Assignment 3 Dataset 2

August 16, 2021

1 CM[6]:

NOTE: In this solution, Parts refer to the points needs to be covered in CM[6] as per the assignment instructions, where, Part 1 is designing the model , Part 2 is designing another variants for classification, Part 3 refers to details of model implementation and Part 4 refers to the Code implementation.

[1]:
```
# Importing neccessary liberaries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import keras.backend as K
import tensorflow as tf
import keras
import pydot
import graphviz
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras import layers
from keras.layers import Input, GlobalAveragePooling2D, Add, Dense, Activation,␣
 ↪ZeroPadding2D, BatchNormalization,add, Flatten, Conv2D,␣
 ↪AveragePooling2D,MaxPooling2D, Dropout
from keras.models import Model, Sequential
from keras.initializers import glorot_uniform
from tensorflow.keras.callbacks import TensorBoard
from tensorflow.keras.optimizers import Adam
from keras.layers.merge import concatenate
from tensorflow.keras import regularizers
import warnings
warnings.filterwarnings('ignore')
```

1.0.1 Retreiving the data:

Fashion MNIST is an Image dataset composed of greyscale images of clothing items such as footwear , shirts, handbags etc., with small 28 X 28 pixels. Here, there are 5 labels provided in target column from 0-4. The retreived dataset consists of 60000 values with 744 columns in training set and 10000 values with 744 image pixels in test set.

[2]:
```
# Retreiving train data
trainX = pd.read_csv('../input/fashion-mnist/trainX.csv')
trainy = pd.read_csv('../input/fashion-mnist/trainy.csv')

# Retreiving test data
testX = pd.read_csv('../input/fashion-mnist/testX.csv')
testy = pd.read_csv('../input/fashion-mnist/testy.csv')
```

1.0.2 Preparing data:

Here, we will drop unneccesary columns like 'Id' from train and test set and prepare it to feed to CNN.

[3]:
```
x_train = trainX.drop(columns='Id')
x_test = testX.drop(columns='Id')
y_train = trainy.drop(columns='Id')
y_test = testy.drop(columns='Id')
```

Reshaping the data: In order to input image data for classification to convolutional neural network layers, their feature dimension will be reshaped into (no. of rows/images, height, width, channel) format. As Fashion MNIST dataset consists of grayscale images, so, channel will be 1 and among 784 features 28 X 28 will be our height and width . Therefore,for training set , they will be reshaped into (60000, 28, 28, 1) and similarly, for test set they will be reshaped into (10000, 28, 28, 1).

[4]:
```
# reshape(examples, height, width, channels)
x_train = x_train.values.reshape((-1, 28, 28, 1))
x_test = x_test.values.reshape((-1, 28, 28, 1))
```

Normalizing the data: Normalizing the data is crucial part as otherwise, the model will give huge loss at some points and weird accuracy, in simple, it will not give good performanec and does not train properly. Also, as each image pixel range lies between 0 to 255. so, to normalize, it will be divided by 255 to put it in range between [0,1] for image classification.

[5]:
```
x_train = x_train.astype("float32")/255
x_test = x_test.astype("float32")/255
```

1.0.3 Visualizing the Image Dataset and its nature description:

From below implementation of train set and test set images , it is visible that data is not classified according to clothes, footwear or handbags(as per original data), rather , it has been classified depending upon some other hidden pattern and generated labels by neural networks. So, there will be chances of getting less accuracy , more loss and overfitting. Therefore, in this dataset our goal will to aquire good enough accuracy and loss without much overfitting. we can see from below images that category 1 (or 0 in dataset) consists of clothes, footwears and handbags and similarly does, the all other categories. So, our network will train and learn that hidden pattern or feature which classifies them. In the last plot, we even checked the balancing of taget data/labels which turns out to be good.

[6]:
```
# Function that returns the images with their respective category
def plot_sample_images(x,y,len_img,cmap="Blues"):
    # Plot the sample images now
    f, ax = plt.subplots(5,8, figsize=(20,12))

    for i in range(200):
        if y.iloc[i].values == 0:
            ax[0, i%8].imshow(x.reshape((len_img,28,28))[i], cmap=cmap)
            ax[0, i%8].axis('off')
            ax[0, i%8].set_title('Category 1',fontweight='bold',fontsize=14)

        if y.iloc[i].values == 1:
            ax[1, i%8].imshow(x.reshape((len_img,28,28))[i], cmap=cmap)
            ax[1, i%8].axis('off')
            ax[1, i%8].set_title('Category 2',fontweight='bold',fontsize=14)
        if y.iloc[i].values == 2:
            ax[2, i%8].imshow(x.reshape((len_img,28,28))[i], cmap=cmap)
            ax[2, i%8].axis('off')
            ax[2, i%8].set_title('Category 3',fontweight='bold',fontsize=14)
        if y.iloc[i].values == 3:
            ax[3, i%8].imshow(x.reshape((len_img,28,28))[i], cmap=cmap)
            ax[3, i%8].axis('off')
            ax[3, i%8].set_title('Category 4',fontweight='bold',fontsize=14)
        if y.iloc[i].values == 4:
            ax[4, i%8].imshow(x.reshape((len_img,28,28))[i], cmap=cmap)
            ax[4, i%8].axis('off')
            ax[4, i%8].set_title('Category 5',fontweight='bold',fontsize=14)
    plt.show()
```

Train data Images:

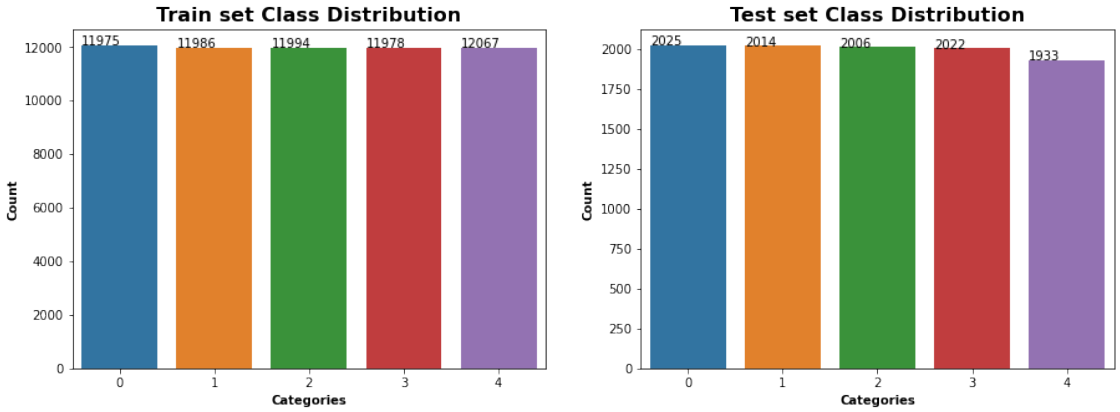
[7]:
```
train_img= plot_sample_images(x_train,y_train,60000,'Greens')
```

Test data Images:

[8]:
```
test_img= plot_sample_images(x_test,y_test,10000)
```

Categorical distribution of Target Classes:

[9]:
```
# Get the count for each train label
label_counts_train = y_train.value_counts()
# Get the count for each test label
label_counts_test = y_test.value_counts()

# Plotting Categorical distribution of target
fig, axes= plt.subplots(1,2,figsize=(15,5))
tr =sns.barplot(np.arange(0,5),label_counts_train, ax= axes[0] )
for p,i in zip(tr.patches, np.arange(0,5)):
        tr.annotate(label_counts_train[i], (p.get_x(), p.get_height()+0.1))
tt =sns.barplot(np.arange(0,5),label_counts_test,ax= axes[1] )
for p,i in zip(tt.patches, np.arange(0,5)):
        tt.annotate(label_counts_test[i], (p.get_x(), p.get_height()+0.1))
axes[0].set_xlabel('Categories',fontweight='bold')
axes[0].set_ylabel('Count',fontweight='bold')
axes[1].set_xlabel('Categories',fontweight='bold')
axes[1].set_ylabel('Count',fontweight='bold')
axes[0].set_title('Train set Class Distribution',fontsize=16,fontweight='bold')
axes[1].set_title('Test set Class Distribution',fontsize=16,fontweight='bold')
plt.show()
```



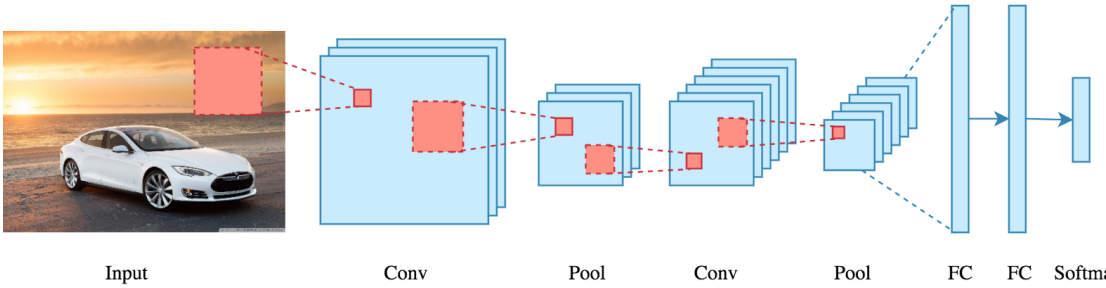
2 Deep Convolutional network:

2.1 CM[6]

2.2 Part 1 & 3 - 1st Model(Deep CNN Model)

Our first implementation will be designing simple deep convolutional neural network. ##### Why Deep CNN: Deep Convolutional neural network is a combination of various convolutional and pooling layers especially, designeded for image classification with large dataset. They provide better performance than simple atificial neural network and classifies well. It can work with n dimensional data. Also, they are able to learn basic filters automatically and combine them

hierarchically for pattern recognition. However, many deep CNNs suffers from overfitting and takes huge time in processing because of large and complex computation. ##### How it works: Convolutional neural networks are composed of multiple layers of artificial neurons(mathematical functions that calculate the weighted sum of multiple inputs and outputs with an activation value) .Once image is input into model, each layer generates several activation functions that will be carried on to the next layer, where the first layer usually extracts edges. AS we dive deeper, it detects more complex features such as objects, faces, etc. In these networks, a filter is applied to a input in order to create a feature map which detects features presence in the input. The whole deep Convolutional network consists of various layers of convolution and pooling, which are then flattened and fed to fully connected dense layers. Its architecture is shown below:



Steps to implement Deep CNN Model: 1) A sequenrial API from keras is called to create layer to layer model .

2) The first Conv2D layer is applied with input which has 32 filters of shape (3,3), 'same' padding and 'relu' activation function .
3) Then a Dropout layer is added with dropout of 0.25.
4) Now, second Conv2D layer is applied which has 64 filters of shape (5,5), 'same' padding and 'relu' activation function .
5) Then, the Max Pooling layer having pool_size of 2 is applied.
6) Then, another Dropout layer is added with dropout of 0.35.
7) Now, third Conv2D layer is applied which has 128 filters of shape (7,7), 'valid' padding and 'relu' activation function .
8) Then, the Max Pooling layer having pool_size of 2 is applied.
9) Then, another Dropout layer is added with dropout of 0.4.
10) Now, Flatten layer will applied to reshape the output from Convolutional layers and then input them to fully connected dense layers.
11) Now, a Dense layer with 600 units will be applied with activation function as 'relu'.
12) This is then followed by Dropout layer of dropout 0.5.
13) Again one more Dense layer is applied with 300 units and activation function as 'relu' .
14) This is then followed by Dropout layer of dropout 0.5.
15) The Fully Connected (Dense) layer reduces its input to the number of classes(i.e.,5)using a softmax activation .

This is how a model is designed.

Compiling , summarizing and plotting the architecture of model: Finally, a model is created and compiled with - optimizer = 'adam', it merge the best characteristics of the AdaGrad and RMSProp algorithms to provide an optimization algorithm .Thus, it can handle sparse gradients on noisy problems and relatively easy to configure. - loss = 'sparse_cross_entropy' as our dataset is image and sparse . Also, it a classification problem where there are more than 2 categories. - and metrics = 'accuracy', to get the performance of model and know how efficient model is.

Once, it is compiled, its summary along with a visual plot of its architecture will be displayed.

Training the model: Now, to train the model, we will fit the parameters into model as: - train feature/input = x_train - train target = y_train - Validation_data = (x_test, y_test) . here, test set is fed as validation data to show runtime performance and plot of testing data, as per the assignment requirement. Else, one can do validation even without test set by inputting size of validation or retreiving the validation set by splitting training into train and validation. But, here as per requirement of Runtime performance and accuracy-loss plots of testing set, we will use testing data as validation data. - Epochs = 30 , in order to avoid or minimize overfitting by early stopping. - Batch_size = 128, to get fast computational results. - verbose = 1

2.3 CM[6]

2.4 Part 4 - 1st Model(Deep CNN Model)

2.4.1 Implementing Deep CNN Model:

[10]:
```
# Building cnn_layers_model with 4 layers
cnn_layers_model=Sequential()

# 1 Convolutional layer
cnn_layers_model.add(Conv2D(filters=32,kernel_size=3,activation='relu',
                            padding='same',input_shape=(x_train.
 ↪shape[1],x_train.shape[2],1)))
cnn_layers_model.add(Dropout(0.25))

# 2 Convolutional + Pooling layer
cnn_layers_model.
 ↪add(Conv2D(filters=64,kernel_size=5,padding='same',activation='relu'))
cnn_layers_model.add(MaxPooling2D(pool_size=2))
cnn_layers_model.add(Dropout(0.35))

# # 3 Convolutional + Pooling layer
cnn_layers_model.add(Conv2D(filters=128,kernel_size=7,activation='relu'))
cnn_layers_model.add(MaxPooling2D(pool_size=2))
cnn_layers_model.add(Dropout(0.4))

# Flatten layer
cnn_layers_model.add(Flatten())
```

```
# Dense Layers
cnn_layers_model.add(Dense(600,activation='relu'))
cnn_layers_model.add(Dropout(0.5))
cnn_layers_model.add(Dense(300,activation='relu'))
cnn_layers_model.add(Dropout(0.5))
cnn_layers_model.add(Dense(5,activation='softmax'))

# Compiling the model
cnn_layers_model.
 ↪compile(loss='sparse_categorical_crossentropy',optimizer='adam',␣
 ↪metrics=['accuracy'])

# Summarizing the model
cnn_layers_model.summary()
```

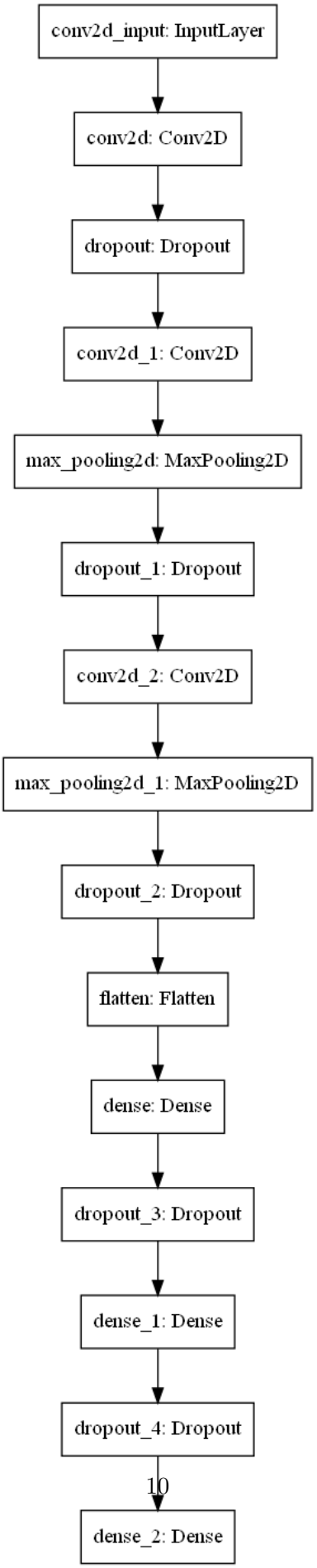
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
dropout (Dropout)            (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 64)        51264
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 128)         401536
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 600)               1229400
_________________________________________________________________
dropout_3 (Dropout)          (None, 600)               0
_________________________________________________________________
dense_1 (Dense)              (None, 300)               180300
_________________________________________________________________
dropout_4 (Dropout)          (None, 300)               0
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 1505
=================================================================
```

```
Total params: 1,864,325
Trainable params: 1,864,325
Non-trainable params: 0
_________________________________________________________________
```

2.4.2 Plotting Deep CNN Model Architecture:

[11]:
```
# plot model architecture
plot_model(cnn_layers_model, to_file='cnn_layers_model.png')
```

[11]:

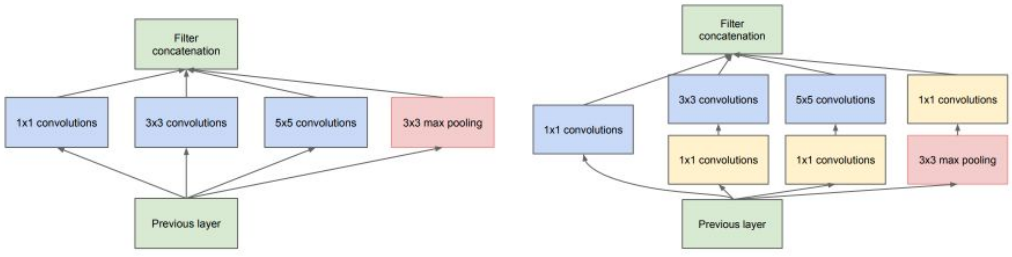

conv2d_input: InputLayer
conv2d: Conv2D
dropout: Dropout
conv2d_1: Conv2D
max_pooling2d: MaxPooling2D
dropout_1: Dropout
conv2d_2: Conv2D
max_pooling2d_1: MaxPooling2D
dropout_2: Dropout
flatten: Flatten
dense: Dense
dropout_3: Dropout
dense_1: Dense
dropout_4: Dropout
dense_2: Dense

3 CNN With Inception:

3.1 CM[6]

3.2 Part 2 & 3 - 2nd Model(CNN With Inception)

Why CNN With Inception: In 1st model, it is visible that as we go deeper or add more layers to a Neural Network . To avoid overfitting, we added dropout layers which decreased the training accuracy from around 97-98 to 92-93 and similarly for loss, we reached minimum at 0.1, and its not decreasing anymore. So, we need to built a model providing good performance . Therefore, Inception model is considered one of the good choice as it attained more than 78.1% of accuracy on ImageNet dataset and various other Image datasets. Also, in this not only depth of the model will be increased but also, the width to boost the performance of network. Furthermore, the upcoming explaination with designing will show more of its characteristics. #### Designing Inception Model: In Convolutional Neural Networks (CNNs), the important job is to choose right layers with right filters, to find the optimal solution. The inception module is created with Conv2D and pooling layers (1x1 filter, 3x3 filter, 5x5 filter or max-pooling) , which were concatenated later . After this, these 'Inception modules' are stacked upon each other. Thus, their output correlation statistics will vary, as higher layers captures features of higher abstraction and their spatial concentration is expected to decrease suggesting that the ratio of 3×3 and 5×5 convolutions should increase as we move to higher layers. Therefore, their computational cost will also increase that is why in second figure, dimension reduction technique is used through 1X1 convolutions. GoogleNet is one of the famous Inception-based algorithm. Below shown are the two verions of Inception model, one is naive version without and dumensionatlity reduction and other, comes with the property of dimensionality reduction by adding layer of 1X1 layers. We have implemented the inception model with dimensionality reduc-



(a) Inception module, naïve version (b) Inception module with dimension reductions

tion.

Inception Module: Steps to implement the Inception Module:

Filter 1 - 1X1 Convolutional layer: - The first CONV2D has f1 filters of shape (1,1) with 'same' padding, 'relu' activation function and l1_l2 kernel_regularizer is applied.

Filter 2 - 3X3 Convolutional layers: - The CONV2D has f2_in filters of shape (1,1) with 'same' padding, 'relu' activation function and l1_l2 kernel_regularizer is applied. - The CONV2D has f2_out filters of shape (3,3) with 'same' padding, 'relu' activation function and l1_l2 kernel_regularizer is applied.

Filter 3 - 5X5 Convolutional layers - The CONV2D has f3_in filters of shape (1,1) with 'same' padding, 'relu' activation function and l1_l2 kernel_regularizer is applied. - The CONV2D has f3_out filters of shape (5,5) with 'same' padding, 'relu' activation function and l1_l2 kernel_regularizer is applied.

Filter 4 - 3X3 Max Pooling layer - The Max Pooling uses a window of shape (3,3) with stride of size (2,2) and 'same' padding. - The CONV2D has f4_out filters of shape (1,1) with 'same' padding, 'relu' activation function and l1_l2 kernel_regularizer is applied. Concatenate all filters: - The four filetrs achieved namely, conv1, conv2, conv3 and pool will be concatenated.

Here, L1_L2 regularizers makes the model to generalize better , improves the model's performance on the unseen data as well and thus, avoid overfitting. ##### Building the Inception Model: As, we have achieved 7 layers of the inception module, they will be concatenated and then, stacked together to make one inception model. In this model, aound 23 layers will be implemented including two inception modules.

Steps to implement our Inception Model with CNN are: 1) Input is provided with shape of (28, 28, 1).

2) inception_module function will be called to apply inception layer with filters (64, 96, 128, 16, 32, 32).

3) BatchNirmalization layer will be applied as, its a deep model so, it will reduce and generalization error during training and boost up the speed.

4) another inception layer will b e applied with filters (128, 128, 192, 32, 96, 64).

5) Now, Flatten layer will applied to reshape the output from Convolutional layers and then input them to fully connected dense layers.

6) Now, a Dense layer with 64 units will be applied with activation function as 'relu' and l1_l2 regularizer to handle overfitting.

7) This is then followed by Dropout layer.

8) Again one more Dense layer is applied with 128 units and activation function as 'relu' and l1_l2 kernel regularizers.

9) This is then followed by Dropout layer.

10) The Fully Connected (Dense) layer reduces its input to the number of classes(i.e.,5)using a softmax activation and l1_l2 kernel regularizers.

11) The Model is created and trained an with given inputs and predicted output from above layers.

Compiling , summarizing and plotting the architecture of model: Finally, a model is created and compiled with - optimizer = 'adam', it merge the best characteristics of the AdaGrad and RMSProp algorithms to provide an optimization algorithm .Thus, it can handle sparse gradients on noisy problems and relatively easy to configure. - loss = 'sparse_cross_entropy' as our dataset is image and sparse . Also, it a classification problem where there are more than 2 categories. - and metrics = 'accuracy', to get the performance of model and know how efficient model is.

Once, it is compiled, its summary along with a visual plot of its architecture will be displayed.

Training the model: Now, to train the model, we will fit the parameters into model as: - train feature/input = x_train - train target = y_train - Validation_data = (x_test, y_test) . here, test set is fed as validation data to show runtime performance and plot of testing data, as per the assignment requirement. Else, one can do validation even without test set by inputting size of validation or retreiving the validation set by splitting training into train and validation. But, here as per requirement of Runtime performance and accuracy-loss plots of testing set, we will use testing data as validation data. - Epochs = 30 , in order to avoid or minimize overfitting by early stopping. - Batch_size = 128, to get fast computational results. - verbose = 1

3.3 CM[6]

3.4 Part 4 - 2nd Model(CNN With Inception)

3.4.1 Implementing CNN With Inception Model:

[17]:
```
# function for creating a projected inception module
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu',
                   kernel_regularizer=regularizers.l1_l2(l1=1e-5,␣
 ↪l2=1e-4))(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu'
                   ,kernel_regularizer=regularizers.l1_l2(l1=1e-5,␣
 ↪l2=1e-4))(layer_in)
    conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu'
                   ,kernel_regularizer=regularizers.l1_l2(l1=1e-5,␣
 ↪l2=1e-4))(conv3)
    # 5x5 conv
    conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu'
                   ,kernel_regularizer=regularizers.l1_l2(l1=1e-5,␣
 ↪l2=1e-4))(layer_in)
    conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu'
                   ,kernel_regularizer=regularizers.l1_l2(l1=1e-5,␣
 ↪l2=1e-4))(conv5)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    pool = Conv2D(f4_out, (1,1), padding='same', activation='relu'
                  ,kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(pool)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out

# define model input
visible = Input(shape=(28, 28, 1))

# add inception layer 1
layer = inception_module(visible, 64, 96, 128, 16, 32, 32)
```

```
layer= BatchNormalization()(layer)

# add inception layer 2
layer = inception_module(layer, 128, 128, 192, 32, 96, 64)

# Flatten layer
layer = Flatten()(layer)

# Dense layers
layer = Dense(64, activation='relu',kernel_regularizer=regularizers.
 ↪l1_l2(l1=1e-5, l2=1e-4))(layer)
layer = Dropout(0.3)(layer)
layer = Dense(128, activation='relu',kernel_regularizer=regularizers.
 ↪l1_l2(l1=1e-5, l2=1e-4))(layer)
layer = Dropout(0.3)(layer)
output_layer = Dense(5,activation='softmax',kernel_regularizer=regularizers.
 ↪l1_l2(l1=1e-5, l2=1e-4))(layer)

# create model
cnn_inception_model = Model([visible], [output_layer])

# Compile model
cnn_inception_model.compile(loss = 'sparse_categorical_crossentropy', optimizer␣
 ↪= 'adam',
                            metrics = ['accuracy'])

# summarize model
cnn_inception_model.summary()
```

```
Model: "model"
__________________________________________________________________________________
__________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
==================
input_1 (InputLayer)            [(None, 28, 28, 1)]  0
__________________________________________________________________________________
__________________
conv2d_4 (Conv2D)               (None, 28, 28, 96)   192         input_1[0][0]
__________________________________________________________________________________
__________________
conv2d_6 (Conv2D)               (None, 28, 28, 16)   32          input_1[0][0]
__________________________________________________________________________________
__________________
max_pooling2d_2 (MaxPooling2D)  (None, 28, 28, 1)    0           input_1[0][0]
__________________________________________________________________________________
__________________
```

```
conv2d_3 (Conv2D)               (None, 28, 28, 64)   128         input_1[0][0]
__________________________________________________________________________________
__________________
conv2d_5 (Conv2D)               (None, 28, 28, 128)  110720      conv2d_4[0][0]
__________________________________________________________________________________
__________________
conv2d_7 (Conv2D)               (None, 28, 28, 32)   12832       conv2d_6[0][0]
__________________________________________________________________________________
__________________
conv2d_8 (Conv2D)               (None, 28, 28, 32)   64
max_pooling2d_2[0][0]
__________________________________________________________________________________
__________________
concatenate (Concatenate)       (None, 28, 28, 256)  0           conv2d_3[0][0]
                                                                 conv2d_5[0][0]
                                                                 conv2d_7[0][0]
                                                                 conv2d_8[0][0]
__________________________________________________________________________________
__________________
batch_normalization (BatchNorma (None, 28, 28, 256)  1024
concatenate[0][0]
__________________________________________________________________________________
__________________
conv2d_10 (Conv2D)              (None, 28, 28, 128)  32896
batch_normalization[0][0]
__________________________________________________________________________________
__________________
conv2d_12 (Conv2D)              (None, 28, 28, 32)   8224
batch_normalization[0][0]
__________________________________________________________________________________
__________________
max_pooling2d_3 (MaxPooling2D)  (None, 28, 28, 256)  0
batch_normalization[0][0]
__________________________________________________________________________________
__________________
conv2d_9 (Conv2D)               (None, 28, 28, 128)  32896
batch_normalization[0][0]
__________________________________________________________________________________
__________________
conv2d_11 (Conv2D)              (None, 28, 28, 192)  221376      conv2d_10[0][0]
__________________________________________________________________________________
__________________
conv2d_13 (Conv2D)              (None, 28, 28, 96)   76896       conv2d_12[0][0]
__________________________________________________________________________________
__________________
conv2d_14 (Conv2D)              (None, 28, 28, 64)   16448
max_pooling2d_3[0][0]
__________________________________________________________________________________
```

```
------------------
concatenate_1 (Concatenate)     (None, 28, 28, 480)  0           conv2d_9[0][0]
                                                                 conv2d_11[0][0]
                                                                 conv2d_13[0][0]
                                                                 conv2d_14[0][0]

------------------------------------------------------------------------------
------------------
flatten_1 (Flatten)             (None, 376320)       0
concatenate_1[0][0]

------------------------------------------------------------------------------
------------------
dense_3 (Dense)                 (None, 64)           24084544    flatten_1[0][0]

------------------------------------------------------------------------------
------------------
dropout_5 (Dropout)             (None, 64)           0           dense_3[0][0]

------------------------------------------------------------------------------
------------------
dense_4 (Dense)                 (None, 128)          8320        dropout_5[0][0]

------------------------------------------------------------------------------
------------------
dropout_6 (Dropout)             (None, 128)          0           dense_4[0][0]

------------------------------------------------------------------------------
------------------
dense_5 (Dense)                 (None, 5)            645         dropout_6[0][0]
==============================================================================
==================
Total params: 24,607,237
Trainable params: 24,606,725
Non-trainable params: 512

------------------------------------------------------------------------------
------------------
```

3.4.2 Plotting CNN Inception Model Architecture:

[18]:
```
# plot model architecture
plot_model(cnn_inception_model, show_shapes=True, to_file='inception_module.
 ↪png')
```

[18]:

4 CNN With Resnet50

4.1 CM[6]:

4.2 Part 2 & 3 - 3rd Model(Resnet50)

4.2.1 Why Resnet50:

From previous observation, it is visible that as we go deeper or add more layers to a Neural Network , it can make model more robust for image-related tasks, but also leads to loosing accuracy.This may be due to overfitting, but in that case,that can be solved using dropout and regularization techniques.So, another possible reason for bad performance can be vanishing gradient problem. That's why we are going to implement a deep CNN with new approach known as Resnet50, to avoid that problem (if it so).Also, with network accuracy will be enhanced compared to others.

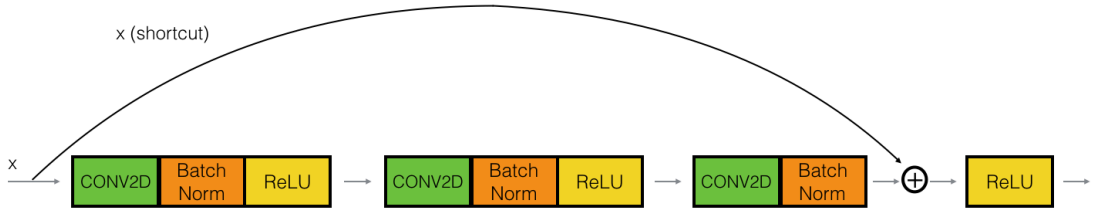
4.2.2 Designing the CNN Resnet50 Model:

Resnet50 is a deep CNN with 50 layers consisting of convolutional and identity blocks and then combined ia form to give good results. The residual units or blocks which have skip connections,

are called identity connections in resnet which allows the gradient to be directly backpropagated to earlier layers.

The left image shows mainpath whereas the other one added shortcut to the main path which makes convinient for blocks to learn the identity function. So, these blocks wilk be stacked upon each other in order to get Resnet50 deep neural network, but there will be chances of harming the tarining performance . Further, two main blocks of resnet will be implemented, namely identity and convolutional blocks.

Identity Block : These are the standard blocks and are suitable for the cases where input and output activation functions have same dimensions.The below diagram reperesents the Identity block with both shortcut and main path and, Conv2D , ReLu and BacthNormalization layers.



The steps followed to implement Identity block are as follows:

First component of main path: - The first CONV2D has F1 filters of shape (1,1) and a stride of (1,1) with 'valid' padding and kerenel initializer as glorot_uniform with random seed =0 .This layer will be named as conv_name_base + '2a'. - The first BatchNorm is normalizing the channels axis = 3 with name as bn_name_base + '2a'. - Then, ReLU activation function is applied without any name or hyperparameters.

Second component of main path: - The second CONV2D has F2 filters of shape (f,f) and a stride of (1,1) with 'same' padding and kerenel initializer as glorot_uniform with random seed =0 .This layer will be named as conv_name_base + '2b'. - The second BatchNorm is normalizing the channels axis = 3 with name as bn_name_base + '2b'. - Then, ReLU activation function is applied without any name or hyperparameters.

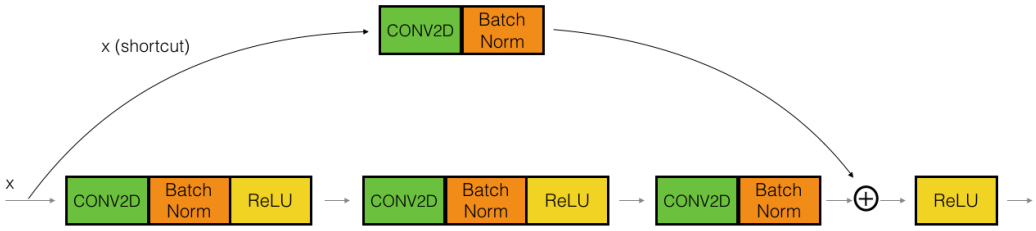
Third component of main path:

- The third CONV2D has F3 filters of shape (1,1) and a stride of (1,1) with 'valid' padding and kerenel initializer as glorot_uniform with random seed =0 .This layer will be named as conv_name_base + '2c'.

- The third BatchNorm is normalizing the channels axis = 3 with name as bn_name_base + '2c'.

Final step:

- The shortcut and the input are added together.
- Then, ReLU activation function is applied without any name or hyperparameters. ##### The convolutional block Now, the other block will be implemented where Con2D layer is added even to shortcut to resize the input x to a different dimension and Thus, unlike identity block, here input and output activation functions can have different dimensions because with this Conv layer, the dimensions will match up in the final addition through shortcut. Below is the figure of representation of convolutional network.



The Steps of the convolutional block are as follows.

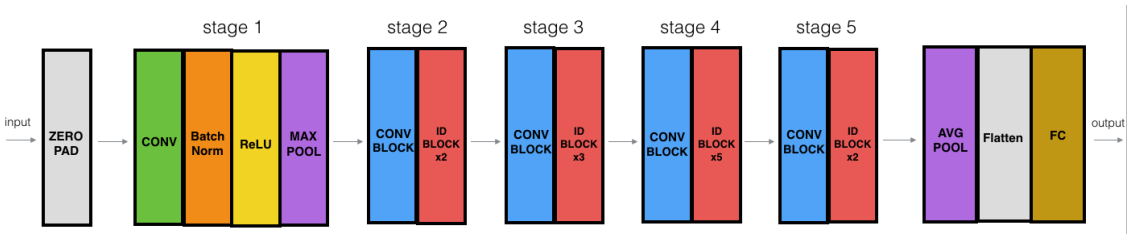
First component of main path: - The first CONV2D has F1 filters of shape (1,1) and a stride of (s,s) with 'valid' padding and name as conv_name_base + '2a'. - The first BatchNorm is normalizing the channels axis = 3 with name as bn_name_base + '2a'. - Then, ReLU activation function is applied without any name or hyperparameters.

Second component of main path: - The second CONV2D has F2 filters of (f,f) and a stride of (1,1) with 'same' padding and name as conv_name_base + '2b'. - The second BatchNorm is normalizing the channels axis = 3 with name as bn_name_base + '2b'. - Then, ReLU activation function is applied without any name or hyperparameters.

Third component of main path: - The third CONV2D has F3 filters of (1,1) and a stride of (1,1) with 'valid' padding and name as conv_name_base + '2c'. - The third BatchNorm is normalizing the channels axis = 3 with name as bn_name_base + '2c', without any ReLU activation function.

Shortcut path: - The CONV2D has F3 filters of shape (1,1) and a stride of (s,s) with 'valid' padding and name as conv_name_base + '1'. - The BatchNorm is normalizing the channels axis = 3 with name as bn_name_base + '1'.

Final step: - The shortcut and the main path values are added together. - Then, ReLU activation function is applied without any name or hyperparameters. ##### Building your first ResNet model (50 layers) Once, identity and convolution blocks are achieved , the next step is put them together to form structure of resnet50. Below is the representation of Resnet50 model with both blocks



stacked.

The steps of this ResNet-50 model are: 1) Input is provided with shape of(28, 28, 1)

2) Zero-padding with pad= (3,3)

3) Stage 1:

- The 2D Convolution has 64 filters of shape (7,7) with stride of (2,2) and name as “conv1”.
- BatchNorm is applied to the channels axis = 3.
- MaxPooling uses a (3,3) window with a (2,2) stride.

4) Stage 2:

- The convolutional block uses three set of filters of size [64,64,256], where “f” is 3, “s” is 1 and the block is “a”.
- The 2 identity blocks use three set of filters of size [64,64,256], where “f” is 3 and the blocks are “b” and “c”.

5) Stage 3:

- The convolutional block uses three set of filters of size [128,128,512], where “f” is 3, “s” is 2 and the block is “a”.
- The 3 identity blocks use three set of filters of size [128,128,512], where “f” is 3 and the blocks are “b”, “c” and “d”.

6) Stage 4:

- The convolutional block uses three set of filters of size [256, 256, 1024], where “f” is 3, “s” is 2 and the block is “a”.
- The 5 identity blocks use three set of filters of size [256, 256, 1024], where “f” is 3 and the blocks are “b”, “c”, “d”, “e” and “f”.

7) Stage 5:

- The convolutional block uses three set of filters of size [512, 512, 2048], where “f” is 3, “s” is 2 and the block is “a”.
- The 2 identity blocks use three set of filters of size [512, 512, 2048], where “f” is 3 and the blocks are “b” and “c”.

8) The 2D Average Pooling uses a window of shape (1,1) as the input has height and width of 28 X 28 ,and with commonly used (2X2) shape , error will arise . Its name is “avg_pool”.

9) Flatten layer with 1 BatchNormalization layer.

- The flatten doesn’t have any hyperparameters or name, and is used to flatten the dimensions for applying futher dense layers.
- Then, a BatchNormalization layer will be applied to normalize, reduce generalization error and boost up the training speed.

10) 3 Dense layers with 3 BatchNormalization and 3 Dropout layers
    - After this a Dense layers is applied with 256 units and ReLu activation function.
    - It is then followed by Dropout layer with dropout of 0.5 to reduce overfitting.
    - Again , a BatchNormalization layer will be applied.
    - Another Dense layer is applied with 128 units and ReLu activation function.
    - It is then followed by Dropout layer with dropout of 0.5 to reduce overfitting.
    - Again , a BatchNormalization layer will be applied.
    - One more, Dense layer is applied with 64 units and ReLu activation function.
    - It is then followed by Dropout layer with dropout of 0.5 to reduce overfitting.
    - Again , a BatchNormalization layer will be applied.
11) The Fully Connected (Dense) layer reduces its input to the number of classes(i.e.,5)using a softmax activation with name should be 'fc' + str(classes).
12) The Model is created and trained an with given inputs and predicted output from above layers.

Compiling , summarizing and plotting the architecture of model: Finally, a model is created and compiled with - optimizer = 'adam', it merge the best characteristics of the AdaGrad and RMSProp algorithms to provide an optimization algorithm .Thus, it can handle sparse gradients on noisy problems and relatively easy to configure. - loss = 'sparse_cross_entropy' as our dataset is image and sparse . Also, it a classification problem where there are more than 2 categories. - and metrics = 'accuracy', to get the performance of model and know how efficient model is.

Once, it is compiled, its summary along with a visual plot of its architecture will be displayed.

Training the model: Now, to train the model, we will fit the parameters into model as: - train feature/input = x_train - train target = y_train - Validation_data = (x_test, y_test) . here, test set is fed as validation data to show runtime performance and plot of testing data, as per the assignment requirement. Else, one can do validation even without test set by inputting size of validation or retreiving the validation set by splitting training into train and validation. But, here as per requirement of Runtime performance and accuracy-loss plots of testing set, we will use testing data as validation data. - Epochs = 30 , in order to avoid or minimize overfitting by early stopping. - Batch_size = 128, to get fast computational results. - verbose = 1

4.3 CM[6]

4.4 Part 4 - 3rd Model(Resnet50)

4.4.1 Implementation of the CNN Resnet50 Model:

[24]:
```
# Function that returns Identity blocks of Resnet50
def identity_block(x,f,filters,stage,block):
    # defining name
    conv_name_base= 'res' + str(stage) + block + '_branch'
```

```
    bn_name = 'bn' + str(stage) + block + '_branch'

    # Retreive filters
    F1, F2, F3 = filters

    # saving input value for later need pf adding back to the main path
    x_shortcut = x

    # first component of main path
    x = Conv2D(filters = F1, kernel_size = 1, strides = 1, padding = 'valid',␣
↪name = conv_name_base + '2a',
               kernel_initializer =glorot_uniform(seed=0))(x)
    x = BatchNormalization(axis= 3 , name= bn_name + '2a')(x)
    x = Activation('relu')(x)

    # second component of main path
    x = Conv2D(filters = F2, kernel_size = f, strides = 1, padding = 'same',␣
↪name = conv_name_base + '2b',
               kernel_initializer =glorot_uniform(seed=0))(x)
    x = BatchNormalization(axis= 3 , name= bn_name + '2b')(x)
    x = Activation('relu')(x)

    # third component of main path
    x = Conv2D(filters = F3, kernel_size = 1, strides = 1, padding = 'valid',␣
↪name = conv_name_base + '2c',
               kernel_initializer =glorot_uniform(seed=0))(x)
    x = BatchNormalization(axis= 3 , name= bn_name + '2c')(x)

    # Adding the original input to main path and applying relu activation
    x = Add()([x,x_shortcut])
    x= Activation('relu')(x)

    return x

# Function that returns convolutional blocks of Resnet50
def convolutional_block(x,f,filters,stage,block,s=2):
    # defining name
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name = 'bn' + str(stage) + block + '_branch'

    # Retreive filters
    F1, F2, F3 = filters

    # saving input value for later need pf adding back to the main path
    x_shortcut = x

    # main path
```

```
    # first component of main path
    x = Conv2D(filters = F1, kernel_size = 1, strides = s, name =␣
↪conv_name_base + '2a',
               kernel_initializer =glorot_uniform(seed=0))(x)
    x = BatchNormalization(axis= 3 , name= bn_name + '2a')(x)
    x = Activation('relu')(x)

    # second component of main path
    x = Conv2D(filters = F2, kernel_size = f, strides = 1, padding = 'same',␣
↪name = conv_name_base + '2b',
               kernel_initializer =glorot_uniform(seed=0))(x)
    x = BatchNormalization(axis= 3 , name= bn_name + '2b')(x)
    x = Activation('relu')(x)

    # third component of main path
    x = Conv2D(filters = F3, kernel_size = 1, strides = 1, padding = 'valid',␣
↪name = conv_name_base + '2c',
               kernel_initializer =glorot_uniform(seed=0))(x)
    x = BatchNormalization(axis= 3 , name= bn_name + '2c')(x)

    # shortcut path
    x_shortcut = Conv2D(filters = F3, kernel_size = 1, strides = s, name =␣
↪conv_name_base + '1',
                        kernel_initializer =glorot_uniform(seed=0))(x_shortcut)
    x_shortcut = BatchNormalization(axis= 3 , name= bn_name + '1')(x_shortcut)

    # Adding the original input to main path and applying relu activation
    x = Add()([x,x_shortcut])
    x= Activation('relu')(x)

    return x

# Function that returns model of Resnet50
def Resnet50(input_shape= (28,28,1), classes =5):
    # defining input
    x_input= Input(input_shape)

    # Converting 1 to 3 channel
#     x = Conv2D(3,(1,1),padding='same')(x_input)
    # zeropadding
    x = ZeroPadding2D((3,3))(x_input)

    # Stage 1
    x =␣
↪Conv2D(filters=64,kernel_size=7,strides=2,name='conv1',kernel_initializer =␣
↪glorot_uniform(seed=0))(x)
    x = BatchNormalization(axis=3,name ='bn_conv1')(x)
```

```
    x = Activation('relu')(x)
    x = MaxPooling2D(pool_size=3,strides = 2)(x)

    # Stage 2
    x = convolutional_block(x, f=3, filters= [64,64,256],stage =␣
↪2,block='a',s=1)
    x = identity_block(x,3,[64,64,256],stage=2,block ='b')
    x = identity_block(x,3,[64,64,256],stage=2,block ='c')

    # Stage 3
    x = convolutional_block(x, f=3, filters= [128,128,512],stage =␣
↪3,block='a',s=2)
    x = identity_block(x,3,[128,128,512],stage=3,block ='b')
    x = identity_block(x,3,[128,128,512],stage=3,block ='c')
    x = identity_block(x,3,[128,128,512],stage=3,block ='d')

    # Stage 4
    x = convolutional_block(x, f=3, filters= [256,256,1024],stage =␣
↪4,block='a',s=2)
    x = identity_block(x,3,[256,256,1024],stage=4,block ='b')
    x = identity_block(x,3,[256,256,1024],stage=4,block ='c')
    x = identity_block(x,3,[256,256,1024],stage=4,block ='d')
    x = identity_block(x,3,[256,256,1024],stage=4,block ='e')
    x = identity_block(x,3,[256,256,1024],stage=4,block ='f')

    # Stage 5
    x = convolutional_block(x, f=3, filters= [512,512,2048],stage =␣
↪5,block='a',s=2)
    x = identity_block(x,3,[512,512,2048],stage=5,block ='b')
    x = identity_block(x,3,[512,512,2048],stage=5,block ='c')

    # Average pooling
    x = AveragePooling2D((1,1),name= 'avg_pool')(x)

    # output layer
    x = Flatten()(x)
    x= BatchNormalization()(x)
    x= Dense(256, activation='relu')(x)
    x= Dropout(0.5)(x)
    x= BatchNormalization()(x)
    x= Dense(128, activation='relu')(x)
    x= Dropout(0.5)(x)
    x= BatchNormalization()(x)
    x= Dense(64, activation='relu')(x)
    x= Dropout(0.5)(x)
    x= BatchNormalization()(x)
```

```
    x = Dense(classes , activation ='softmax', name='fc' +␣
 ↪str(classes),kernel_initializer = glorot_uniform(seed=0))(x)

    # create model
    model = Model(inputs = x_input , outputs = x, name = 'Resnet50')

    return model

# Calling the model
resnet50_model = Resnet50()

# Compiling the model
resnet50_model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',␣
 ↪metrics=['accuracy'])

# Summarizing the model
resnet50_model.summary()
```

```
Model: "Resnet50"
__________________________________________________________________________________
__________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
==================
input_2 (InputLayer)            [(None, 28, 28, 1)]  0
__________________________________________________________________________________
__________________
zero_padding2d (ZeroPadding2D)  (None, 34, 34, 1)    0           input_2[0][0]
__________________________________________________________________________________
__________________
conv1 (Conv2D)                  (None, 14, 14, 64)   3200
zero_padding2d[0][0]
__________________________________________________________________________________
__________________
bn_conv1 (BatchNormalization)   (None, 14, 14, 64)   256         conv1[0][0]
__________________________________________________________________________________
__________________
activation (Activation)         (None, 14, 14, 64)   0           bn_conv1[0][0]
__________________________________________________________________________________
__________________
max_pooling2d_4 (MaxPooling2D)  (None, 6, 6, 64)     0
activation[0][0]
__________________________________________________________________________________
__________________
res2a_branch2a (Conv2D)         (None, 6, 6, 64)     4160
max_pooling2d_4[0][0]
__________________________________________________________________________________
```

```
__________________
bn2a_branch2a (BatchNormalizati (None, 6, 6, 64)     256
res2a_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_1 (Activation)       (None, 6, 6, 64)     0
bn2a_branch2a[0][0]
__________________________________________________________________________________
__________________
res2a_branch2b (Conv2D)         (None, 6, 6, 64)     36928
activation_1[0][0]
__________________________________________________________________________________
__________________
bn2a_branch2b (BatchNormalizati (None, 6, 6, 64)     256
res2a_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_2 (Activation)       (None, 6, 6, 64)     0
bn2a_branch2b[0][0]
__________________________________________________________________________________
__________________
res2a_branch2c (Conv2D)         (None, 6, 6, 256)    16640
activation_2[0][0]
__________________________________________________________________________________
__________________
res2a_branch1 (Conv2D)          (None, 6, 6, 256)    16640
max_pooling2d_4[0][0]
__________________________________________________________________________________
__________________
bn2a_branch2c (BatchNormalizati (None, 6, 6, 256)    1024
res2a_branch2c[0][0]
__________________________________________________________________________________
__________________
bn2a_branch1 (BatchNormalizatio (None, 6, 6, 256)    1024
res2a_branch1[0][0]
__________________________________________________________________________________
__________________
add (Add)                       (None, 6, 6, 256)    0
bn2a_branch2c[0][0]
bn2a_branch1[0][0]
__________________________________________________________________________________
__________________
activation_3 (Activation)       (None, 6, 6, 256)    0           add[0][0]
__________________________________________________________________________________
__________________
res2b_branch2a (Conv2D)         (None, 6, 6, 64)     16448
activation_3[0][0]
__________________________________________________________________________________
```

```
------------------
bn2b_branch2a (BatchNormalizati (None, 6, 6, 64)     256
res2b_branch2a[0][0]
------------------------------------------------------------------------------
------------------
activation_4 (Activation)       (None, 6, 6, 64)     0
bn2b_branch2a[0][0]
------------------------------------------------------------------------------
------------------
res2b_branch2b (Conv2D)         (None, 6, 6, 64)     36928
activation_4[0][0]
------------------------------------------------------------------------------
------------------
bn2b_branch2b (BatchNormalizati (None, 6, 6, 64)     256
res2b_branch2b[0][0]
------------------------------------------------------------------------------
------------------
activation_5 (Activation)       (None, 6, 6, 64)     0
bn2b_branch2b[0][0]
------------------------------------------------------------------------------
------------------
res2b_branch2c (Conv2D)         (None, 6, 6, 256)    16640
activation_5[0][0]
------------------------------------------------------------------------------
------------------
bn2b_branch2c (BatchNormalizati (None, 6, 6, 256)    1024
res2b_branch2c[0][0]
------------------------------------------------------------------------------
------------------
add_1 (Add)                     (None, 6, 6, 256)    0
bn2b_branch2c[0][0]
activation_3[0][0]
------------------------------------------------------------------------------
------------------
activation_6 (Activation)       (None, 6, 6, 256)    0           add_1[0][0]
------------------------------------------------------------------------------
------------------
res2c_branch2a (Conv2D)         (None, 6, 6, 64)     16448
activation_6[0][0]
------------------------------------------------------------------------------
------------------
bn2c_branch2a (BatchNormalizati (None, 6, 6, 64)     256
res2c_branch2a[0][0]
------------------------------------------------------------------------------
------------------
activation_7 (Activation)       (None, 6, 6, 64)     0
bn2c_branch2a[0][0]
------------------------------------------------------------------------------
```

```
__________________
res2c_branch2b (Conv2D)         (None, 6, 6, 64)     36928
activation_7[0][0]
____________________________________________________________________________
__________________
bn2c_branch2b (BatchNormalizati (None, 6, 6, 64)     256
res2c_branch2b[0][0]
____________________________________________________________________________
__________________
activation_8 (Activation)       (None, 6, 6, 64)     0
bn2c_branch2b[0][0]
____________________________________________________________________________
__________________
res2c_branch2c (Conv2D)         (None, 6, 6, 256)    16640
activation_8[0][0]
____________________________________________________________________________
__________________
bn2c_branch2c (BatchNormalizati (None, 6, 6, 256)    1024
res2c_branch2c[0][0]
____________________________________________________________________________
__________________
add_2 (Add)                     (None, 6, 6, 256)    0
bn2c_branch2c[0][0]
activation_6[0][0]
____________________________________________________________________________
__________________
activation_9 (Activation)       (None, 6, 6, 256)    0           add_2[0][0]
____________________________________________________________________________
__________________
res3a_branch2a (Conv2D)         (None, 3, 3, 128)    32896
activation_9[0][0]
____________________________________________________________________________
__________________
bn3a_branch2a (BatchNormalizati (None, 3, 3, 128)    512
res3a_branch2a[0][0]
____________________________________________________________________________
__________________
activation_10 (Activation)      (None, 3, 3, 128)    0
bn3a_branch2a[0][0]
____________________________________________________________________________
__________________
res3a_branch2b (Conv2D)         (None, 3, 3, 128)    147584
activation_10[0][0]
____________________________________________________________________________
__________________
bn3a_branch2b (BatchNormalizati (None, 3, 3, 128)    512
res3a_branch2b[0][0]
____________________________________________________________________________
```

```
__________________
activation_11 (Activation)      (None, 3, 3, 128)    0
bn3a_branch2b[0][0]
__________________________________________________________________________________
__________________
res3a_branch2c (Conv2D)         (None, 3, 3, 512)    66048
activation_11[0][0]
__________________________________________________________________________________
__________________
res3a_branch1 (Conv2D)          (None, 3, 3, 512)    131584
activation_9[0][0]
__________________________________________________________________________________
__________________
bn3a_branch2c (BatchNormalizati (None, 3, 3, 512)    2048
res3a_branch2c[0][0]
__________________________________________________________________________________
__________________
bn3a_branch1 (BatchNormalizatio (None, 3, 3, 512)    2048
res3a_branch1[0][0]
__________________________________________________________________________________
__________________
add_3 (Add)                     (None, 3, 3, 512)    0
bn3a_branch2c[0][0]
bn3a_branch1[0][0]
__________________________________________________________________________________
__________________
activation_12 (Activation)      (None, 3, 3, 512)    0           add_3[0][0]
__________________________________________________________________________________
__________________
res3b_branch2a (Conv2D)         (None, 3, 3, 128)    65664
activation_12[0][0]
__________________________________________________________________________________
__________________
bn3b_branch2a (BatchNormalizati (None, 3, 3, 128)    512
res3b_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_13 (Activation)      (None, 3, 3, 128)    0
bn3b_branch2a[0][0]
__________________________________________________________________________________
__________________
res3b_branch2b (Conv2D)         (None, 3, 3, 128)    147584
activation_13[0][0]
__________________________________________________________________________________
__________________
bn3b_branch2b (BatchNormalizati (None, 3, 3, 128)    512
res3b_branch2b[0][0]
__________________________________________________________________________________
```

```
------------------
activation_14 (Activation)      (None, 3, 3, 128)    0
bn3b_branch2b[0][0]
------------------------------------------------------------------------------
------------------
res3b_branch2c (Conv2D)         (None, 3, 3, 512)    66048
activation_14[0][0]
------------------------------------------------------------------------------
------------------
bn3b_branch2c (BatchNormalizati (None, 3, 3, 512)    2048
res3b_branch2c[0][0]
------------------------------------------------------------------------------
------------------
add_4 (Add)                     (None, 3, 3, 512)    0
bn3b_branch2c[0][0]
activation_12[0][0]
------------------------------------------------------------------------------
------------------
activation_15 (Activation)      (None, 3, 3, 512)    0           add_4[0][0]
------------------------------------------------------------------------------
------------------
res3c_branch2a (Conv2D)         (None, 3, 3, 128)    65664
activation_15[0][0]
------------------------------------------------------------------------------
------------------
bn3c_branch2a (BatchNormalizati (None, 3, 3, 128)    512
res3c_branch2a[0][0]
------------------------------------------------------------------------------
------------------
activation_16 (Activation)      (None, 3, 3, 128)    0
bn3c_branch2a[0][0]
------------------------------------------------------------------------------
------------------
res3c_branch2b (Conv2D)         (None, 3, 3, 128)    147584
activation_16[0][0]
------------------------------------------------------------------------------
------------------
bn3c_branch2b (BatchNormalizati (None, 3, 3, 128)    512
res3c_branch2b[0][0]
------------------------------------------------------------------------------
------------------
activation_17 (Activation)      (None, 3, 3, 128)    0
bn3c_branch2b[0][0]
------------------------------------------------------------------------------
------------------
res3c_branch2c (Conv2D)         (None, 3, 3, 512)    66048
activation_17[0][0]
------------------------------------------------------------------------------
```

```
------------------
bn3c_branch2c (BatchNormalizati (None, 3, 3, 512)    2048
res3c_branch2c[0][0]
--------------------------------------------------------------------------------
------------------
add_5 (Add)                     (None, 3, 3, 512)    0
bn3c_branch2c[0][0]
activation_15[0][0]
--------------------------------------------------------------------------------
------------------
activation_18 (Activation)      (None, 3, 3, 512)    0           add_5[0][0]
--------------------------------------------------------------------------------
------------------
res3d_branch2a (Conv2D)         (None, 3, 3, 128)    65664
activation_18[0][0]
--------------------------------------------------------------------------------
------------------
bn3d_branch2a (BatchNormalizati (None, 3, 3, 128)    512
res3d_branch2a[0][0]
--------------------------------------------------------------------------------
------------------
activation_19 (Activation)      (None, 3, 3, 128)    0
bn3d_branch2a[0][0]
--------------------------------------------------------------------------------
------------------
res3d_branch2b (Conv2D)         (None, 3, 3, 128)    147584
activation_19[0][0]
--------------------------------------------------------------------------------
------------------
bn3d_branch2b (BatchNormalizati (None, 3, 3, 128)    512
res3d_branch2b[0][0]
--------------------------------------------------------------------------------
------------------
activation_20 (Activation)      (None, 3, 3, 128)    0
bn3d_branch2b[0][0]
--------------------------------------------------------------------------------
------------------
res3d_branch2c (Conv2D)         (None, 3, 3, 512)    66048
activation_20[0][0]
--------------------------------------------------------------------------------
------------------
bn3d_branch2c (BatchNormalizati (None, 3, 3, 512)    2048
res3d_branch2c[0][0]
--------------------------------------------------------------------------------
------------------
add_6 (Add)                     (None, 3, 3, 512)    0
bn3d_branch2c[0][0]
activation_18[0][0]
```

```
__________________________________________________________________________________
__________________
activation_21 (Activation)      (None, 3, 3, 512)    0           add_6[0][0]
__________________________________________________________________________________
__________________
res4a_branch2a (Conv2D)         (None, 2, 2, 256)    131328
activation_21[0][0]
__________________________________________________________________________________
__________________
bn4a_branch2a (BatchNormalizati (None, 2, 2, 256)    1024
res4a_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_22 (Activation)      (None, 2, 2, 256)    0
bn4a_branch2a[0][0]
__________________________________________________________________________________
__________________
res4a_branch2b (Conv2D)         (None, 2, 2, 256)    590080
activation_22[0][0]
__________________________________________________________________________________
__________________
bn4a_branch2b (BatchNormalizati (None, 2, 2, 256)    1024
res4a_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_23 (Activation)      (None, 2, 2, 256)    0
bn4a_branch2b[0][0]
__________________________________________________________________________________
__________________
res4a_branch2c (Conv2D)         (None, 2, 2, 1024)   263168
activation_23[0][0]
__________________________________________________________________________________
__________________
res4a_branch1 (Conv2D)          (None, 2, 2, 1024)   525312
activation_21[0][0]
__________________________________________________________________________________
__________________
bn4a_branch2c (BatchNormalizati (None, 2, 2, 1024)   4096
res4a_branch2c[0][0]
__________________________________________________________________________________
__________________
bn4a_branch1 (BatchNormalizatio (None, 2, 2, 1024)   4096
res4a_branch1[0][0]
__________________________________________________________________________________
__________________
add_7 (Add)                     (None, 2, 2, 1024)   0
bn4a_branch2c[0][0]
bn4a_branch1[0][0]
```

```
__________________________________________________________________________________
__________________
activation_24 (Activation)      (None, 2, 2, 1024)   0           add_7[0][0]
__________________________________________________________________________________
__________________
res4b_branch2a (Conv2D)         (None, 2, 2, 256)    262400
activation_24[0][0]
__________________________________________________________________________________
__________________
bn4b_branch2a (BatchNormalizati (None, 2, 2, 256)    1024
res4b_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_25 (Activation)      (None, 2, 2, 256)    0
bn4b_branch2a[0][0]
__________________________________________________________________________________
__________________
res4b_branch2b (Conv2D)         (None, 2, 2, 256)    590080
activation_25[0][0]
__________________________________________________________________________________
__________________
bn4b_branch2b (BatchNormalizati (None, 2, 2, 256)    1024
res4b_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_26 (Activation)      (None, 2, 2, 256)    0
bn4b_branch2b[0][0]
__________________________________________________________________________________
__________________
res4b_branch2c (Conv2D)         (None, 2, 2, 1024)   263168
activation_26[0][0]
__________________________________________________________________________________
__________________
bn4b_branch2c (BatchNormalizati (None, 2, 2, 1024)   4096
res4b_branch2c[0][0]
__________________________________________________________________________________
__________________
add_8 (Add)                     (None, 2, 2, 1024)   0
bn4b_branch2c[0][0]
activation_24[0][0]
__________________________________________________________________________________
__________________
activation_27 (Activation)      (None, 2, 2, 1024)   0           add_8[0][0]
__________________________________________________________________________________
__________________
res4c_branch2a (Conv2D)         (None, 2, 2, 256)    262400
activation_27[0][0]
__________________________________________________________________________________
```

```
__________________
bn4c_branch2a (BatchNormalizati (None, 2, 2, 256)    1024
res4c_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_28 (Activation)      (None, 2, 2, 256)    0
bn4c_branch2a[0][0]
__________________________________________________________________________________
__________________
res4c_branch2b (Conv2D)         (None, 2, 2, 256)    590080
activation_28[0][0]
__________________________________________________________________________________
__________________
bn4c_branch2b (BatchNormalizati (None, 2, 2, 256)    1024
res4c_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_29 (Activation)      (None, 2, 2, 256)    0
bn4c_branch2b[0][0]
__________________________________________________________________________________
__________________
res4c_branch2c (Conv2D)         (None, 2, 2, 1024)   263168
activation_29[0][0]
__________________________________________________________________________________
__________________
bn4c_branch2c (BatchNormalizati (None, 2, 2, 1024)   4096
res4c_branch2c[0][0]
__________________________________________________________________________________
__________________
add_9 (Add)                     (None, 2, 2, 1024)   0
bn4c_branch2c[0][0]
activation_27[0][0]
__________________________________________________________________________________
__________________
activation_30 (Activation)      (None, 2, 2, 1024)   0           add_9[0][0]
__________________________________________________________________________________
__________________
res4d_branch2a (Conv2D)         (None, 2, 2, 256)    262400
activation_30[0][0]
__________________________________________________________________________________
__________________
bn4d_branch2a (BatchNormalizati (None, 2, 2, 256)    1024
res4d_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_31 (Activation)      (None, 2, 2, 256)    0
bn4d_branch2a[0][0]
__________________________________________________________________________________
```

```
__________________
res4d_branch2b (Conv2D)         (None, 2, 2, 256)    590080
activation_31[0][0]
__________________________________________________________________________________
__________________
bn4d_branch2b (BatchNormalizati (None, 2, 2, 256)    1024
res4d_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_32 (Activation)      (None, 2, 2, 256)    0
bn4d_branch2b[0][0]
__________________________________________________________________________________
__________________
res4d_branch2c (Conv2D)         (None, 2, 2, 1024)   263168
activation_32[0][0]
__________________________________________________________________________________
__________________
bn4d_branch2c (BatchNormalizati (None, 2, 2, 1024)   4096
res4d_branch2c[0][0]
__________________________________________________________________________________
__________________
add_10 (Add)                    (None, 2, 2, 1024)   0
bn4d_branch2c[0][0]
activation_30[0][0]
__________________________________________________________________________________
__________________
activation_33 (Activation)      (None, 2, 2, 1024)   0           add_10[0][0]
__________________________________________________________________________________
__________________
res4e_branch2a (Conv2D)         (None, 2, 2, 256)    262400
activation_33[0][0]
__________________________________________________________________________________
__________________
bn4e_branch2a (BatchNormalizati (None, 2, 2, 256)    1024
res4e_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_34 (Activation)      (None, 2, 2, 256)    0
bn4e_branch2a[0][0]
__________________________________________________________________________________
__________________
res4e_branch2b (Conv2D)         (None, 2, 2, 256)    590080
activation_34[0][0]
__________________________________________________________________________________
__________________
bn4e_branch2b (BatchNormalizati (None, 2, 2, 256)    1024
res4e_branch2b[0][0]
__________________________________________________________________________________
```

```
__________________
activation_35 (Activation)      (None, 2, 2, 256)    0
bn4e_branch2b[0][0]
__________________________________________________________________________________
__________________
res4e_branch2c (Conv2D)         (None, 2, 2, 1024)   263168
activation_35[0][0]
__________________________________________________________________________________
__________________
bn4e_branch2c (BatchNormalizati (None, 2, 2, 1024)   4096
res4e_branch2c[0][0]
__________________________________________________________________________________
__________________
add_11 (Add)                    (None, 2, 2, 1024)   0
bn4e_branch2c[0][0]
activation_33[0][0]
__________________________________________________________________________________
__________________
activation_36 (Activation)      (None, 2, 2, 1024)   0           add_11[0][0]
__________________________________________________________________________________
__________________
res4f_branch2a (Conv2D)         (None, 2, 2, 256)    262400
activation_36[0][0]
__________________________________________________________________________________
__________________
bn4f_branch2a (BatchNormalizati (None, 2, 2, 256)    1024
res4f_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_37 (Activation)      (None, 2, 2, 256)    0
bn4f_branch2a[0][0]
__________________________________________________________________________________
__________________
res4f_branch2b (Conv2D)         (None, 2, 2, 256)    590080
activation_37[0][0]
__________________________________________________________________________________
__________________
bn4f_branch2b (BatchNormalizati (None, 2, 2, 256)    1024
res4f_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_38 (Activation)      (None, 2, 2, 256)    0
bn4f_branch2b[0][0]
__________________________________________________________________________________
__________________
res4f_branch2c (Conv2D)         (None, 2, 2, 1024)   263168
activation_38[0][0]
__________________________________________________________________________________
```

```
__________________
bn4f_branch2c (BatchNormalizati (None, 2, 2, 1024)   4096
res4f_branch2c[0][0]
__________________________________________________________________________________
__________________
add_12 (Add)                    (None, 2, 2, 1024)   0
bn4f_branch2c[0][0]
activation_36[0][0]
__________________________________________________________________________________
__________________
activation_39 (Activation)      (None, 2, 2, 1024)   0           add_12[0][0]
__________________________________________________________________________________
__________________
res5a_branch2a (Conv2D)         (None, 1, 1, 512)    524800
activation_39[0][0]
__________________________________________________________________________________
__________________
bn5a_branch2a (BatchNormalizati (None, 1, 1, 512)    2048
res5a_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_40 (Activation)      (None, 1, 1, 512)    0
bn5a_branch2a[0][0]
__________________________________________________________________________________
__________________
res5a_branch2b (Conv2D)         (None, 1, 1, 512)    2359808
activation_40[0][0]
__________________________________________________________________________________
__________________
bn5a_branch2b (BatchNormalizati (None, 1, 1, 512)    2048
res5a_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_41 (Activation)      (None, 1, 1, 512)    0
bn5a_branch2b[0][0]
__________________________________________________________________________________
__________________
res5a_branch2c (Conv2D)         (None, 1, 1, 2048)   1050624
activation_41[0][0]
__________________________________________________________________________________
__________________
res5a_branch1 (Conv2D)          (None, 1, 1, 2048)   2099200
activation_39[0][0]
__________________________________________________________________________________
__________________
bn5a_branch2c (BatchNormalizati (None, 1, 1, 2048)   8192
res5a_branch2c[0][0]
__________________________________________________________________________________
```

```
__________________
bn5a_branch1 (BatchNormalizatio (None, 1, 1, 2048)   8192
res5a_branch1[0][0]
__________________________________________________________________________________
__________________
add_13 (Add)                    (None, 1, 1, 2048)   0
bn5a_branch2c[0][0]
bn5a_branch1[0][0]
__________________________________________________________________________________
__________________
activation_42 (Activation)      (None, 1, 1, 2048)   0           add_13[0][0]
__________________________________________________________________________________
__________________
res5b_branch2a (Conv2D)         (None, 1, 1, 512)    1049088
activation_42[0][0]
__________________________________________________________________________________
__________________
bn5b_branch2a (BatchNormalizati (None, 1, 1, 512)    2048
res5b_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_43 (Activation)      (None, 1, 1, 512)    0
bn5b_branch2a[0][0]
__________________________________________________________________________________
__________________
res5b_branch2b (Conv2D)         (None, 1, 1, 512)    2359808
activation_43[0][0]
__________________________________________________________________________________
__________________
bn5b_branch2b (BatchNormalizati (None, 1, 1, 512)    2048
res5b_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_44 (Activation)      (None, 1, 1, 512)    0
bn5b_branch2b[0][0]
__________________________________________________________________________________
__________________
res5b_branch2c (Conv2D)         (None, 1, 1, 2048)   1050624
activation_44[0][0]
__________________________________________________________________________________
__________________
bn5b_branch2c (BatchNormalizati (None, 1, 1, 2048)   8192
res5b_branch2c[0][0]
__________________________________________________________________________________
__________________
add_14 (Add)                    (None, 1, 1, 2048)   0
bn5b_branch2c[0][0]
activation_42[0][0]
```

```
__________________________________________________________________________________
__________________
activation_45 (Activation)      (None, 1, 1, 2048)   0           add_14[0][0]
__________________________________________________________________________________
__________________
res5c_branch2a (Conv2D)         (None, 1, 1, 512)    1049088
activation_45[0][0]
__________________________________________________________________________________
__________________
bn5c_branch2a (BatchNormalizati (None, 1, 1, 512)    2048
res5c_branch2a[0][0]
__________________________________________________________________________________
__________________
activation_46 (Activation)      (None, 1, 1, 512)    0
bn5c_branch2a[0][0]
__________________________________________________________________________________
__________________
res5c_branch2b (Conv2D)         (None, 1, 1, 512)    2359808
activation_46[0][0]
__________________________________________________________________________________
__________________
bn5c_branch2b (BatchNormalizati (None, 1, 1, 512)    2048
res5c_branch2b[0][0]
__________________________________________________________________________________
__________________
activation_47 (Activation)      (None, 1, 1, 512)    0
bn5c_branch2b[0][0]
__________________________________________________________________________________
__________________
res5c_branch2c (Conv2D)         (None, 1, 1, 2048)   1050624
activation_47[0][0]
__________________________________________________________________________________
__________________
bn5c_branch2c (BatchNormalizati (None, 1, 1, 2048)   8192
res5c_branch2c[0][0]
__________________________________________________________________________________
__________________
add_15 (Add)                    (None, 1, 1, 2048)   0
bn5c_branch2c[0][0]
activation_45[0][0]
__________________________________________________________________________________
__________________
activation_48 (Activation)      (None, 1, 1, 2048)   0           add_15[0][0]
__________________________________________________________________________________
__________________
avg_pool (AveragePooling2D)     (None, 1, 1, 2048)   0
activation_48[0][0]
__________________________________________________________________________________
```

```
__________________
flatten_2 (Flatten)             (None, 2048)         0           avg_pool[0][0]
____________________________________________________________________________
__________________
batch_normalization_1 (BatchNor (None, 2048)         8192        flatten_2[0][0]
____________________________________________________________________________
__________________
dense_6 (Dense)                 (None, 256)          524544
batch_normalization_1[0][0]
____________________________________________________________________________
__________________
dropout_7 (Dropout)             (None, 256)          0           dense_6[0][0]
____________________________________________________________________________
__________________
batch_normalization_2 (BatchNor (None, 256)          1024        dropout_7[0][0]
____________________________________________________________________________
__________________
dense_7 (Dense)                 (None, 128)          32896
batch_normalization_2[0][0]
____________________________________________________________________________
__________________
dropout_8 (Dropout)             (None, 128)          0           dense_7[0][0]
____________________________________________________________________________
__________________
batch_normalization_3 (BatchNor (None, 128)          512         dropout_8[0][0]
____________________________________________________________________________
__________________
dense_8 (Dense)                 (None, 64)           8256
batch_normalization_3[0][0]
____________________________________________________________________________
__________________
dropout_9 (Dropout)             (None, 64)           0           dense_8[0][0]
____________________________________________________________________________
__________________
batch_normalization_4 (BatchNor (None, 64)           256         dropout_9[0][0]
____________________________________________________________________________
__________________
fc5 (Dense)                     (None, 5)            325
batch_normalization_4[0][0]
============================================================================
==================
Total params: 24,157,445
Trainable params: 24,099,333
Non-trainable params: 58,112
____________________________________________________________________________
__________________
```

4.4.2 Plotting of CNN Resnet50 Model Architecture:

[1]:
```
# Plotting the Resnet50 model
# plot_model(resnet50_model, show_shapes=True, to_file='resnet50_model.png')
```

5 CM[7]:

NOTE: In this solution, Parts refer to the points needs to be covered in CM[7] as per the assignment instructions, where, Part 1 is displaying Runtime performance of training and testing models, Part 2 is comparision of all model(it is mentioned at last) and Part 3 refers to plotting of testing and training accuracy and loss.

6 Deep CNN Model

6.1 CM[7]

6.2 Part 1 - 1st Model(Deep CNN Model)

6.2.1 Runtime Performance of Training and Testing with Deep CNN Model:

In Runtime performance , the training data achieved accuracy of 0.9093 and loss of 0.2326, whereas , the test data achieved accuracy of 0.9002 and loss of 0.2480, with 30 epochs.

[12]:
```
deep_cnn_history=cnn_layers_model.
↪fit(x_train,y_train,validation_data=(x_test,y_test),epochs=30,batch_size=128,verbose=1)
```

```
Epoch 1/30
469/469 [==============================] - 25s 16ms/step - loss: 0.9248 -
accuracy: 0.5899 - val_loss: 0.4024 - val_accuracy: 0.8421
Epoch 2/30
469/469 [==============================] - 7s 15ms/step - loss: 0.4697 -
accuracy: 0.8080 - val_loss: 0.4739 - val_accuracy: 0.7924
Epoch 3/30
469/469 [==============================] - 7s 15ms/step - loss: 0.4154 -
accuracy: 0.8333 - val_loss: 0.3636 - val_accuracy: 0.8563
Epoch 4/30
469/469 [==============================] - 7s 15ms/step - loss: 0.3823 -
accuracy: 0.8444 - val_loss: 0.3534 - val_accuracy: 0.8564
Epoch 5/30
469/469 [==============================] - 7s 15ms/step - loss: 0.3532 -
accuracy: 0.8553 - val_loss: 0.3121 - val_accuracy: 0.8705
Epoch 6/30
469/469 [==============================] - 7s 15ms/step - loss: 0.3408 -
accuracy: 0.8637 - val_loss: 0.2875 - val_accuracy: 0.8839
Epoch 7/30
469/469 [==============================] - 7s 15ms/step - loss: 0.3220 -
accuracy: 0.8702 - val_loss: 0.2902 - val_accuracy: 0.8832
Epoch 8/30
469/469 [==============================] - 7s 15ms/step - loss: 0.3154 -
```

```
accuracy: 0.8736 - val_loss: 0.2982 - val_accuracy: 0.8712
Epoch 9/30
469/469 [==============================] - 7s 15ms/step - loss: 0.3064 -
accuracy: 0.8742 - val_loss: 0.2822 - val_accuracy: 0.8842
Epoch 10/30
469/469 [==============================] - 7s 15ms/step - loss: 0.3004 -
accuracy: 0.8774 - val_loss: 0.2804 - val_accuracy: 0.8860
Epoch 11/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2977 -
accuracy: 0.8795 - val_loss: 0.2573 - val_accuracy: 0.8978
Epoch 12/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2917 -
accuracy: 0.8827 - val_loss: 0.2624 - val_accuracy: 0.8981
Epoch 13/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2828 -
accuracy: 0.8873 - val_loss: 0.2667 - val_accuracy: 0.8915
Epoch 14/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2748 -
accuracy: 0.8896 - val_loss: 0.2641 - val_accuracy: 0.8929
Epoch 15/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2776 -
accuracy: 0.8893 - val_loss: 0.2578 - val_accuracy: 0.9001
Epoch 16/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2650 -
accuracy: 0.8939 - val_loss: 0.2792 - val_accuracy: 0.8876
Epoch 17/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2571 -
accuracy: 0.8962 - val_loss: 0.2591 - val_accuracy: 0.8936
Epoch 18/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2626 -
accuracy: 0.8951 - val_loss: 0.2587 - val_accuracy: 0.8951
Epoch 19/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2564 -
accuracy: 0.8982 - val_loss: 0.3101 - val_accuracy: 0.8728
Epoch 20/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2581 -
accuracy: 0.8981 - val_loss: 0.2503 - val_accuracy: 0.8993
Epoch 21/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2505 -
accuracy: 0.9006 - val_loss: 0.2538 - val_accuracy: 0.9011
Epoch 22/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2491 -
accuracy: 0.9001 - val_loss: 0.2727 - val_accuracy: 0.8894
Epoch 23/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2433 -
accuracy: 0.9025 - val_loss: 0.2547 - val_accuracy: 0.8967
Epoch 24/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2513 -
```

```
accuracy: 0.9014 - val_loss: 0.2758 - val_accuracy: 0.8815
Epoch 25/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2474 -
accuracy: 0.9014 - val_loss: 0.2512 - val_accuracy: 0.9001
Epoch 26/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2410 -
accuracy: 0.9043 - val_loss: 0.2503 - val_accuracy: 0.8981
Epoch 27/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2439 -
accuracy: 0.9028 - val_loss: 0.2491 - val_accuracy: 0.9010
Epoch 28/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2381 -
accuracy: 0.9034 - val_loss: 0.2504 - val_accuracy: 0.9025
Epoch 29/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2336 -
accuracy: 0.9053 - val_loss: 0.2559 - val_accuracy: 0.8997
Epoch 30/30
469/469 [==============================] - 7s 15ms/step - loss: 0.2326 -
accuracy: 0.9093 - val_loss: 0.2480 - val_accuracy: 0.9002
```

6.2.2 Evaluation of Testing with Deep CNN Model:

[13]:
```
deep_cnn_score=cnn_layers_model.evaluate(x_test,y_test,verbose=0)
print('The loss while evaluation of test data with Deep CNN is: {} '.
 ↪format(deep_cnn_score[0]))
print('The accuracy while evaluation of test data with Deep CNN is: {} '.
 ↪format(deep_cnn_score[1]))
```

```
The loss while evaluation of test data with Deep CNN is: 0.24798935651779175
The accuracy while evaluation of test data with Deep CNN is: 0.9002000093460083
```

6.3 CM[7]

6.4 Part 3 - 1st Model(Deep CNN Model)

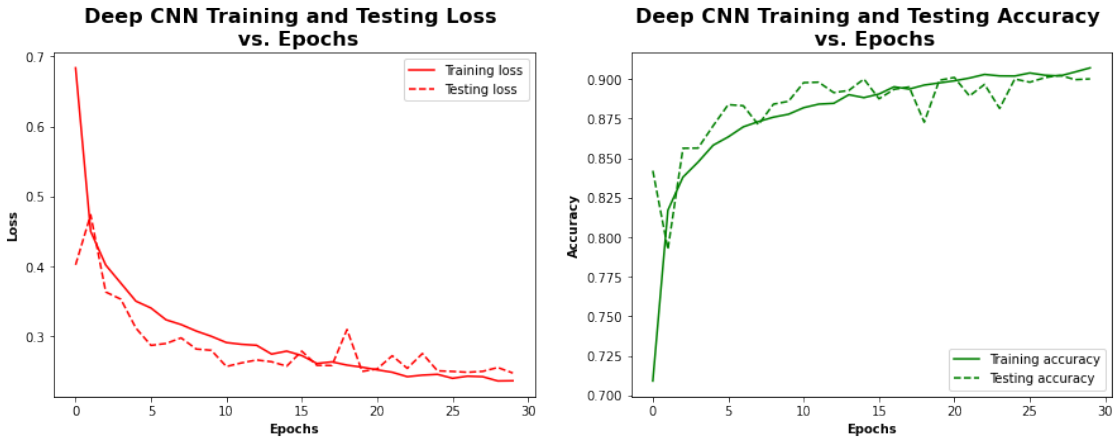
6.4.1 Plotting Loss and Accuracy vs. Epochs of Deep CNN Model:

Here,the below plots shows the accuracy and loss rate with respect to epochs, where, training set and testing accuracy is increasing and loss is decreasing with almost same pace. Here, accuracy of test set and train increses and loss decreases which tells that the model is learning and as it overlaps , so, it also represents that the model of good fit and not going through any kind of overfitting or underfittiing. This good fit is achieved using dropout layers to mitigate the effect of overfitting.

Although , in case where dropout layers are not added , model gets overfitted and reached upto training accuracy of 97 and loss of 0.02, whereas , after reducing the overfitting, the accuracy and loss of model is also compromised and reduced to 90 % with loss of 0.2.

Apart from this, the labels of this dataset are also classified with some hidden pattern , even confusing to human eye. So, our accuracy is quite good in such condition and further, we will impove it or atleast try with different implementations to improve it.

[14]:
```
fig, axes= plt.subplots(1,2,figsize=(15,5))
axes[0].plot(deep_cnn_history.history['loss'],label='Training loss',color='red')
axes[0].plot(deep_cnn_history.history['val_loss'],label='Testing␣
 ↪loss',color='red',linestyle ='dashed')
axes[1].plot(deep_cnn_history.history['accuracy'],label='Training␣
 ↪accuracy',color='green')
axes[1].plot(deep_cnn_history.history['val_accuracy'],label='Testing␣
 ↪accuracy',color='green',linestyle ='dashed')
axes[0].set_xlabel('Epochs',fontweight='bold')
axes[0].set_ylabel('Loss',fontweight='bold')
axes[1].set_xlabel('Epochs',fontweight='bold')
axes[1].set_ylabel('Accuracy',fontweight='bold')
axes[0].set_title('Deep CNN Training and Testing Loss \n vs.␣
 ↪Epochs',fontsize=16,fontweight='bold')
axes[1].set_title('Deep CNN Training and Testing Accuracy \n vs.␣
 ↪Epochs',fontsize=16,fontweight='bold')
axes[0].legend()
axes[1].legend()
plt.show()
```

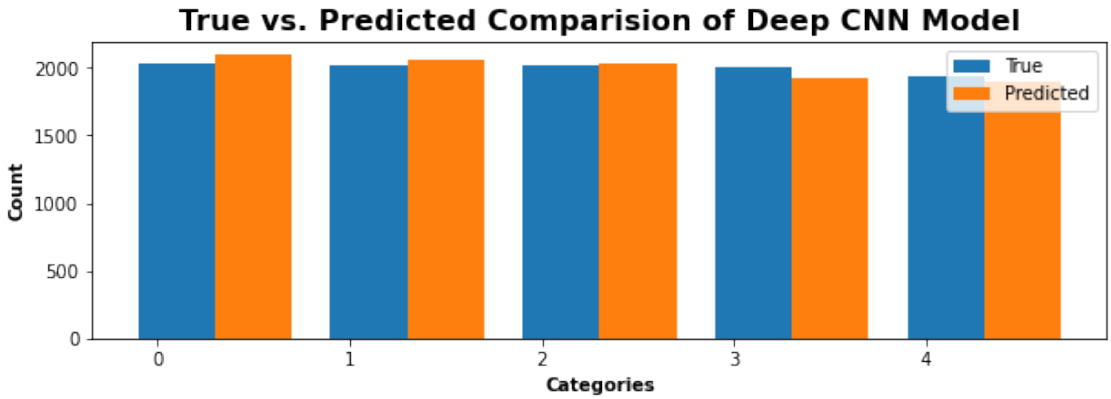


6.4.2 Visualizing True and Predicted Class with Deep CNN Model:

This barplot displays the true and predicted class labels, where model seems to have slightly more false positives in Category 0 and 3 as compared to others.

[15]:
```
# Function that returns True and Predicted barplot
def true_pred(model,model_name):
    predict = model.predict(x_test)
    pred=np.argmax(predict,axis=1)
    pred = pd.DataFrame(pred)
    plt.figure(figsize=(10,3))
```

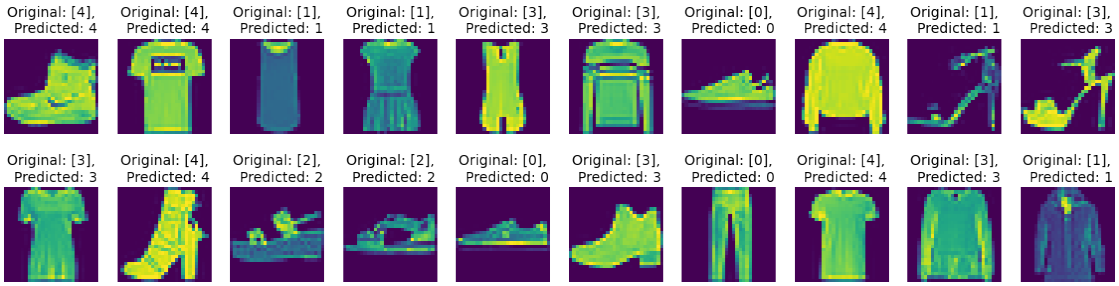
```
    plt.bar(np.arange(0,5)+0.1,y_test.value_counts(),width=0.4, label = 'True')
    plt.bar(np.arange(0,5)+0.5,pred.value_counts(),width=0.4, label =␣
 ↪'Predicted')
    plt.xlabel('Categories',fontweight='bold')
    plt.ylabel('Count',fontweight='bold')
    plt.legend()
    plt.title('True vs. Predicted Comparision of {}'.
 ↪format(model_name),fontweight='bold',fontsize= 16)
# Function that return the num no. of images of actual data and its true and␣
 ↪predicted classes
def Vis_true_pred(model,num):
    predict = model.predict(x_test)
    pred=np.argmax(predict,axis=1)
    f, ax = plt.subplots(int(num/10),10, figsize=(20,5*int((num/20)+0.5)))
    for i in range(num):
        ax[i//10, i%10].imshow(x_test.reshape((10000,28,28))[i])
        ax[i//10, i%10].axis('off')
        ax[i//10, i%10].set_title('Original: {},\n  Predicted: {}'.
 ↪format(y_test.iloc[i].values,
                                                                    ␣
 ↪pred[i]),fontsize=14)
    plt.show()

deep_cnn_true_pred= true_pred(cnn_layers_model,'Deep CNN Model')
```



6.4.3 True and Predicted visualization of 20 images of Deep CNN Model:

The below plot displays the actual 20 images with their original category and their predicted ones. Here, there are no image categories which are predicted wrong, among the passed 20 images.

[16]:
```
# True and Predicted visualization of 20 images
deep_cnn_visualization= Vis_true_pred(cnn_layers_model,20)
```



7 CNN Inception Model

7.1 CM[7]

7.2 Part 1 - 2nd Model(CNN Inception Model)

7.2.1 Runtime Performance of Training and Testing with CNN Inception Model:

In Runtime performance , the training data achieved accuracy of 0.9157 and loss of 0.4237 whereas , the test data achieved accuracy of 0.8489 and loss of 0.600, with 30 epochs.

[19]:
```
cnn_inception_history=cnn_inception_model.
↪fit(x_train,y_train,validation_data=(x_test,y_test),epochs=30,batch_size=128,verbose=1)
```

```
Epoch 1/30
469/469 [==============================] - 63s 119ms/step - loss: 2.7836 -
accuracy: 0.5044 - val_loss: 1.1522 - val_accuracy: 0.7064
Epoch 2/30
469/469 [==============================] - 54s 116ms/step - loss: 0.9548 -
accuracy: 0.7385 - val_loss: 0.7687 - val_accuracy: 0.8109
Epoch 3/30
469/469 [==============================] - 55s 117ms/step - loss: 0.8156 -
accuracy: 0.7840 - val_loss: 0.6728 - val_accuracy: 0.8343
Epoch 4/30
469/469 [==============================] - 54s 115ms/step - loss: 0.7147 -
accuracy: 0.8166 - val_loss: 0.6396 - val_accuracy: 0.8478
Epoch 5/30
469/469 [==============================] - 54s 115ms/step - loss: 0.6696 -
accuracy: 0.8270 - val_loss: 0.6199 - val_accuracy: 0.8456
Epoch 6/30
469/469 [==============================] - 54s 115ms/step - loss: 0.6277 -
accuracy: 0.8339 - val_loss: 0.5820 - val_accuracy: 0.8619
Epoch 7/30
469/469 [==============================] - 54s 115ms/step - loss: 0.6019 -
accuracy: 0.8451 - val_loss: 0.5968 - val_accuracy: 0.8454
Epoch 8/30
469/469 [==============================] - 54s 115ms/step - loss: 0.5692 -
accuracy: 0.8511 - val_loss: 0.5719 - val_accuracy: 0.8464
```

```
Epoch 9/30
469/469 [==============================] - 54s 115ms/step - loss: 0.5474 -
accuracy: 0.8563 - val_loss: 0.5460 - val_accuracy: 0.8585
Epoch 10/30
469/469 [==============================] - 54s 115ms/step - loss: 0.5335 -
accuracy: 0.8583 - val_loss: 0.6498 - val_accuracy: 0.8004
Epoch 11/30
469/469 [==============================] - 54s 115ms/step - loss: 0.5200 -
accuracy: 0.8629 - val_loss: 0.5377 - val_accuracy: 0.8688
Epoch 12/30
469/469 [==============================] - 54s 115ms/step - loss: 0.5097 -
accuracy: 0.8699 - val_loss: 0.5583 - val_accuracy: 0.8532
Epoch 13/30
469/469 [==============================] - 54s 115ms/step - loss: 0.5019 -
accuracy: 0.8735 - val_loss: 0.5572 - val_accuracy: 0.8494
Epoch 14/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4871 -
accuracy: 0.8757 - val_loss: 0.5237 - val_accuracy: 0.8636
Epoch 15/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4815 -
accuracy: 0.8810 - val_loss: 0.5928 - val_accuracy: 0.8315
Epoch 16/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4796 -
accuracy: 0.8811 - val_loss: 0.5298 - val_accuracy: 0.8615
Epoch 17/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4702 -
accuracy: 0.8856 - val_loss: 0.5285 - val_accuracy: 0.8655
Epoch 18/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4677 -
accuracy: 0.8870 - val_loss: 0.5560 - val_accuracy: 0.8535
Epoch 19/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4551 -
accuracy: 0.8907 - val_loss: 0.5589 - val_accuracy: 0.8538
Epoch 20/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4553 -
accuracy: 0.8951 - val_loss: 0.5671 - val_accuracy: 0.8500
Epoch 21/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4489 -
accuracy: 0.8968 - val_loss: 0.5726 - val_accuracy: 0.8520
Epoch 22/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4468 -
accuracy: 0.8999 - val_loss: 0.5629 - val_accuracy: 0.8607
Epoch 23/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4409 -
accuracy: 0.9022 - val_loss: 0.5988 - val_accuracy: 0.8408
Epoch 24/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4395 -
accuracy: 0.9020 - val_loss: 0.5771 - val_accuracy: 0.8530
```

```
Epoch 25/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4379 -
accuracy: 0.9049 - val_loss: 0.5702 - val_accuracy: 0.8573
Epoch 26/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4377 -
accuracy: 0.9066 - val_loss: 0.5761 - val_accuracy: 0.8506
Epoch 27/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4360 -
accuracy: 0.9090 - val_loss: 0.6288 - val_accuracy: 0.8404
Epoch 28/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4327 -
accuracy: 0.9095 - val_loss: 0.5776 - val_accuracy: 0.8559
Epoch 29/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4277 -
accuracy: 0.9144 - val_loss: 0.5859 - val_accuracy: 0.8536
Epoch 30/30
469/469 [==============================] - 54s 115ms/step - loss: 0.4237 -
accuracy: 0.9157 - val_loss: 0.6001 - val_accuracy: 0.8489
```

7.2.2 Evaluation of Testing with CNN Inception Model:

[20]:
```
cnn_inception_score=cnn_inception_model.evaluate(x_test,y_test,verbose=0)
print('The loss while evaluation of test data with Deep CNN is: {} '.
 ↪format(cnn_inception_score[0]))
print('The accuracy while evaluation of test data with Deep CNN is: {} '.
 ↪format(cnn_inception_score[1]))
```

```
The loss while evaluation of test data with Deep CNN is: 0.6000663042068481
The accuracy while evaluation of test data with Deep CNN is: 0.8489000201225281
```

7.3 CM[7]

7.4 Part 3 - 2nd Model(CNN Inception Model)

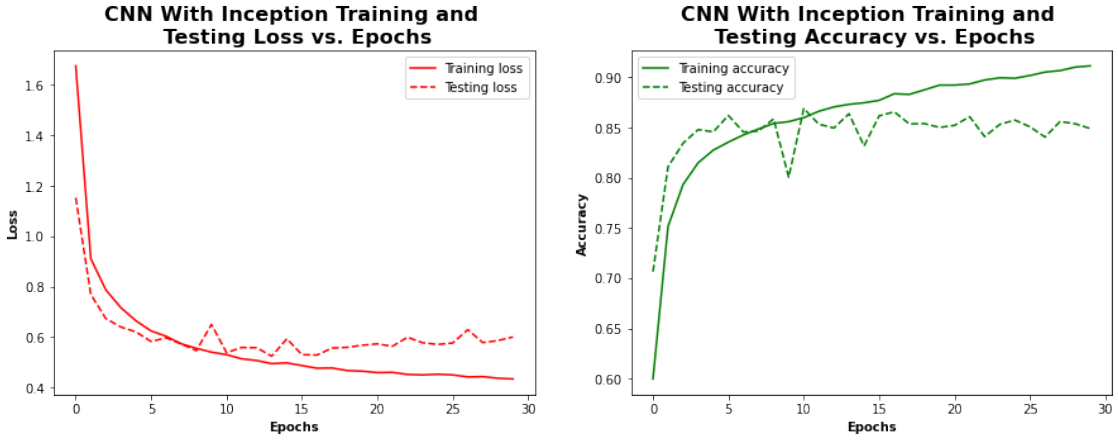
7.4.1 Plotting of Loss and Accuracy vs. Epochs of CNN Inception Model:

Here,the below plots shows the accuracy and loss rate with respect to epochs, where, training set accuracy keeps on increasing and loss is decreasing, but in test set, there seems to be minimal or no change in accuracy and loss and, even after applying the batchnormalization on layers, test set training has fluctuations . But, the good thing is its still increasing with slower pace and thus, learning. This may be due to some elements in the dataset are randomly classified. Also, as our test set accuracy in not increasing much and its loss is not decreasing to great extent .So, overfitting can be problem and if it so, then the L1-L2 regularization is applied with a dropout layer, which reduced the problem to some extent.

Without regularization and dropout layer, model was giving good accuracy of about 99.5 on training but, worse in test set.In that case, the test loss started increasing and test accuracy started decreasing on test set at significant rate which also tells that moedl is not learning the testing points , rather memorizing it. So, the dropout and regularization reduces overfitting and further may be vanishing gradient problem in neural networks to some extent , but still, in our case, an

even better model is required, the one without any overfitting. Apart from it, after trying many hyperparameters and adding dense layers using Inception concept, this is the best one inception with CNN model achieved, with minimal overfitting.

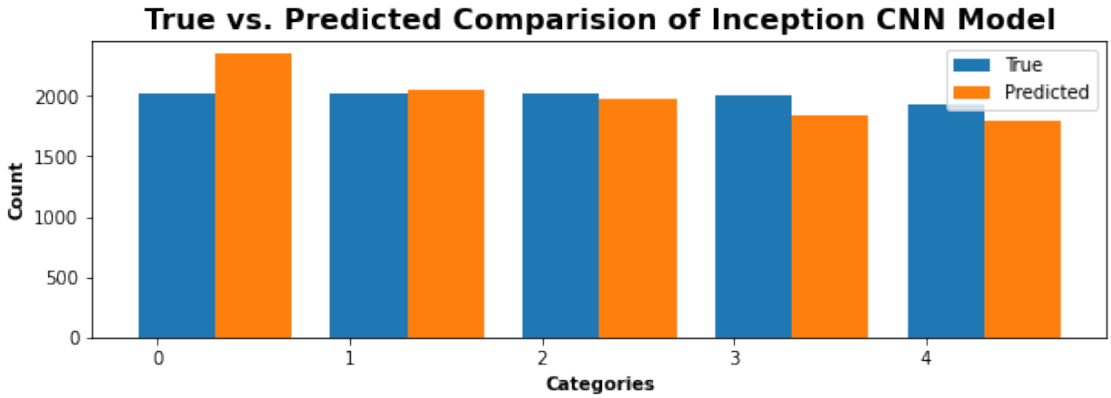
[21]:
```
fig, axes= plt.subplots(1,2,figsize=(15,5))
axes[0].plot(cnn_inception_history.history['loss'],label='Training␣
 ↪loss',color='red')
axes[0].plot(cnn_inception_history.history['val_loss'],label='Testing␣
 ↪loss',color='red',linestyle ='dashed')
axes[1].plot(cnn_inception_history.history['accuracy'],label='Training␣
 ↪accuracy',color='green')
axes[1].plot(cnn_inception_history.history['val_accuracy'],label='Testing␣
 ↪accuracy',color='green',linestyle ='dashed')
axes[0].set_xlabel('Epochs',fontweight='bold')
axes[0].set_ylabel('Loss',fontweight='bold')
axes[1].set_xlabel('Epochs',fontweight='bold')
axes[1].set_ylabel('Accuracy',fontweight='bold')
axes[0].set_title('CNN With Inception Training and \n Testing Loss vs.␣
 ↪Epochs',fontsize=16,fontweight='bold')
axes[1].set_title('CNN With Inception Training and \n Testing Accuracy vs.␣
 ↪Epochs',fontsize=16,fontweight='bold')
axes[0].legend()
axes[1].legend()
plt.show()
```



7.4.2 Visualizing True and Predicted Class with CNN Inception Model:

This barplot displays the true and predicted class labels, where Inception with CNN Model seems to have more false positives in Category 0. Other categories, seems to have less error in prediction.

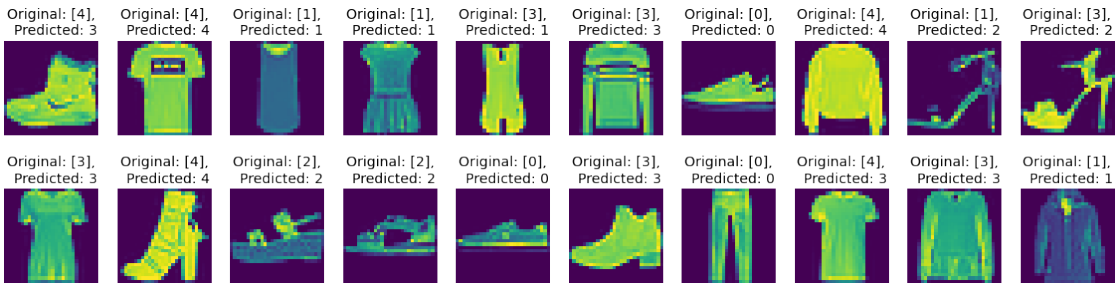
[22]:
```
cnn_inception_true_pred= true_pred(cnn_inception_model,'Inception CNN Model')
```



7.4.3 True and Predicted visualization of 20 images of Inception CNN Model:

The below plot displays the actual 20 images with their original category and their predicted ones. Here, there are 5 image categories which are predicted wrong.

[23]:
```
# True and Predicted visualization of 20 images
cnn_inception_visualization= Vis_true_pred(cnn_inception_model,20)
```



8 Resnet50 CNN Model

8.1 CM[7]

8.2 Part 1 - 3rd Model(Resnet50)

8.2.1 Runtime Performance of Training and Testing of CNN Resnet50 Model:

In Runtime performance , the training data achieved accuracy of 0.9462 and loss of 0.1469 whereas , the test data achieved accuracy of 0.8892 and loss of 0.3696, with 30 epochs.

[26]:
```
resnet50_history= resnet50_model.
↪fit(x_train,y_train,validation_data=(x_test,y_test),epochs=30,batch_size=128,verbose=1)
```

```
Epoch 1/30
469/469 [==============================] - 62s 94ms/step - loss: 1.1654 -
accuracy: 0.5490 - val_loss: 0.8591 - val_accuracy: 0.6607
Epoch 2/30
469/469 [==============================] - 41s 87ms/step - loss: 0.5380 -
accuracy: 0.8034 - val_loss: 0.4147 - val_accuracy: 0.8365
Epoch 3/30
469/469 [==============================] - 40s 86ms/step - loss: 0.4558 -
accuracy: 0.8314 - val_loss: 0.6165 - val_accuracy: 0.7824
Epoch 4/30
469/469 [==============================] - 40s 84ms/step - loss: 0.4239 -
accuracy: 0.8445 - val_loss: 0.5968 - val_accuracy: 0.7779
Epoch 5/30
469/469 [==============================] - 39s 83ms/step - loss: 0.3949 -
accuracy: 0.8536 - val_loss: 0.3908 - val_accuracy: 0.8518
Epoch 6/30
469/469 [==============================] - 39s 84ms/step - loss: 0.3798 -
accuracy: 0.8593 - val_loss: 0.8141 - val_accuracy: 0.7379
Epoch 7/30
469/469 [==============================] - 39s 84ms/step - loss: 0.3748 -
accuracy: 0.8612 - val_loss: 0.3794 - val_accuracy: 0.8474
Epoch 8/30
469/469 [==============================] - 39s 83ms/step - loss: 0.3409 -
accuracy: 0.8723 - val_loss: 0.3451 - val_accuracy: 0.8601
Epoch 9/30
469/469 [==============================] - 39s 83ms/step - loss: 0.3363 -
accuracy: 0.8733 - val_loss: 0.4058 - val_accuracy: 0.8438
Epoch 10/30
469/469 [==============================] - 39s 83ms/step - loss: 0.3294 -
accuracy: 0.8791 - val_loss: 0.3594 - val_accuracy: 0.8604
Epoch 11/30
469/469 [==============================] - 39s 83ms/step - loss: 0.3232 -
accuracy: 0.8803 - val_loss: 0.5182 - val_accuracy: 0.8267
Epoch 12/30
469/469 [==============================] - 39s 83ms/step - loss: 0.3028 -
accuracy: 0.8880 - val_loss: 0.3783 - val_accuracy: 0.8517
Epoch 13/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2917 -
accuracy: 0.8899 - val_loss: 0.5031 - val_accuracy: 0.8215
Epoch 14/30
469/469 [==============================] - 39s 84ms/step - loss: 0.2922 -
accuracy: 0.8920 - val_loss: 0.3195 - val_accuracy: 0.8709
Epoch 15/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2881 -
accuracy: 0.8938 - val_loss: 0.3887 - val_accuracy: 0.8623
Epoch 16/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2821 -
accuracy: 0.8938 - val_loss: 0.4901 - val_accuracy: 0.8295
```

```
Epoch 17/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2611 -
accuracy: 0.9034 - val_loss: 0.3219 - val_accuracy: 0.8813
Epoch 18/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2725 -
accuracy: 0.8966 - val_loss: 0.3461 - val_accuracy: 0.8770
Epoch 19/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2505 -
accuracy: 0.9076 - val_loss: 0.3740 - val_accuracy: 0.8692
Epoch 20/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2280 -
accuracy: 0.9152 - val_loss: 0.4464 - val_accuracy: 0.8483
Epoch 21/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2300 -
accuracy: 0.9179 - val_loss: 0.3688 - val_accuracy: 0.8706
Epoch 22/30
469/469 [==============================] - 39s 83ms/step - loss: 0.2143 -
accuracy: 0.9199 - val_loss: 0.3405 - val_accuracy: 0.8786
Epoch 23/30
469/469 [==============================] - 39s 83ms/step - loss: 0.1986 -
accuracy: 0.9268 - val_loss: 0.4170 - val_accuracy: 0.8437
Epoch 24/30
469/469 [==============================] - 39s 83ms/step - loss: 0.1931 -
accuracy: 0.9295 - val_loss: 0.3533 - val_accuracy: 0.8775
Epoch 25/30
469/469 [==============================] - 39s 83ms/step - loss: 0.1937 -
accuracy: 0.9283 - val_loss: 0.3977 - val_accuracy: 0.8667
Epoch 26/30
469/469 [==============================] - 39s 83ms/step - loss: 0.1781 -
accuracy: 0.9349 - val_loss: 0.3592 - val_accuracy: 0.8754
Epoch 27/30
469/469 [==============================] - 39s 83ms/step - loss: 0.1630 -
accuracy: 0.9396 - val_loss: 0.5776 - val_accuracy: 0.8274
Epoch 28/30
469/469 [==============================] - 39s 83ms/step - loss: 0.1779 -
accuracy: 0.9349 - val_loss: 0.4123 - val_accuracy: 0.8641
Epoch 29/30
469/469 [==============================] - 39s 84ms/step - loss: 0.1606 -
accuracy: 0.9430 - val_loss: 0.3654 - val_accuracy: 0.8787
Epoch 30/30
469/469 [==============================] - 39s 83ms/step - loss: 0.1469 -
accuracy: 0.9462 - val_loss: 0.3696 - val_accuracy: 0.8892
```

8.2.2 Evaluation of Testing with CNN Resnet50 Model:

[27]:
```
resnet50_score=resnet50_model.evaluate(x_test,y_test,verbose=0)
print('The loss while evaluation of test data with Resnet50 CNN is: {} '.
 ↪format(resnet50_score[0]))
print('The accuracy while evaluation of test data with Resnet50 CNN is: {} '.
 ↪format(resnet50_score[1]))
```

```
The loss while evaluation of test data with Resnet50 CNN is: 0.3696388602256775
The accuracy while evaluation of test data with Resnet50 CNN is:
0.88919997215271
```

8.3 CM[7]

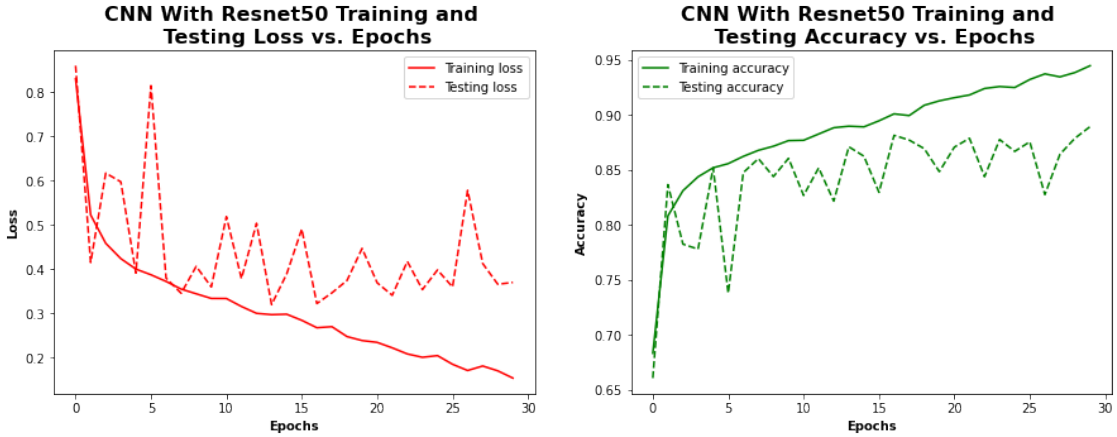
8.4 Part 3 - 3rd Model(Resnet50)

8.4.1 Plotting of Loss and Accuracy vs. Epochs of CNN Resnet50 Model:

Here,the below plots shows the accuracy and loss rate with respect to epochs, where, training set accuracy keeps on increasing and loss is decreasing, but in test set, there seems to be very slight increase and it almost fluctuates in same level. Even after applying the batchnormalization on various layers, test set training has heavy fluctuations . This may be due to some elements in the dataset are randomly classified. Also, as our test set accuracy in not increasing and its loss is not decreasing significantly, but slightly , almost seems like fluctuating at same level. So, overfitting can be problem, but not a major one and if it so, then the dropout layers are already added. Apart from it, It can be due to less data points which we cannot modify . Although, after trying many hyperparameters and adding dense layers , this is the best model achieved in tems of accuracy, but not in terms of good fit. It seems after some epochs the test data points were geting memorized instead of getting learned by model. Also, as this is resnet model so, vanishing gradient problem is already handled within the appropriate structured layers. Also, to avoid more overfitting, the model is stopped early at 30 epochs.

[28]:
```
fig, axes= plt.subplots(1,2,figsize=(15,5))
axes[0].plot(resnet50_history.history['loss'],label='Training loss',color='red')
axes[0].plot(resnet50_history.history['val_loss'],label='Testing␣
 ↪loss',color='red',linestyle ='dashed')
axes[1].plot(resnet50_history.history['accuracy'],label='Training␣
 ↪accuracy',color='green')
axes[1].plot(resnet50_history.history['val_accuracy'],label='Testing␣
 ↪accuracy',color='green',linestyle ='dashed')
axes[0].set_xlabel('Epochs',fontweight='bold')
axes[0].set_ylabel('Loss',fontweight='bold')
axes[1].set_xlabel('Epochs',fontweight='bold')
axes[1].set_ylabel('Accuracy',fontweight='bold')
axes[0].set_title('CNN With Resnet50 Training and \n Testing Loss vs.␣
 ↪Epochs',fontsize=16,fontweight='bold')
axes[1].set_title('CNN With Resnet50 Training and \n Testing Accuracy vs.␣
 ↪Epochs',fontsize=16,fontweight='bold')
axes[0].legend()
```

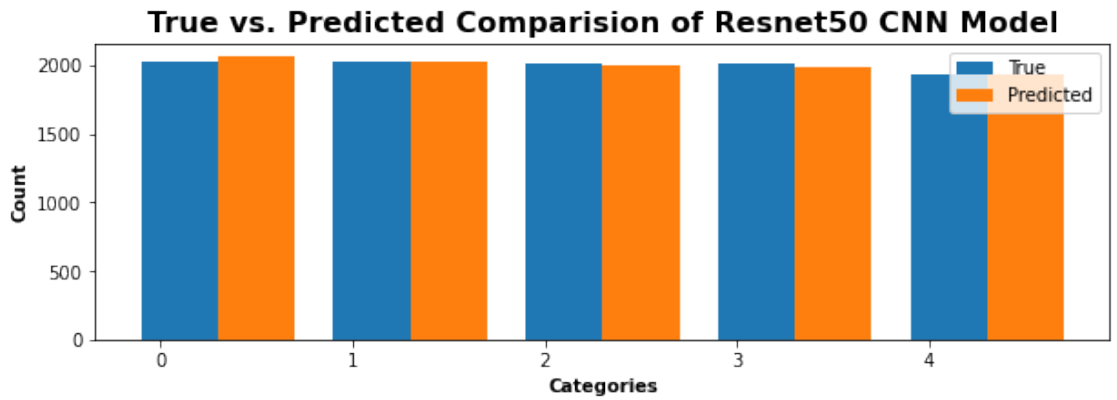
```
axes[1].legend()
plt.show()
```



8.4.2 Visulaizing True and Predicted Class with CNN Resnet50 Model:

This barplot displays the true and predicted class labels, where Resnet50 seems to have almost similar and very small error in prediction in all categories.

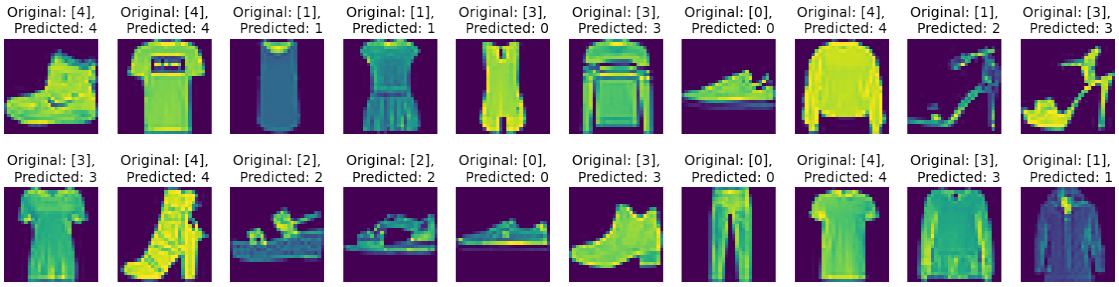
[29]:
```
resnet50_true_pred= true_pred(resnet50_model,'Resnet50 CNN Model')
```



8.4.3 True and Predicted visualization of 20 images of Resnet50 CNN Model:

The below plot displays the actual 20 images with their original category and their predicted ones. Here, there 2 Image categories which are predicted wrong.

[30]:
```
# True and Predicted visualization of 20 images
resnet50_model_visualization= Vis_true_pred(resnet50_model,20)
```



9 CM[7]

10 Part 2 - Comparision of Different Models

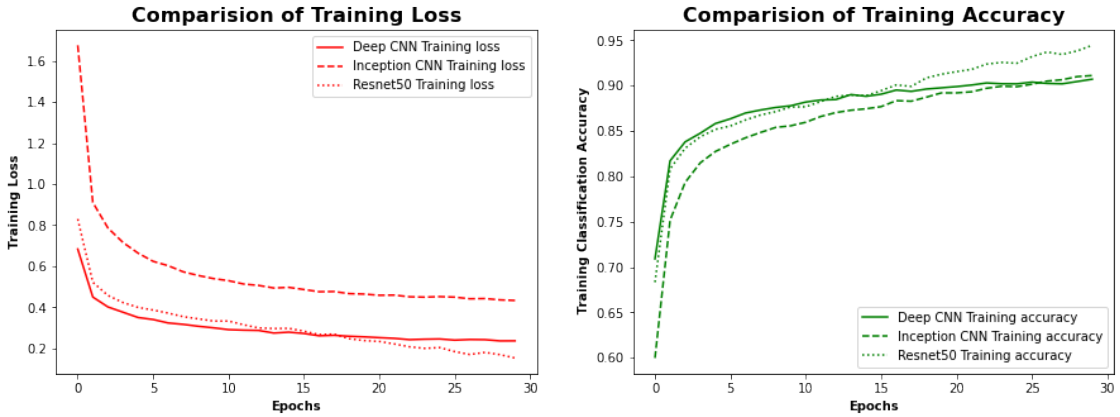
10.0.1 1) Comparision of Training Loss and Accuracy among Different Models:

In training data, - The Deep CNN achieved accuracy is 0.9093 and loss is 0.23. - The Deep CNN achieved accuracy is 0.9197 and loss is 0.42. - The Deep CNN achieved accuracy is 0.946 and loss is 0.146.

So, in case of training data, the Resnet50 model achieved maximum accuracy and minimum loss of 0.1 among all models. This is achieved as, among all models, Resnet50 seems to show more overfitting than Inception and simple Deep CNN. Due to good fit, Deep CNN gave almost same accuracies and loss of as test data, which tells that models is learning both training and testing at same pace and not memorizing , unlike other overfitted methods. Therefore, just depensing upon training accuracy and loss, Resnet50 performed better and Deep CNN performed bad, comparitively.

[31]:
```
fig, axes= plt.subplots(1,2,figsize=(15,5))
axes[0].plot(deep_cnn_history.history['loss'],label='Deep CNN Training␣
 ↪loss',color='red')
axes[0].plot(cnn_inception_history.history['loss'],label='Inception CNN␣
 ↪Training loss',color='red',linestyle ='dashed')
axes[0].plot(resnet50_history.history['loss'],label='Resnet50 Training␣
 ↪loss',color='red',linestyle ='dotted')
axes[1].plot(deep_cnn_history.history['accuracy'],label='Deep CNN Training␣
 ↪accuracy',color='green')
axes[1].plot(cnn_inception_history.history['accuracy'],label='Inception CNN␣
 ↪Training accuracy',color='green',linestyle ='dashed')
axes[1].plot(resnet50_history.history['accuracy'],label='Resnet50 Training␣
 ↪accuracy',color='green',linestyle ='dotted')
axes[0].set_xlabel('Epochs',fontweight='bold')
axes[0].set_ylabel('Training Loss',fontweight='bold')
axes[1].set_xlabel('Epochs',fontweight='bold')
axes[1].set_ylabel(' Training Classification Accuracy',fontweight='bold')
axes[0].set_title('Comparision of Training Loss',fontsize=16,fontweight='bold')
```

```
axes[1].set_title('Comparision of Training␣
 ↪Accuracy',fontsize=16,fontweight='bold')
axes[0].legend()
axes[1].legend()
plt.show()
```



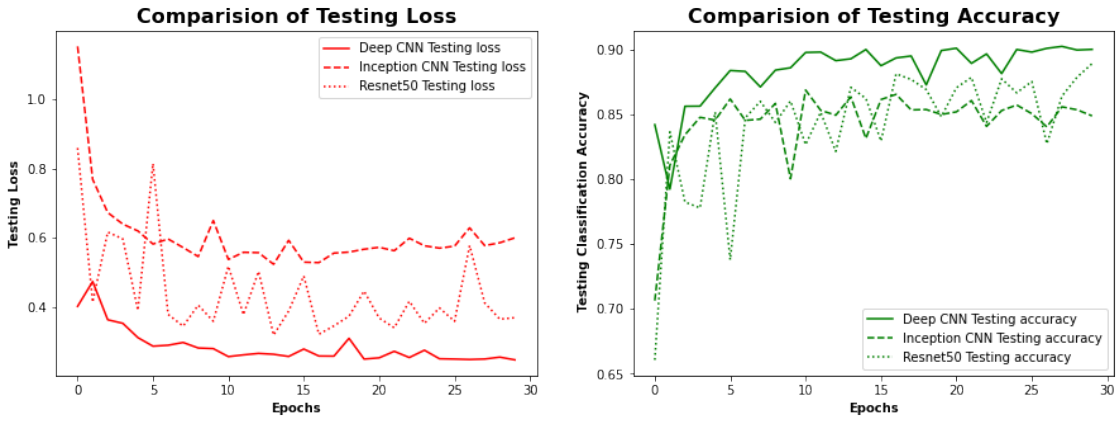
10.0.2 2) Comparision Testing Loss and Accuracy among Different Models:

In test data, - Deep CNN achieved accuracy of 0.90 and loss of 0.2 - Inception CNN achieved accuracy of 0.84 and loss of 0.6 - Resnet CNN achieved accuracy of 0.88 and loss of 0.369.

So, Deep CNN model seems to have gained better accuracy and loss as compared to others , also because of its good fit. Wheraes, the other models gave less accuracy and more loss due to overfitting in which Inception gave worse accuracy and loss, among all three models. Although, from graph it looks like Resnet50 model suffers from more overfitting as compared to Inception , but may be due to more layers and good structure , it machieved better accuracy than Inception. In Inception , it seems that model stopped learning earlier than Resnet50 model. But, Deep CNN provided better performance in case of testing accuracy and loss.

[32]:
```
fig, axes= plt.subplots(1,2,figsize=(15,5))
axes[0].plot(deep_cnn_history.history['val_loss'],label='Deep CNN Testing␣
 ↪loss',color='red')
axes[0].plot(cnn_inception_history.history['val_loss'],label='Inception CNN␣
 ↪Testing loss',color='red',linestyle ='dashed')
axes[0].plot(resnet50_history.history['val_loss'],label='Resnet50 Testing␣
 ↪loss',color='red',linestyle ='dotted')
axes[1].plot(deep_cnn_history.history['val_accuracy'],label='Deep CNN Testing␣
 ↪accuracy',color='green')
axes[1].plot(cnn_inception_history.history['val_accuracy'],label='Inception CNN␣
 ↪Testing accuracy',color='green',linestyle ='dashed')
axes[1].plot(resnet50_history.history['val_accuracy'],label='Resnet50 Testing␣
 ↪accuracy',color='green',linestyle ='dotted')
```

```
axes[0].set_xlabel('Epochs',fontweight='bold')
axes[0].set_ylabel('Testing Loss',fontweight='bold')
axes[1].set_xlabel('Epochs',fontweight='bold')
axes[1].set_ylabel(' Testing Classification Accuracy',fontweight='bold')
axes[0].set_title('Comparision of Testing Loss',fontsize=16,fontweight='bold')
axes[1].set_title('Comparision of Testing␣
 ↪Accuracy',fontsize=16,fontweight='bold')
axes[0].legend()
axes[1].legend()
plt.show()
```



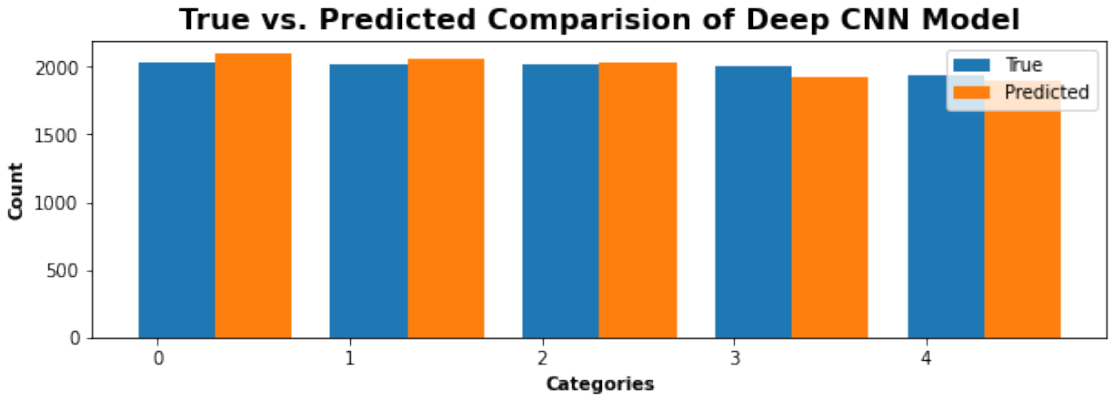
10.0.3 3) Based on True and Predicted labels plot:

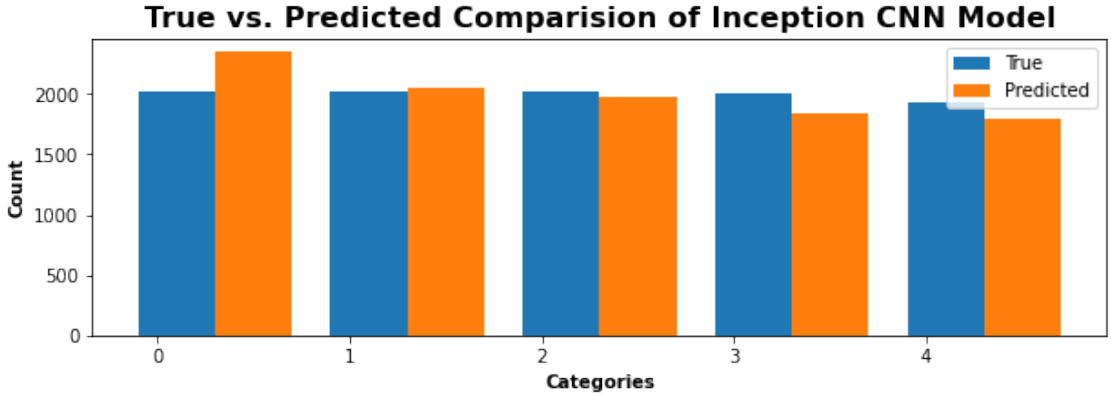
From Below barplots , it is visible that

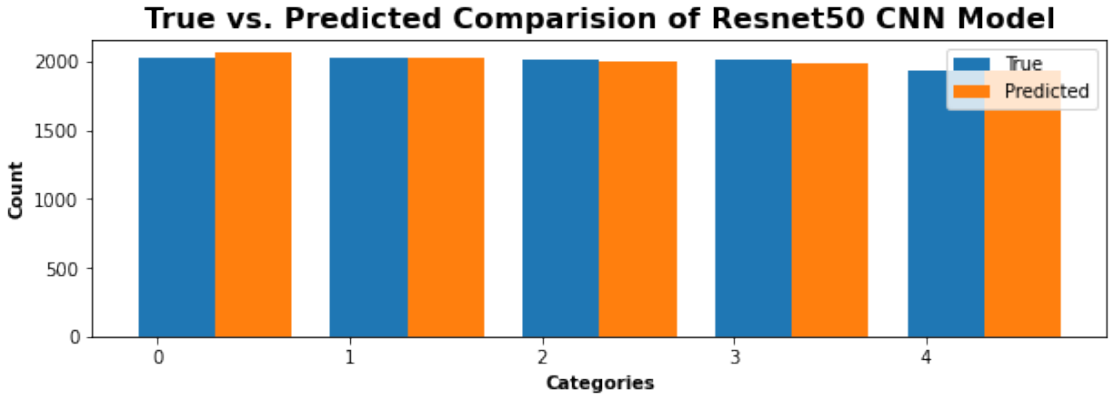
- Inception CNN model predicted more false positives in category 0.
- Deep CNN model predicted slightly more error in category 0 and 3 as compared to other labels.
- Whereas, Resnet50 model seems to have almost similar predictions in all categories.

So, In correctly classifying the label, Resnet50 performed better and Inception CNN performed worse, comparitively.

[36]:
```
# calling True and Predicted label of Deep CNN Model
true_pred(cnn_layers_model,'Deep CNN Model')
# calling True and Predicted label of Inception CNN Model
true_pred(cnn_inception_model,'Inception CNN Model')
# calling True and Predicted label of Resnet50 CNN Model
true_pred(resnet50_model,'Resnet50 CNN Model')
```


True vs. Predicted Comparision of Deep CNN Model
True
Predicted
Count
0
500
1000
1500
2000
0
1
2
3
4
Categories


True vs. Predicted Comparision of Inception CNN Model
True
Predicted
Count
0
500
1000
1500
2000
0
1
2
3
4
Categories


True vs. Predicted Comparision of Resnet50 CNN Model
True
Predicted
Count
0
500
1000
1500
2000
0
1
2
3
4
Categories

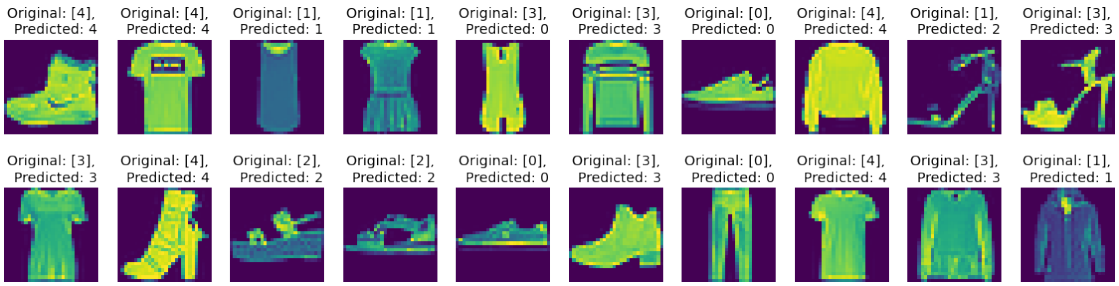
10.0.4 4) Based on correctly classified original and predicted labels of 20 sample images:

below are the 20 Images generated frokm test set with their true and predicted categories where, - Inception CNN has predicted 5 wrong labels. - Resnet50 CNN has predicted 2 wrong labels. - Surprisingly, simple Deep CNN has predicted no wrong labels.

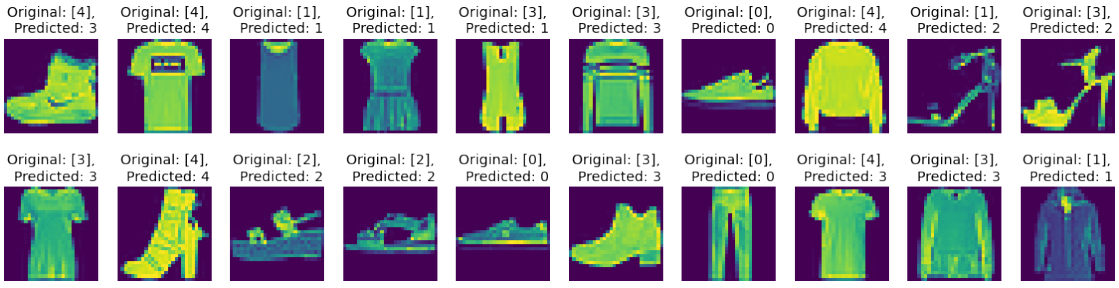
So, among the given set of images , simple Deep CNN has outperformed all and agin, Inception performed worse in this case also.

[37]:
```
# calling original and predicetd label with 20 images of Deep CNN Model
print('For Deep CNN:')
Vis_true_pred(resnet50_model,20)
# calling original and predicetd label with 20 images of Inception CNN Model
print('For CNN Inception:')
Vis_true_pred(cnn_inception_model,20)
# calling original and predicetd label with 20 images of Resnet50 CNN Model
print('For CNN with Resnet50:')
Vis_true_pred(resnet50_model,20)
```

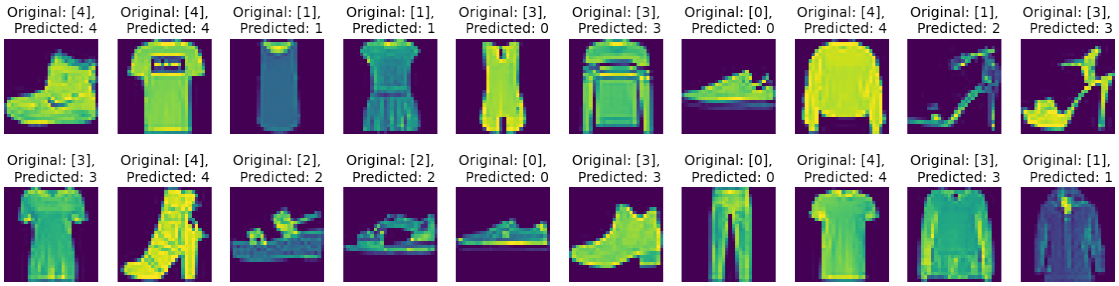
```
For Deep CNN:
```




```
For CNN Inception:
```




```
For CNN with Resnet50:
```



10.0.5 Conclusion:

- The Resnet50 model provided better training accuracy of around 94% and loss of 0.1.
- The Deep CNN model provided better testing accuracy of about 90% and loss of 0.2.
- The best fit model among all is also simple implemented Deep learning model.
- Resnet50 seens to have suffered more overfitting.
- Inception model does not provided good results both in case of accuracy and fit.
- so, for our dataset, Deep CNN is best suitable.

10.0.6 References:

- https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/
- https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/
- https://machinelearningmastery.com/how-to-implement-major-architecture-innovations-for-convolutional-neural-networks/
- https://maelfabien.github.io/deeplearning/inception/#what-is-an-inception-module
- https://datascience-enthusiast.com/DL/Residual_Networks_v2.html