniform',
           kernel_size=7, strides=2,
           kernel_regularizer=l2(1e-4))(inputs)
x = BatchNormalization()(x)
x = Activation('relu')(x)

#Check by applying max pooling later (setting it false as size of image is small i.e. 28x28)
if use_max_pool:
    x = MaxPooling2D(pool_size=3,padding='same', strides=2)(x)
    num_blocks =3
#Creating Conv base stack

# Instantiate convolutional base (stack of blocks).
for i in range(num_blocks):
    for j in range(num_sub_blocks):
        strides = 1
        is_first_layer_but_not_first_block = j == 0 and i > 0
        if is_first_layer_but_not_first_block:
            strides = 2
        #Creating residual mapping using y
        y = Conv2D(num_filters,
                   kernel_size=3,
                   padding='same',
                   strides=strides,
                   kernel_initializer='he_uniform',
                   kernel_regularizer=l2(1e-4))(x)
        y = BatchNormalization()(y)
        y = Activation('relu')(y)
        y = Conv2D(num_filters,
                   kernel_size=3,
                   padding='same',
                   kernel_initializer='he_uniform',
                   kernel_regularizer=l2(1e-4))(y)
        y = BatchNormalization()(y)
```

```
        if is_first_layer_but_not_first_block:
            x = Conv2D(num_filters,
                       kernel_size=1,
                       padding='same',
                       strides=2,
                       kernel_initializer='he_uniform',
                       kernel_regularizer=l2(1e-4))(x)
        #Adding back residual mapping
        x = keras.layers.add([x, y])
        x = Activation('relu')(x)

    num_filters = 2 * num_filters

# Add classifier on top.
x = AveragePooling2D()(x)
y = Flatten()(x)
outputs = Dense(num_classes,
                activation='softmax',
                kernel_initializer='he_normal')(y)

# Instantiate and compile model.
model = Model(inputs=inputs, outputs=outputs)
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
model.summary()
```

```
scores = model.evaluate(X_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
188/188 [==============================] - 9s 48ms/step - loss: 0.5705 - accuracy: 0.8745
Test loss: 0.5704861879348755
Test accuracy: 0.8744999766349792
```

We see that the Resnet is a complex architecture and as a result the time taken for implementation is around 5-5.5 hours which might increase computational costs compared to the simple CNN. Even though the epochs are less compared to the model which we finalized, the time taken for implementation is more. Also, the accuracy 87.45% is not so better compared to the original model. So, we inferred that the implemented CNN model is better for this dataset than Resnet implementation for better accuracy and performance.

3. You can use any plots to explain the performance of your approach.

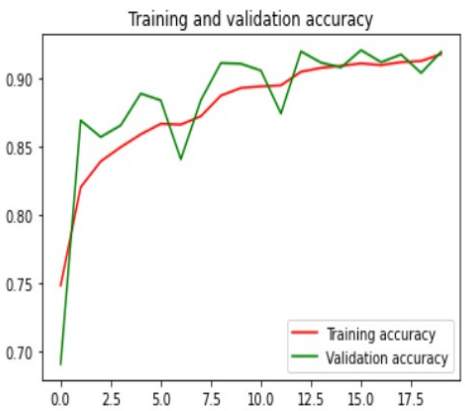
CNN MODEL #1: With 5 hidden layers:

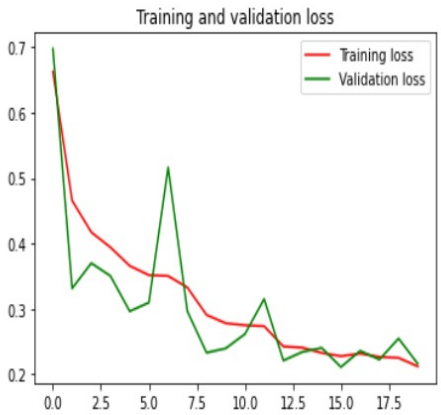
Plotting the validation and training curves for model 1:

In [22]:
```
accuracy = cnn_train_model.history['accuracy']
val_accuracy = cnn_train_model.history['val_accuracy']
loss = cnn_train_model.history['loss']
val_loss = cnn_train_model.history['val_loss']
epochs = range(len(accuracy))

plt.plot(epochs, accuracy, 'r', label='Training accuracy')
plt.plot(epochs, val_accuracy, 'g', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.show()

plt.plot(epochs, loss, 'r', label='Training loss')
plt.plot(epochs, val_loss, 'g', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```



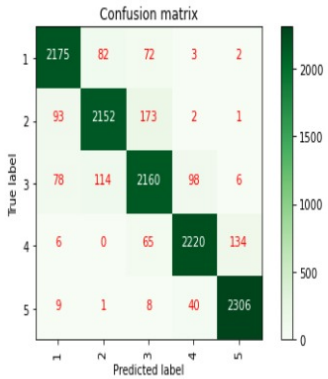


We plotted the training set accuracy and validation set accuracy against the epochs as well as the training and validation losses. We observe that the curves of both training and validation are converging and the model generally achieved a good fit.

CONFUSION MATRIX PLOT:

In [24]:
```
# Predict the values from the validation dataset for model 1
Y_pred = cnn_model.predict(X_test)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred,axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(Y_test,axis = 1)
```

In [26]:
```
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = ['1','2','3','4','5'])
```



We tried plotting the confusion matrix plot for the test data on the trained model. From the confusion matrix, we could see that the model actually performed well as most of the samples have been predicted correctly which are represented in dark green color.

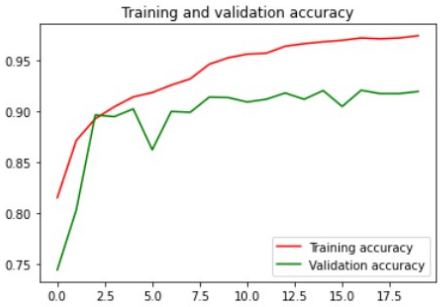
CNN MODEL #2: With 2 hidden layers:

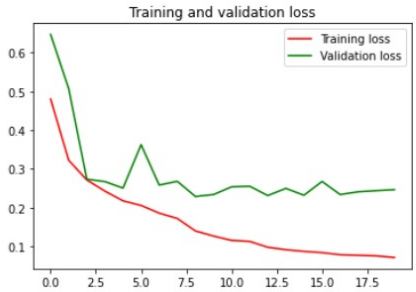
Plotting the validation and training curves for model 2:

In [33]:
```
accuracy1 = cnn_train_model1.history['accuracy']
val_accuracy1 = cnn_train_model1.history['val_accuracy']
loss1 = cnn_train_model1.history['loss']
val_loss1 = cnn_train_model1.history['val_loss']
epochs1 = range(len(accuracy))

plt.plot(epochs1, accuracy1, 'r', label='Training accuracy')
plt.plot(epochs1, val_accuracy1, 'g', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.show()

plt.plot(epochs1, loss1, 'r', label='Training loss')
plt.plot(epochs1, val_loss1, 'g', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```



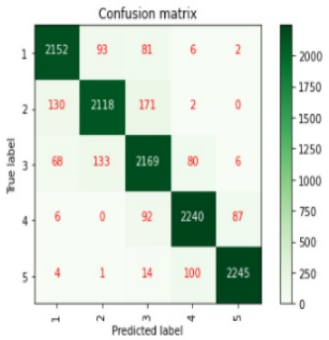


We plotted the training set accuracy and validation set accuracy against the epochs as well as the training and validation losses. We observe that the curves of both training and validation are not converging and are varying a little too much with each other, and hence the model could be underfitting.

CONFUSION MATRIX PLOT:

Confusion Matrix of model 2:

In [37]:
```
# compute the confusion matrix for model 2:
confusion_mtx1 = confusion_matrix(Y_true1, Y_pred_classes1)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx1, classes = ['1','2','3','4','5'])
```



We tried plotting the confusion matrix plot for the test data on the trained model #2. From the confusion matrix, we could see that the model actually performed good but not better than model #1 as most of the many of the samples have been predicted wrongly compared to model #1.

4. Evaluate your code with other metrics on the training data:

Metrics(Precision,Recall, F-Score) for model 1:

```
In [27]: #classification Report for model 1
         targets = ['1','2','3','4','5']
         print(classification_report(Y_true, Y_pred_classes, target_names = targets))
```

```
              precision    recall  f1-score   support

           1       0.92      0.93      0.93      2334
           2       0.92      0.89      0.90      2421
           3       0.87      0.88      0.88      2456
           4       0.94      0.92      0.93      2425
           5       0.94      0.98      0.96      2364

    accuracy                           0.92     12000
   macro avg       0.92      0.92      0.92     12000
weighted avg       0.92      0.92      0.92     12000
```

From the above classification matrix, we can say that the model #1 is well fitted however it might be little underfitted for the label 3 compared to other labels.

Metrics(Precision,Recall, F-Score) for model 2:

```
In [38]: #classification Report for model 2
         targets = ['1','2','3','4','5']
         print(classification_report(Y_true1, Y_pred_classes1, target_names = targets))
```

```
              precision    recall  f1-score   support

           1       0.91      0.92      0.92      2334
           2       0.90      0.87      0.89      2421
           3       0.86      0.88      0.87      2456
           4       0.92      0.92      0.92      2425
           5       0.96      0.95      0.95      2364

    accuracy                           0.91     12000
   macro avg       0.91      0.91      0.91     12000
weighted avg       0.91      0.91      0.91     12000
```

From the above classification matrix, we can say that the model #2 is well fitted good however it might be little underfitted for the label 3 compared to other labels.

But, when both the models are compared, we see that model #1 is fit good for the given data and model #2 is slightly underfit in comparison.

↑ Back to top