ECE 657 ASSIGNMENT 3: Problem 1

CNN

Harnoor Singh: 20870613
Jubilee Imhanzenobe: 20809735
Olohireme Ajayi: 20869827

In [1]:
```
# importing libraries
# !pip install -q -U tensorflow>=1.8.0
!pip install keras-visualizer
import tensorflow as tf
import keras
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os, time
```

```
Requirement already satisfied: keras-visualizer in /usr/local/lib/python3.7/dist
-packages (2.4)
```

Function for plotting curves

In [2]:
```
def accuracy_loss_plot(model):
  """ function that plots training and validation curves """
  hist = model.history
  acc = hist['accuracy']
  val_acc = hist['val_accuracy']
  loss = hist['loss']
  val_loss = hist['val_loss']
  epoch = range(1, 6)                # Number of epochs is 5

  fig = plt.figure(figsize = (12,10))
  fig.tight_layout(pad=7.0)
  plt.subplot(2,2,1)
  plt.plot(acc,loss)
  plt.xlabel('Accuracy')
  plt.ylabel('Loss')
  plt.legend(['train set'], loc='upper right')
  plt.title('Training Accuracy vs Loss')

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

  plt.subplot(2,2,3)
  plt.plot(epoch, acc)
  plt.plot(epoch, val_acc)
  plt.xlabel('Epoch')
```

```
    plt.xticks(range(1, 6))
    plt.ylabel('Accuracy')
    plt.legend(['train set', 'validation set'], loc='lower right')
    plt.title('Accuracy vs Epoch')

    plt.subplot(2,2,4)
    plt.plot(epoch, loss)
    plt.plot(epoch, val_loss)
    plt.xlabel('Epoch')
    plt.xticks(range(1, 6))
    plt.ylabel('Loss')
    plt.legend(['train set', 'validation set'], loc='upper right')
    plt.title('Loss vs Epoch')

    plt.show()
```

In [3]:

```
def compare_models(history_list, labels, epochs = 5):
    """ Function for plotting the accuracy and loss for training and validation
    fig, axs = plt.subplots(2, 2, figsize=(12,8))
    count = 0
    epochs = list(range(1, epochs + 1))
    final_train = []
    final_val = []
    final_loss = []
    final_val_loss = []
    for history in history_list:
        label = labels[count]
        val_accuracy = history.history['val_accuracy']
        val_loss = history.history['val_loss']
        train_accuracy = history.history['accuracy']
        train_loss = history.history['loss']
        axs[0,0].plot(epochs, train_accuracy, label= label)
        axs[1,0].plot(epochs, train_loss, label=label)
        axs[0,1].plot(epochs, val_accuracy, label= label)
        axs[1,1].plot(epochs, val_loss, label=label)
        count += 1

        final_train.append(round(history.history['accuracy'][-1] * 100, 2))
        final_val.append(round(history.history['loss'][-1], 4))
        final_loss.append(round(history.history['val_accuracy'][-1] * 100, 2))
        final_val_loss.append(round(history.history['val_loss'][-1], 4))

    axs[0,0].set_ylabel('Training accuracy')
    axs[1,0].set_ylabel('Training loss')
    axs[0,1].set_ylabel('Validation accuracy')
    axs[1,1].set_ylabel('Validation loss')
    axs[0,0].set_xlabel('Epochs')
    axs[1,0].set_xlabel('Epochs')
    axs[0,1].set_xlabel('Epochs')
    axs[1,1].set_xlabel('Epochs')

    axs[0,0].legend(loc='upper center', bbox_to_anchor=(1.1, -1.4),
              ncol=4, fancybox=True, shadow=True)

    columns = ["Train Accuracy", "Train Loss", "Val Accuracy", "Val Loss"]
    result = pd.DataFrame(np.array([final_train, final_val, final_loss, final_va
    return result
```

Preparing our dataset: Loading and Preprocessing

1. PREPROCESSING STEPS -

- Randomly sampling 20% of the training set, and using that as new training set.
- Using the test set for validation.
- Normalizing the data by scaling pixels in the range 0-1.
- One-hot encoding the labels.
- Reshaping the train set and validation set for feeding them to the MLP.

In [4]:
```
# downloading and loading the dataset
from keras.datasets import cifar10
from sklearn.model_selection import train_test_split

(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Randomly sampling 20% of the training set and using that as new training set
X_train, X_test2, y_train, y_test2 = train_test_split(X_train, y_train, train_si

# Using the test set for validation
X_val, y_val = X_test, y_test
```

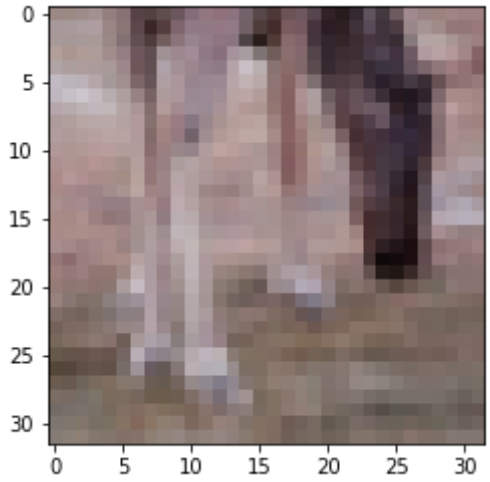
In [5]:
```
# normalizing the data
X_train = X_train / 255.0
X_val = X_val / 255.0
```

In [6]:
```
print('X_train shape: ', X_train.shape)
print('y_train shape: ', y_train.shape)
print('X_val shape: ', X_val.shape)
print('y_val shape: ', y_val.shape)
```

```
X_train shape:  (10000, 32, 32, 3)
y_train shape:  (10000, 1)
X_val shape:  (10000, 32, 32, 3)
y_val shape:  (10000, 1)
```

In [7]:
```
# one-hot encoding the labels
from keras.utils.np_utils import to_categorical
y_train = to_categorical(y_train, 10)
y_val = to_categorical(y_val, 10)
```

In [8]:
```
# printing one training sample
plt.imshow(X_train[1])
plt.show()
```



2. DESCRIPTION OF OUTPUT LAYER AND LOSS FUNCTION -

Number of Units - There should be 'n' neurons in the output layer if we are classifying between 'n' categories/classes. Since the number of classes are 10 (10 outputs to the classification problem), the number of nodes in the output layer is 10.

Activation Function - The activation function used in the output layer is Softmax. A Softmax function is a type of squashing function. Squashing functions limit the output of the function into the range 0 to 1. This allows the output to be interpreted directly as a probability. Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. Softmax assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would. Since our dataset has 10 classes, we are using softmax. Sigmoid and Softmax both give output in [0,1] range but softmax ensures that the sum of outputs is 1 i.e., they are probabilities. Sigmoid just makes output between 0 to 1, the sum of probabilities might not be 1 and it works better for binary classification problem. More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.

Loss Function - The choice of loss function is directly related to the activation function used in the output layer of the neural network. These two design elements are connected. For regression problems, functions like Mean Squared Error (MSE) and Mean Absolute Error (MAE) are used since we predict a real-value quantity. For classification problems, cross-entropy loss functions are used. Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. Since our problem is multi-class classification, and we are using softmax activation in the output layer, we can use either categorical cross entropy or sparse categorical cross entropy. Both, categorical cross entropy, and sparse categorical cross entropy have the same loss function. The only difference is the format in which we mention the true labels. If true-labels are one-hot encoded, we use

categorical_crossentropy. But if labels are integers, we use sparse_categorical_crossentropy. Our labels are one-hot encoded, hence the use of categorical cross-entropy.

Model definition and training: MLP

In [9]:
```
# reshaping data for MLP
X_train_MLP = X_train.reshape(-1, 3072)
X_val_MLP = X_val.reshape(-1, 3072)
```

In [10]:
```
# batch size and number of epochs
EPOCHS = 5
BATCH_SIZE = 32
```

In [11]:
```
# shapes of train and validation sets that are to be fed to MLP
print('X_train_MLP shape: ', X_train_MLP.shape)
print('X_val_MLP shape: ', X_val_MLP.shape)
```

```
X_train_MLP shape:  (10000, 3072)
X_val_MLP shape:  (10000, 3072)
```

In [12]:
```
# importing libraries
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras import activations
from tensorflow.keras.optimizers import Adam

MLP = Sequential()
MLP.add(Dense(512, activation='sigmoid', input_shape=(3072, )))
MLP.add(Dense(512, activation='sigmoid'))
MLP.add(Dense(10,  activation='softmax'))

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

In [13]:
```
MLP.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               1573376
_________________________________________________________________
dense_1 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,841,162
Trainable params: 1,841,162
Non-trainable params: 0
_________________________________________________________________
```

In [14]:

```
# Train the model
start_time = time.time()
with tf.device('/device:GPU:0'):
    train_MLP = MLP.fit(X_train_MLP, y_train, batch_size=BATCH_SIZE,
                        epochs=EPOCHS, verbose=1, validation_data=(X_val_MLP, y_val)
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0
```

```
Epoch 1/5
313/313 [==============================] - 14s 8ms/step - loss: 2.1867 - accurac
y: 0.1980 - val_loss: 1.9364 - val_accuracy: 0.2650
Epoch 2/5
313/313 [==============================] - 2s 5ms/step - loss: 1.9111 - accurac
y: 0.3075 - val_loss: 1.8711 - val_accuracy: 0.3309
Epoch 3/5
313/313 [==============================] - 2s 5ms/step - loss: 1.8645 - accurac
y: 0.3207 - val_loss: 1.8145 - val_accuracy: 0.3412
Epoch 4/5
313/313 [==============================] - 2s 5ms/step - loss: 1.8059 - accurac
y: 0.3457 - val_loss: 1.8214 - val_accuracy: 0.3388
Epoch 5/5
313/313 [==============================] - 2s 7ms/step - loss: 1.7588 - accurac
y: 0.3559 - val_loss: 1.8008 - val_accuracy: 0.3388
Total training time : 0.36 minute
```

In [15]:

```
# training accuracy and loss for MLP
score_train_MLP = MLP.evaluate(X_train_MLP, y_train, verbose=0)
print('Train accuracy for MLP:', score_train_MLP[1])
print('Train loss for MLP:', score_train_MLP[0])
```

```
Train accuracy for MLP: 0.35429999232292175
Train loss for MLP: 1.7643574476242065
```

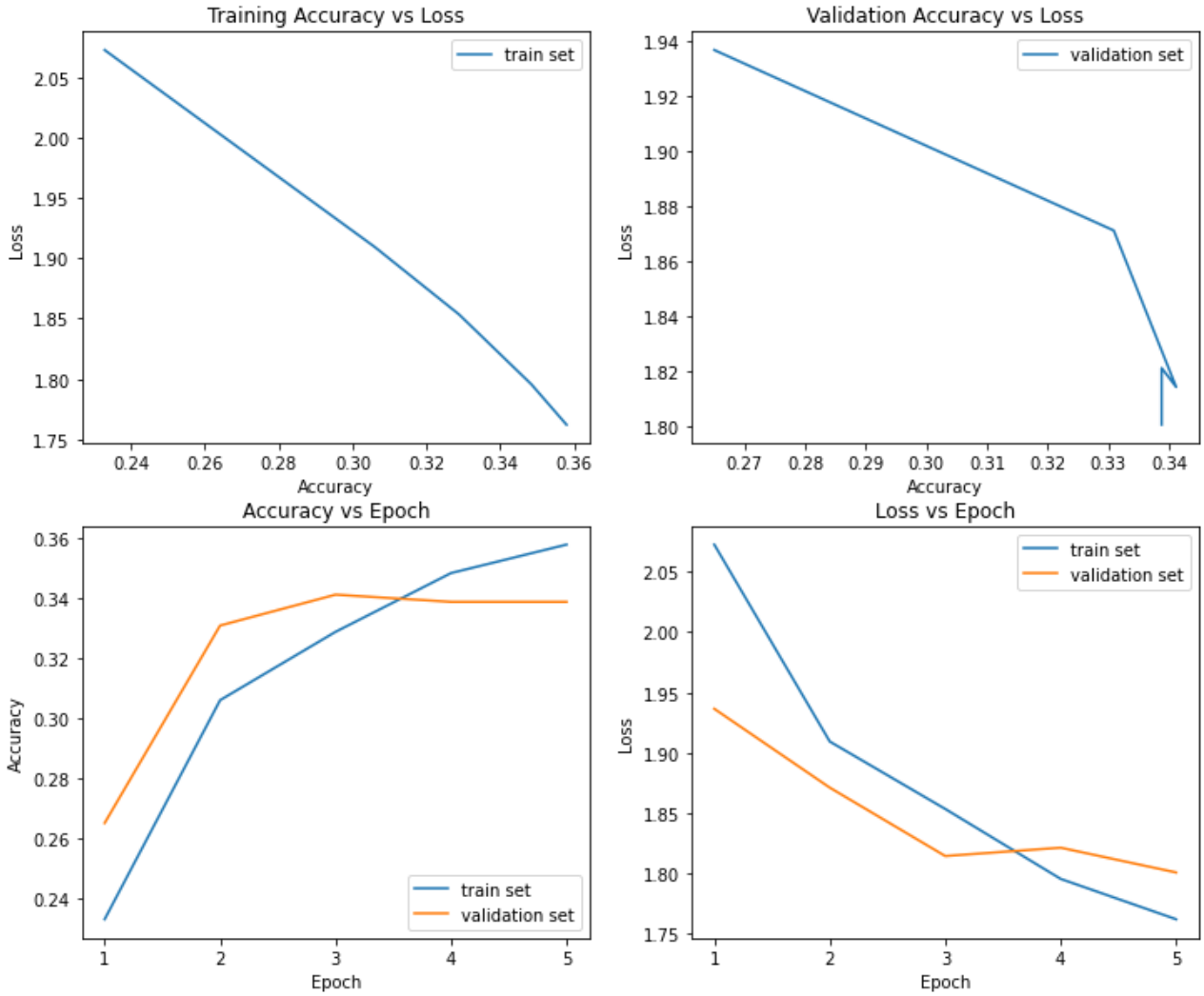
In [16]:

```
# validation accuracy and loss for MLP
score_val_MLP = MLP.evaluate(X_val_MLP, y_val, verbose=0)
print('Validation accuracy for MLP:', score_val_MLP[1])
print('Validation loss for MLP:', score_val_MLP[0])
```

```
Validation accuracy for MLP: 0.33880001306533813
Validation loss for MLP: 1.800845742225647
```

In [17]:

```
accuracy_loss_plot(train_MLP)
```



Model definition and training: CNN1

In [18]:
```
# importing libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Dense
from tensorflow.keras import backend as K
from tensorflow.keras.optimizers import Adam, SGD
from keras_visualizer import visualizer
```

In [19]:
```
CNN1 = Sequential()
CNN1.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
                input_shape=(32, 32, 3)))
CNN1.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
CNN1.add(Flatten())
CNN1.add(Dense(512, activation='sigmoid'))
CNN1.add(Dense(512, activation='sigmoid'))
CNN1.add(Dense(10, activation='softmax'))
```

```
CNN1.compile(loss='categorical_crossentropy', optimizer=Adam(),
             metrics=['accuracy'])
```

In [20]:
```
CNN1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 64)        1792
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 64)        36928
_________________________________________________________________
flatten (Flatten)            (None, 50176)             0
_________________________________________________________________
dense (Dense)                (None, 512)               25690624
_________________________________________________________________
dense_1 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 25,997,130
Trainable params: 25,997,130
Non-trainable params: 0
_________________________________________________________________
```

In [21]:
```
# Train the the model
start_time = time.time()
with tf.device('/device:GPU:0'):
    train_CNN1 = CNN1.fit(X_train, y_train, batch_size=BATCH_SIZE,
                          epochs=EPOCHS, verbose=1, validation_data=(X_val, y_val))
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0
```

```
Epoch 1/5
313/313 [==============================] - 7s 16ms/step - loss: 1.7629 - accurac
y: 0.3509 - val_loss: 1.4514 - val_accuracy: 0.4608
Epoch 2/5
313/313 [==============================] - 4s 13ms/step - loss: 1.3392 - accurac
y: 0.5168 - val_loss: 1.3624 - val_accuracy: 0.5064
Epoch 3/5
313/313 [==============================] - 5s 15ms/step - loss: 1.0393 - accurac
y: 0.6306 - val_loss: 1.3153 - val_accuracy: 0.5331
Epoch 4/5
313/313 [==============================] - 5s 15ms/step - loss: 0.6606 - accurac
y: 0.7761 - val_loss: 1.3813 - val_accuracy: 0.5472
Epoch 5/5
313/313 [==============================] - 4s 13ms/step - loss: 0.2962 - accurac
y: 0.9115 - val_loss: 1.5696 - val_accuracy: 0.5391
Total training time : 0.41 minute
```

In [22]:
```
# training accuracy and loss for CNN1
score_train_CNN1 = CNN1.evaluate(X_train, y_train, verbose=1)
print('Train accuracy for CNN1:', score_train_CNN1[1])
print('Train loss for CNN1:', score_train_CNN1[0])
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.1565 - accurac
```

```
y: 0.9648
Train accuracy for CNN1: 0.9648000001907349
Train loss for CNN1: 0.1564616709947586
```

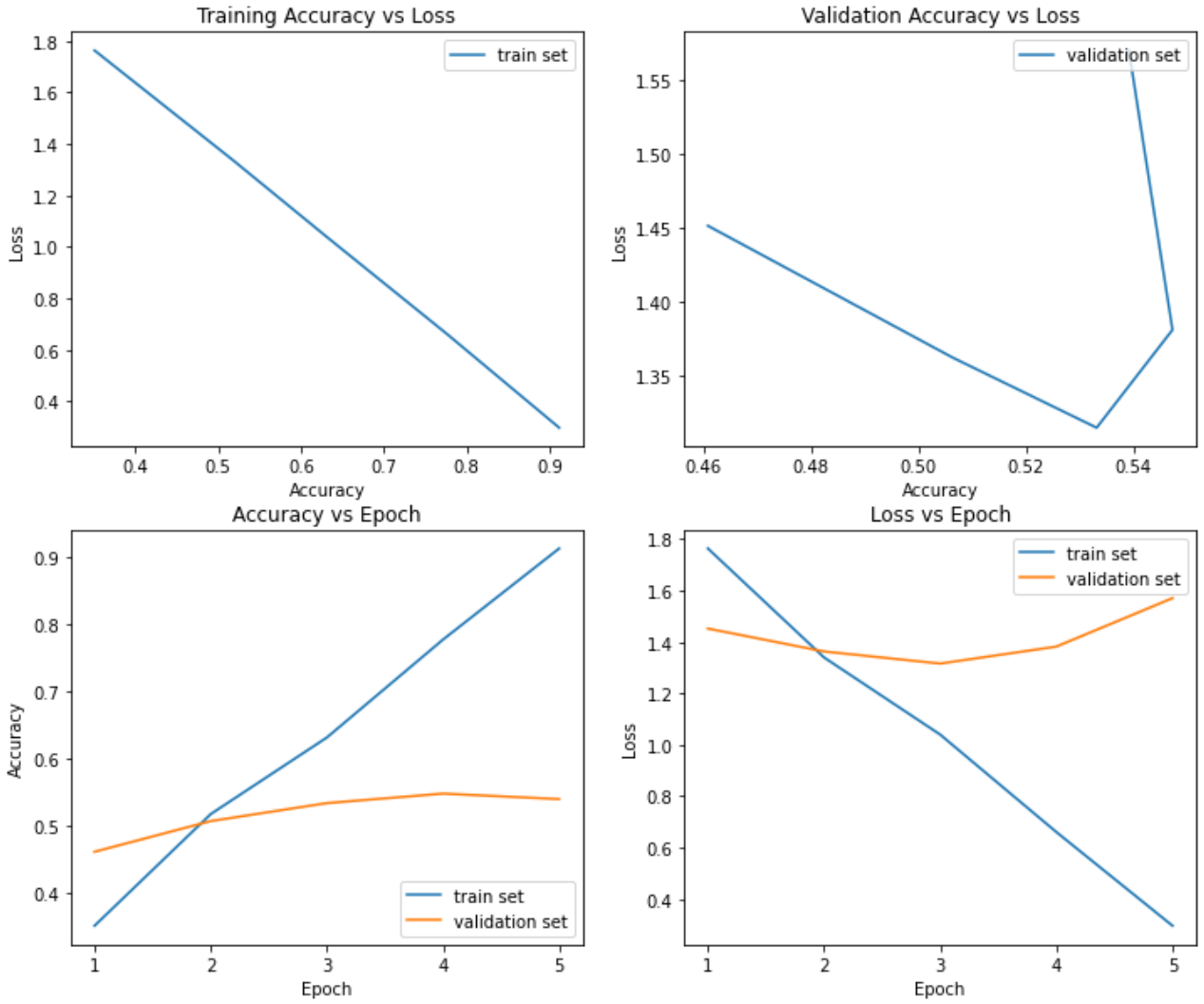
In [23]:

```
# validation accuracy and loss for CNN1
score_val_CNN1 = CNN1.evaluate(X_val, y_val, verbose=1)
print('Validation accuracy for CNN1:', score_val_CNN1[1])
print('Validation loss for CNN1:', score_val_CNN1[0])
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.5696 - accurac
y: 0.5391
Validation accuracy for CNN1: 0.5390999913215637
Validation loss for CNN1: 1.5696282386779785
```

In [24]:

```
accuracy_loss_plot(train_CNN1)
```



Model definition and training: CNN2

In [25]:

```
CNN2 = Sequential()
CNN2.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
                input_shape=(32, 32, 3)))
CNN2.add(MaxPooling2D((2, 2)))
CNN2.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
```

```
CNN2.add(MaxPooling2D((2, 2)))
CNN2.add(Flatten())
CNN2.add(Dense(512, activation='sigmoid'))
CNN2.add(Dropout(0.2))
CNN2.add(Dense(512, activation='sigmoid'))
CNN2.add(Dropout(0.2))
CNN2.add(Dense(10, activation='softmax'))

CNN2.compile(loss='categorical_crossentropy', optimizer=Adam(),
             metrics=['accuracy'])
```

In [26]:

```
CNN2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 30, 30, 64)        1792
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               1180160
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,486,666
Trainable params: 1,486,666
Non-trainable params: 0
_________________________________________________________________
```

In [27]:

```
# Train the the model
start_time = time.time()
with tf.device('/device:GPU:0'):
    train_CNN2 = CNN2.fit(X_train, y_train, batch_size=BATCH_SIZE,
                          epochs=EPOCHS, verbose=1, validation_data=(X_val, y_val))
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0
```

```
Epoch 1/5
313/313 [==============================] - 3s 7ms/step - loss: 1.9403 - accurac
y: 0.2829 - val_loss: 1.5937 - val_accuracy: 0.4041
Epoch 2/5
313/313 [==============================] - 2s 6ms/step - loss: 1.5332 - accurac
y: 0.4352 - val_loss: 1.4321 - val_accuracy: 0.4706
Epoch 3/5
313/313 [==============================] - 3s 8ms/step - loss: 1.3842 - accurac
y: 0.4993 - val_loss: 1.3537 - val_accuracy: 0.5141
```

```
Epoch 4/5
313/313 [==============================] - 2s 6ms/step - loss: 1.2086 - accurac
y: 0.5671 - val_loss: 1.2367 - val_accuracy: 0.5567
Epoch 5/5
313/313 [==============================] - 2s 8ms/step - loss: 1.0903 - accurac
y: 0.6156 - val_loss: 1.1752 - val_accuracy: 0.5815
Total training time : 0.35 minute
```

In [28]:

```
# training accuracy and loss for CNN1
score_train_CNN2 = CNN2.evaluate(X_train, y_train, verbose=0)
print('Train accuracy for CNN2:', score_train_CNN2[1])
print('Train loss for CNN2:', score_train_CNN2[0])
```

```
Train accuracy for CNN2: 0.677299976348877
Train loss for CNN2: 0.9176803827285767
```

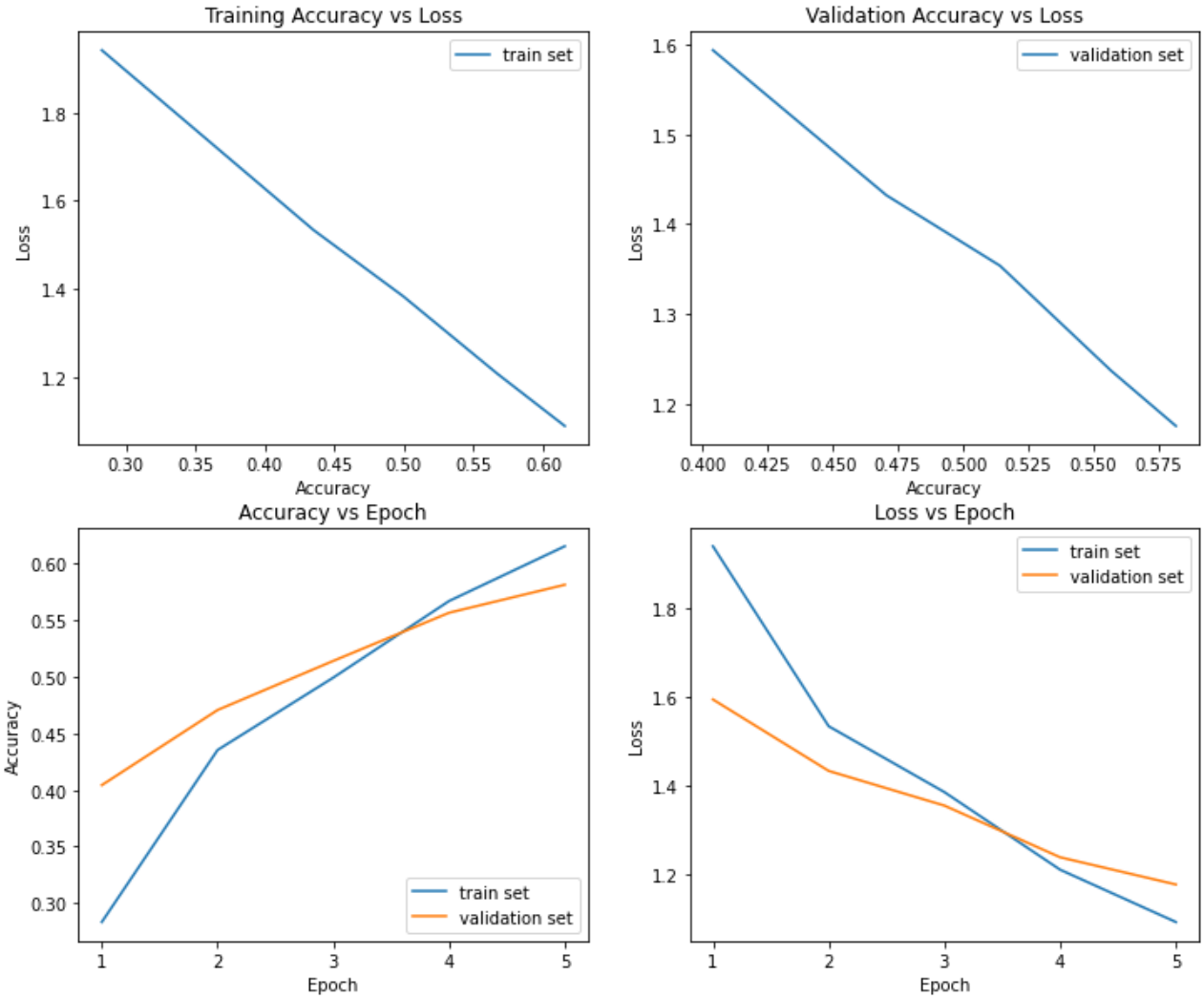
In [29]:

```
# validation accuracy and loss for CNN2
score_val_CNN2 = CNN2.evaluate(X_val, y_val, verbose=0)
print('Validation accuracy for CNN2:', score_val_CNN2[1])
print('Validation loss for CNN2:', score_val_CNN2[0])
```

```
Validation accuracy for CNN2: 0.5814999938011169
Validation loss for CNN2: 1.175216555595398
```

In [30]:

```
accuracy_loss_plot(train_CNN2)
```



Additional MLP models - Changing number of layers and number of neurons per layer

Model 1 - 2 Hidden layers with 1024 units each (in addition to input and output layer)

MLP1 - Model Architecture

- Fully connected layer with 1024 units and a sigmoid activation function
- Fully connected layer with 1024 units and a sigmoid activation function
- Fully connected layer with 1024 units and a sigmoid activation function
- Output layer with 10 units and a softmax activation function

In [31]:
```
# Model 1
MLP1 = Sequential()
MLP1.add(Dense(1024, activation='sigmoid', input_shape=(3072, )))
MLP1.add(Dense(1024, activation='sigmoid'))
MLP1.add(Dense(1024, activation='sigmoid'))
MLP1.add(Dense(10,   activation='softmax'))
```

```
MLP1.compile(loss='categorical_crossentropy',
             optimizer=Adam(), metrics=['accuracy'])
```

In [32]:
```
MLP1.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 1024)              3146752
_________________________________________________________________
dense_7 (Dense)              (None, 1024)              1049600
_________________________________________________________________
dense_8 (Dense)              (None, 1024)              1049600
_________________________________________________________________
dense_9 (Dense)              (None, 10)                10250
=================================================================
Total params: 5,256,202
Trainable params: 5,256,202
Non-trainable params: 0
_________________________________________________________________
```

In [33]:
```
# Train the model
start_time = time.time()
with tf.device('/device:GPU:0'):
    train_MLP1 = MLP1.fit(X_train_MLP, y_train, batch_size=BATCH_SIZE,
                          epochs=EPOCHS, verbose=1, validation_data=(X_val_MLP, y_
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0
```

```
Epoch 1/5
313/313 [==============================] - 2s 6ms/step - loss: 2.2171 - accurac
y: 0.1602 - val_loss: 2.2082 - val_accuracy: 0.1649
Epoch 2/5
313/313 [==============================] - 2s 5ms/step - loss: 2.0508 - accurac
y: 0.2288 - val_loss: 1.9718 - val_accuracy: 0.2785
Epoch 3/5
313/313 [==============================] - 2s 5ms/step - loss: 1.9621 - accurac
y: 0.2701 - val_loss: 1.9387 - val_accuracy: 0.2739
Epoch 4/5
313/313 [==============================] - 2s 5ms/step - loss: 1.9220 - accurac
y: 0.2839 - val_loss: 1.9813 - val_accuracy: 0.2939
Epoch 5/5
313/313 [==============================] - 2s 5ms/step - loss: 1.8825 - accurac
y: 0.3089 - val_loss: 1.8720 - val_accuracy: 0.3036
Total training time : 0.15 minute
```

In [34]:
```
# training accuracy and loss for MLP1
score_train_MLP1 = MLP1.evaluate(X_train_MLP, y_train, verbose=0)
print('Train accuracy for MLP1:', score_train_MLP1[1])
print('Train loss for MLP1:', score_train_MLP1[0])
```

```
Train accuracy for MLP1: 0.31369999051094055
Train loss for MLP1: 1.8453556299209595
```

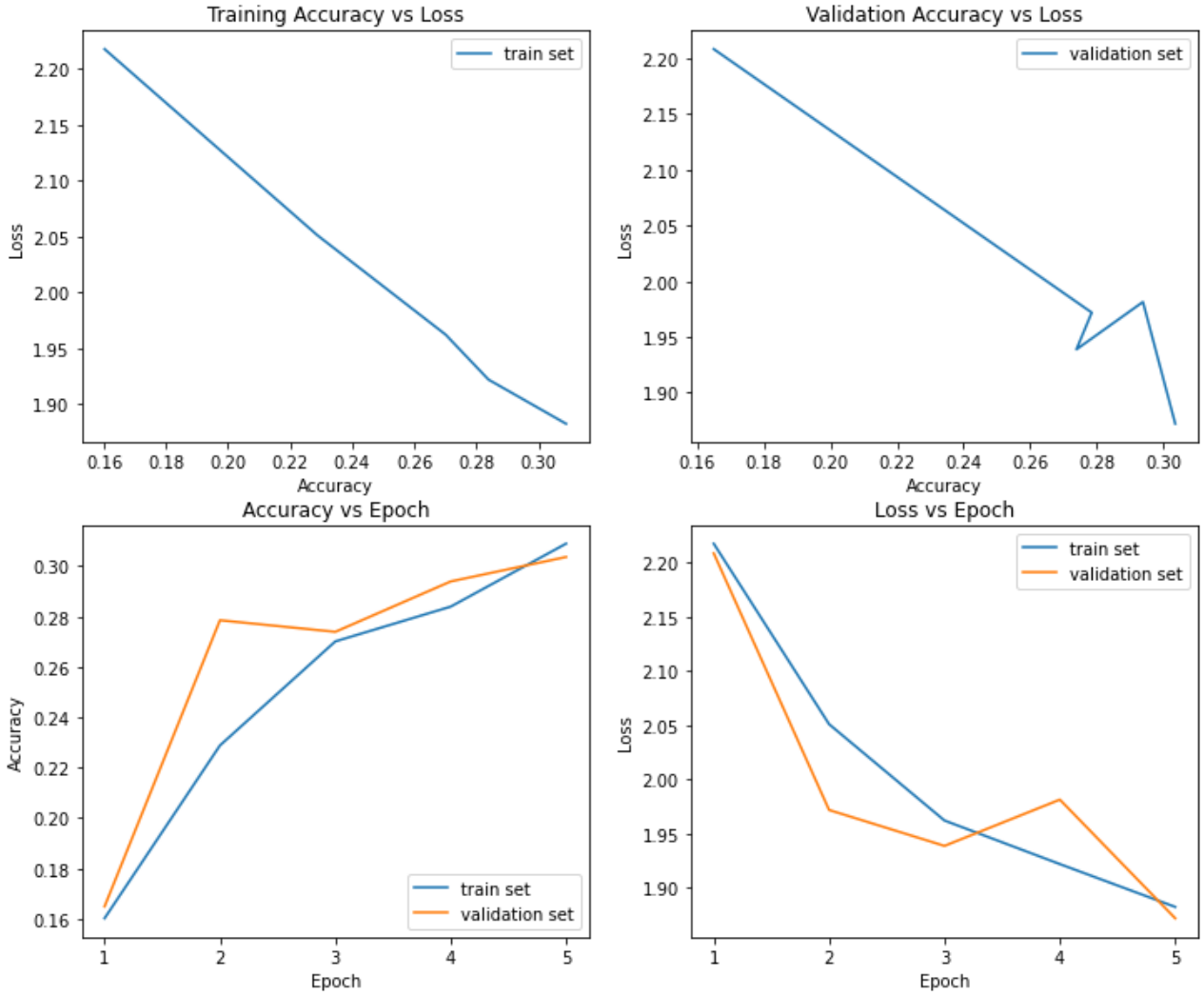
In [35]:
```
# validation accuracy and loss for MLP1
score_val_MLP1 = MLP1.evaluate(X_val_MLP, y_val, verbose=0)
```

```
print('Validation accuracy for MLP1:', score_val_MLP1[1])
print('Validation loss for MLP1:', score_val_MLP1[0])
```

```
Validation accuracy for MLP1: 0.303600013256073
Validation loss for MLP1: 1.871996521949768
```

In [36]:

```
accuracy_loss_plot(train_MLP1)
```



Model 2 - 3 Hidden layers with 256 units each (in addition to input and output layer)

MLP2 - Model Architecture

- Fully connected layer with 256 units and a sigmoid activation function
- Fully connected layer with 256 units and a sigmoid activation function
- Fully connected layer with 256 units and a sigmoid activation function
- Fully connected layer with 256 units and a sigmoid activation function
- Output layer with 10 units and a softmax activation function

In [37]:

```
# Model 2
```

```
MLP2 = Sequential()
MLP2.add(Dense(256, activation='sigmoid', input_shape=(3072, )))
MLP2.add(Dense(256, activation='sigmoid'))
MLP2.add(Dense(256, activation='sigmoid'))
MLP2.add(Dense(256, activation='sigmoid'))
MLP2.add(Dense(10,  activation='softmax'))

MLP2.compile(loss='categorical_crossentropy',
             optimizer=Adam(), metrics=['accuracy'])
```

In [38]:

```
MLP2.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_10 (Dense)             (None, 256)               786688
_________________________________________________________________
dense_11 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_12 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_13 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_14 (Dense)             (None, 10)                2570
=================================================================
Total params: 986,634
Trainable params: 986,634
Non-trainable params: 0
_________________________________________________________________
```

In [39]:

```
# Train the model
start_time = time.time()
with tf.device('/device:GPU:0'):
    train_MLP2 = MLP2.fit(X_train_MLP, y_train, batch_size=BATCH_SIZE,
                          epochs=EPOCHS, verbose=1, validation_data=(X_val_MLP, y_
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0
```

```
Epoch 1/5
313/313 [==============================] - 2s 6ms/step - loss: 2.1929 - accurac
y: 0.1491 - val_loss: 2.1316 - val_accuracy: 0.1562
Epoch 2/5
313/313 [==============================] - 2s 7ms/step - loss: 2.0821 - accurac
y: 0.1806 - val_loss: 2.1248 - val_accuracy: 0.1722
Epoch 3/5
313/313 [==============================] - 2s 5ms/step - loss: 2.0271 - accurac
y: 0.2158 - val_loss: 2.0413 - val_accuracy: 0.2252
Epoch 4/5
313/313 [==============================] - 2s 5ms/step - loss: 1.9826 - accurac
y: 0.2292 - val_loss: 2.0214 - val_accuracy: 0.2297
Epoch 5/5
313/313 [==============================] - 2s 5ms/step - loss: 1.9622 - accurac
y: 0.2464 - val_loss: 1.9492 - val_accuracy: 0.2508
Total training time : 0.18 minute
```

In [40]:

```
# training accuracy and loss for MLP2
score_train_MLP2 = MLP2.evaluate(X_train_MLP, y_train, verbose=0)
```

```
print('Train accuracy for MLP2:', score_train_MLP2[1])
print('Train loss for MLP2:', score_train_MLP2[0])
```

```
Train accuracy for MLP2: 0.24869999289512634
Train loss for MLP2: 1.9285228252410889
```

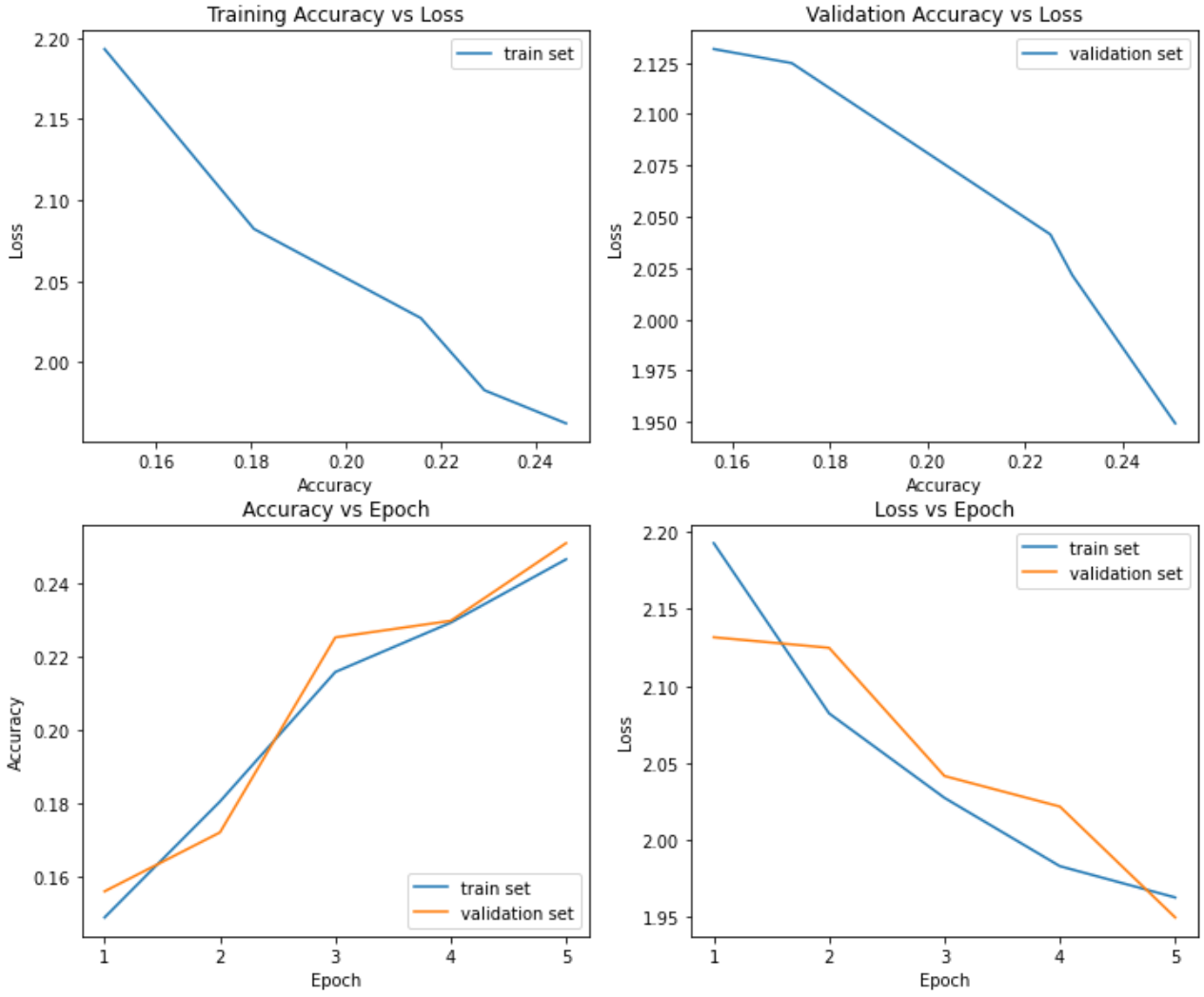
In [41]:

```
# validation accuracy and loss for MLP2
score_val_MLP2 = MLP2.evaluate(X_val_MLP, y_val, verbose=0)
print('Validation accuracy for MLP2:', score_val_MLP2[1])
print('Validation loss for MLP2:', score_val_MLP2[0])
```

```
Validation accuracy for MLP2: 0.2508000135421753
Validation loss for MLP2: 1.9492193460464478
```

In [42]:

```
accuracy_loss_plot(train_MLP2)
```



Model 3 - 5 Hidden layers with 512 units each (in addition to input and output layer)

MLP3 - Model Architecture

- Fully connected layer with 512 units and a sigmoid activation function
- Fully connected layer with 512 units and a sigmoid activation function
- Fully connected layer with 512 units and a sigmoid activation function
- Fully connected layer with 512 units and a sigmoid activation function
- Fully connected layer with 512 units and a sigmoid activation function
- Fully connected layer with 512 units and a sigmoid activation function
- Output layer with 10 units and a softmax activation function

In [43]:
```
# Model 3
MLP3 = Sequential()
MLP3.add(Dense(512, activation='sigmoid', input_shape=(3072, )))
MLP3.add(Dense(512, activation='sigmoid'))
MLP3.add(Dense(512, activation='sigmoid'))
MLP3.add(Dense(512, activation='sigmoid'))
MLP3.add(Dense(512, activation='sigmoid'))
MLP3.add(Dense(512, activation='sigmoid'))
MLP3.add(Dense(10,  activation='softmax'))

MLP3.compile(loss='categorical_crossentropy',
             optimizer=Adam(), metrics=['accuracy'])
```

In [44]:
```
MLP3.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_15 (Dense)             (None, 512)               1573376
_________________________________________________________________
dense_16 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_17 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_18 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_19 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_20 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_21 (Dense)             (None, 10)                5130
=================================================================
Total params: 2,891,786
Trainable params: 2,891,786
Non-trainable params: 0
_________________________________________________________________
```

In [45]:
```
# Train the model
start_time = time.time()
with tf.device('/device:GPU:0'):
    train_MLP3 = MLP3.fit(X_train_MLP, y_train, batch_size=BATCH_SIZE,
                          epochs=EPOCHS, verbose=1, validation_data=(X_val_MLP, y_
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0
```

```
Epoch 1/5
```

```
313/313 [==============================] - 3s 9ms/step - loss: 2.2606 - accurac
y: 0.1345 - val_loss: 2.0821 - val_accuracy: 0.1587
Epoch 2/5
313/313 [==============================] - 2s 8ms/step - loss: 2.1196 - accurac
y: 0.1760 - val_loss: 2.1538 - val_accuracy: 0.1626
Epoch 3/5
313/313 [==============================] - 2s 6ms/step - loss: 2.0955 - accurac
y: 0.1795 - val_loss: 2.0699 - val_accuracy: 0.1965
Epoch 4/5
313/313 [==============================] - 2s 8ms/step - loss: 2.0746 - accurac
y: 0.1815 - val_loss: 2.0703 - val_accuracy: 0.1929
Epoch 5/5
313/313 [==============================] - 2s 8ms/step - loss: 2.0629 - accurac
y: 0.1960 - val_loss: 2.0517 - val_accuracy: 0.2003
Total training time : 0.21 minute
```

In [46]:

```
# training accuracy and loss for MLP3
score_train_MLP3 = MLP3.evaluate(X_train_MLP, y_train, verbose=0)
print('Train accuracy for MLP3:', score_train_MLP3[1])
print('Train loss for MLP3:', score_train_MLP3[0])
```

```
Train accuracy for MLP3: 0.1981000006198883
Train loss for MLP3: 2.04082989692688
```

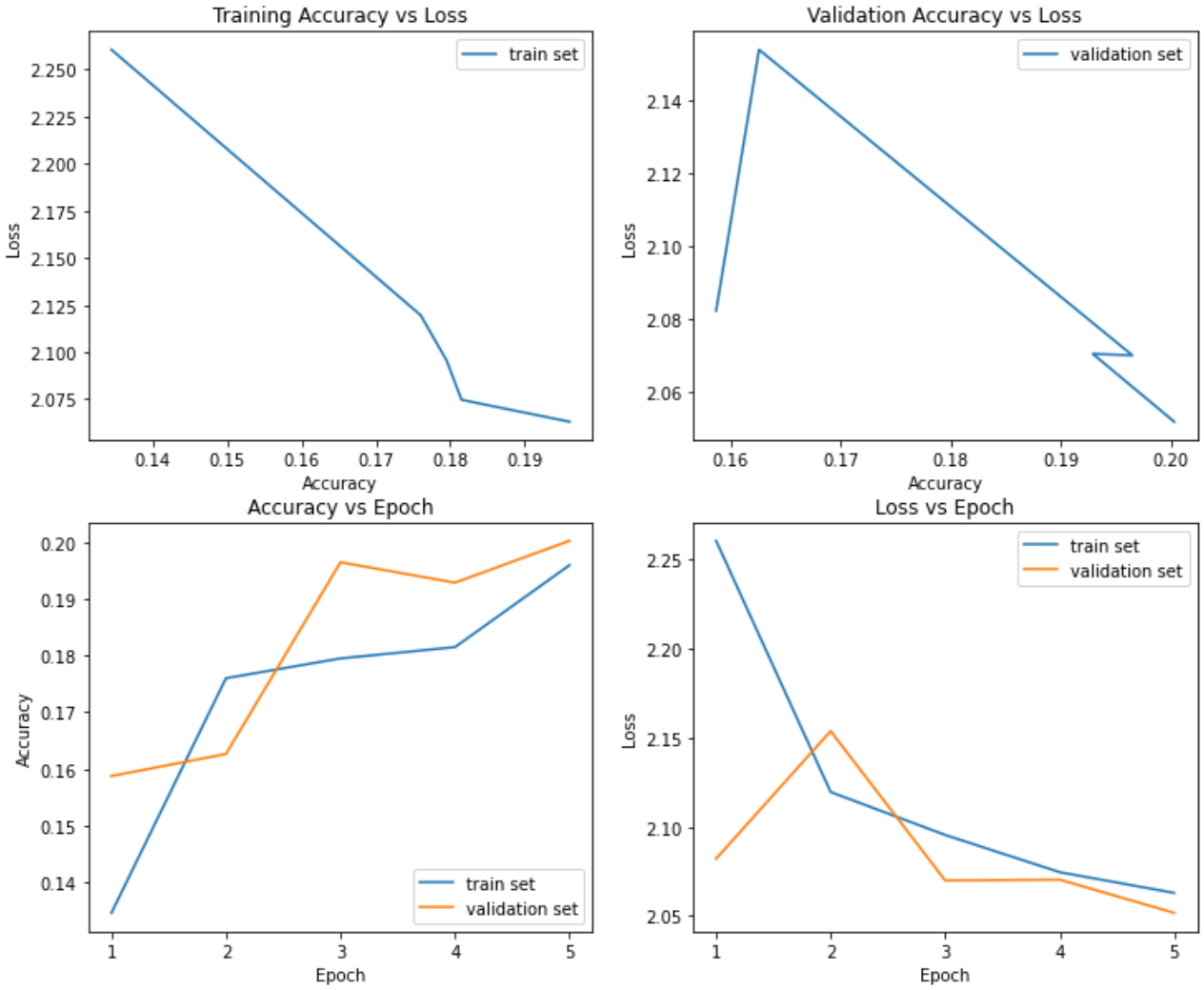
In [47]:

```
# validation accuracy and loss for MLP3
score_val_MLP3 = MLP3.evaluate(X_val_MLP, y_val, verbose=0)
print('Validation accuracy for MLP3:', score_val_MLP3[1])
print('Validation loss for MLP3:', score_val_MLP3[0])
```

```
Validation accuracy for MLP3: 0.20029999315738678
Validation loss for MLP3: 2.0517077445983887
```

In [48]:

```
accuracy_loss_plot(train_MLP3)
```



Model 4 - 2 Hidden layers with 64 units each (in addition to input and output layer)

MLP4 - Model Architecture

- Fully connected layer with 64 units and a sigmoid activation function
- Fully connected layer with 64 units and a sigmoid activation function
- Fully connected layer with 64 units and a sigmoid activation function
- Output layer with 10 units and a softmax activation function

In [49]:

```
# Model 4
MLP4 = Sequential()
MLP4.add(Dense(64, activation='sigmoid', input_shape=(3072, )))
MLP4.add(Dense(64, activation='sigmoid'))
MLP4.add(Dense(64, activation='sigmoid'))
MLP4.add(Dense(10, activation='softmax'))

MLP4.compile(loss='categorical_crossentropy',
             optimizer=Adam(), metrics=['accuracy'])
```

In [50]:
```
MLP4.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_22 (Dense)             (None, 64)                196672
_________________________________________________________________
dense_23 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_24 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_25 (Dense)             (None, 10)                650
=================================================================
Total params: 205,642
Trainable params: 205,642
Non-trainable params: 0
_________________________________________________________________
```

In [51]:
```
# Train the model
start_time = time.time()
with tf.device('/device:GPU:0'):
    train_MLP4 = MLP4.fit(X_train_MLP, y_train, batch_size=BATCH_SIZE,
                          epochs=EPOCHS, verbose=1, validation_data=(X_val_MLP, y_
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0
```

```
Epoch 1/5
313/313 [==============================] - 2s 6ms/step - loss: 2.2149 - accurac
y: 0.1554 - val_loss: 2.0877 - val_accuracy: 0.1967
Epoch 2/5
313/313 [==============================] - 1s 5ms/step - loss: 2.0785 - accurac
y: 0.1862 - val_loss: 2.0574 - val_accuracy: 0.2028
Epoch 3/5
313/313 [==============================] - 2s 5ms/step - loss: 2.0418 - accurac
y: 0.2114 - val_loss: 2.0264 - val_accuracy: 0.2144
Epoch 4/5
313/313 [==============================] - 2s 5ms/step - loss: 2.0116 - accurac
y: 0.2369 - val_loss: 1.9903 - val_accuracy: 0.2566
Epoch 5/5
313/313 [==============================] - 1s 5ms/step - loss: 1.9623 - accurac
y: 0.2694 - val_loss: 1.9355 - val_accuracy: 0.2774
Total training time : 0.14 minute
```

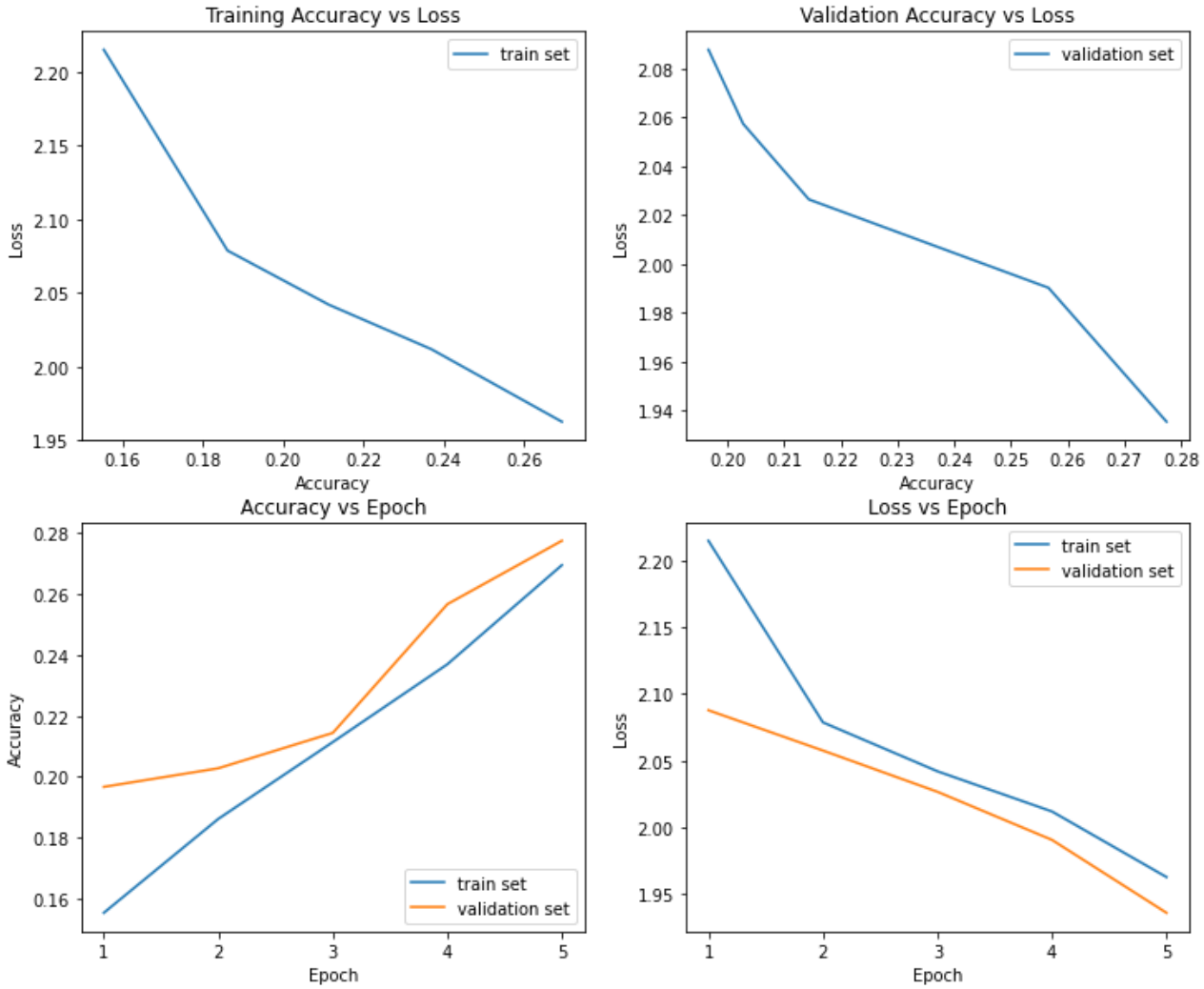
In [52]:
```
# training accuracy and loss for MLP4
score_train_MLP4 = MLP4.evaluate(X_train_MLP, y_train, verbose=0)
print('Train accuracy for MLP4:', score_train_MLP4[1])
print('Train loss for MLP4:', score_train_MLP4[0])
```

```
Train accuracy for MLP4: 0.28139999508857727
Train loss for MLP4: 1.9198554754257202
```

In [53]:
```
# validation accuracy and loss for MLP4
score_val_MLP4 = MLP4.evaluate(X_val_MLP, y_val, verbose=0)
print('Validation accuracy for MLP4:', score_val_MLP4[1])
print('Validation loss for MLP4:', score_val_MLP4[0])
```

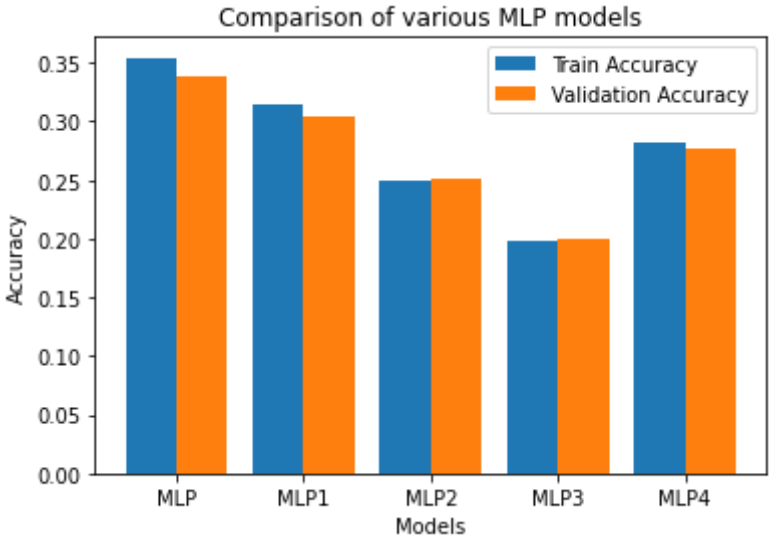
```
Validation accuracy for MLP4: 0.2773999869823456
Validation loss for MLP4: 1.9354931116104126
```

In [54]:
```
accuracy_loss_plot(train_MLP4)
```



Comparison of MLP models -

In [55]:
```
train_accuracies = [score_train_MLP[1], score_train_MLP1[1], score_train_MLP2[1]
val_accuracies = [score_val_MLP[1], score_val_MLP1[1], score_val_MLP2[1], score_

models = ['MLP','MLP1','MLP2','MLP3','MLP4']
X_axis = np.arange(len(models))

plt.bar(X_axis - 0.2, train_accuracies, 0.4, label = 'Train Accuracy')
plt.bar(X_axis + 0.2, val_accuracies, 0.4, label = 'Validation Accuracy')

plt.xticks(X_axis, models)
plt.xlabel("Models")
plt.ylabel("Accuracy")
plt.title("Comparison of various MLP models")
plt.legend()
plt.show()
```

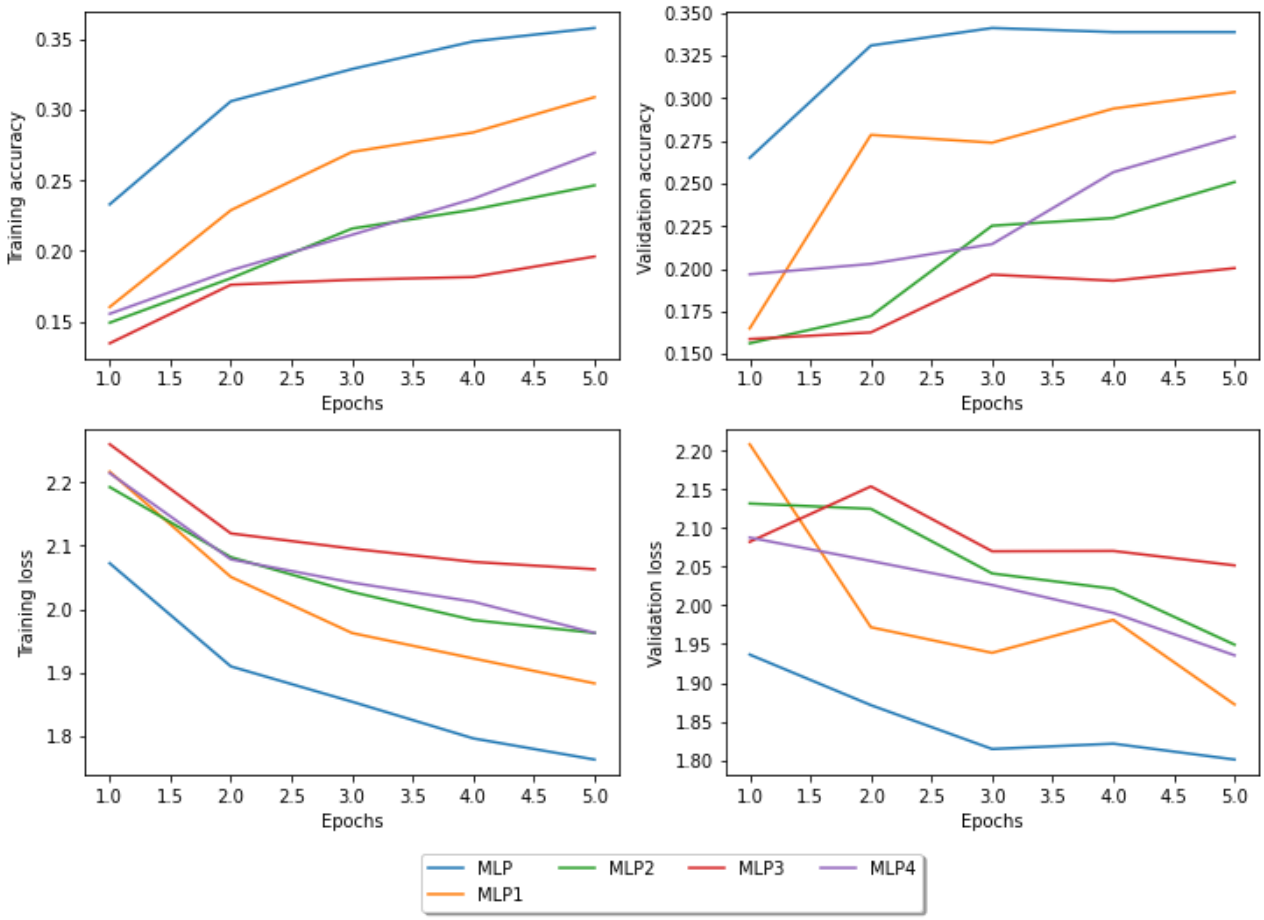


From the above bar graph, it can be observed that highest training and validation accuracy is for the original MLP model followed by model MLP1. MLP3 performs the worst among the 5 models. MLP3 has 5 hidden layers with 512 units each and by adding more layers, more trainable parameters (2,891,786) are added to the model which increases the model complexity. Increasing the number of hidden layers much more than the sufficient number of layers causes accuracy to decrease. MLP2 has 986,634 trainable parameter and hence the complexity is comparatively low. MLP2 and MLP4 give almost the same accuracy. The lower accuracy < 40 percent for each of these models could be attributed to the fact that MLPs do not do very well on image datasets due to loss of spatial information and not being translation invariant. Also, we are only training the dataset on the 20 percent of the total training set. Using the entire set could improve the accuracy (reaches upto 50 percent) for MLP.

In [56]:
```
history_list1 = [train_MLP, train_MLP1, train_MLP2, train_MLP3, train_MLP4]
labels1 = ['MLP','MLP1','MLP2','MLP3','MLP4']
compare_models(history_list1, labels1)
```

Out[56]:

	Train Accuracy	Train Loss	Val Accuracy	Val Loss
MLP	35.79	1.7621	33.88	1.8008
MLP1	30.89	1.8825	30.36	1.8720
MLP2	24.64	1.9622	25.08	1.9492
MLP3	19.60	2.0629	20.03	2.0517
MLP4	26.94	1.9623	27.74	1.9355

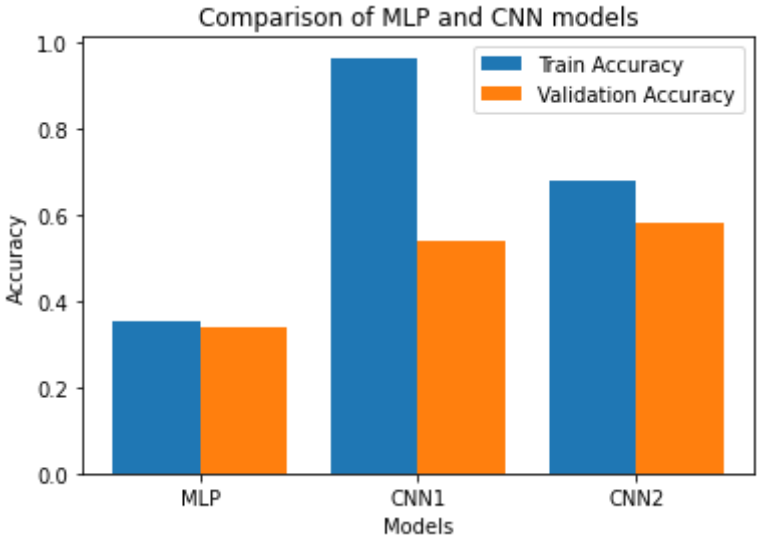


From the above figure, we can observe MLP performs better as comapared to other MLP models. The training accuracy stops increasing after 4 epochs (horizontal line) but the validation accuracy keeps on increasing for original MLP. For MLP3, no or little learning happens after epoch 2 for the training set and after epoch 3 for validation set and performs the worst among these models. MLP2 and MLP4 perform similar on both the training and validation sets. MLP1 is the second best model after original MLP.

4. PERFORMANCE MLP vs CNNs

In [57]:

```
# plotting bar graph for comparison
train_accuracies2 = [score_train_MLP[1], score_train_CNN1[1], score_train_CNN2[1
val_accuracies2 = [score_val_MLP[1], score_val_CNN1[1], score_val_CNN2[1]]

models = ['MLP','CNN1','CNN2']
X_axis = np.arange(len(models))

plt.bar(X_axis - 0.2, train_accuracies2, 0.4, label = 'Train Accuracy')
plt.bar(X_axis + 0.2, val_accuracies2, 0.4, label = 'Validation Accuracy')

plt.xticks(X_axis, models)
plt.xlabel("Models")
plt.ylabel("Accuracy")
plt.title("Comparison of MLP and CNN models")
plt.legend()
plt.show()
```



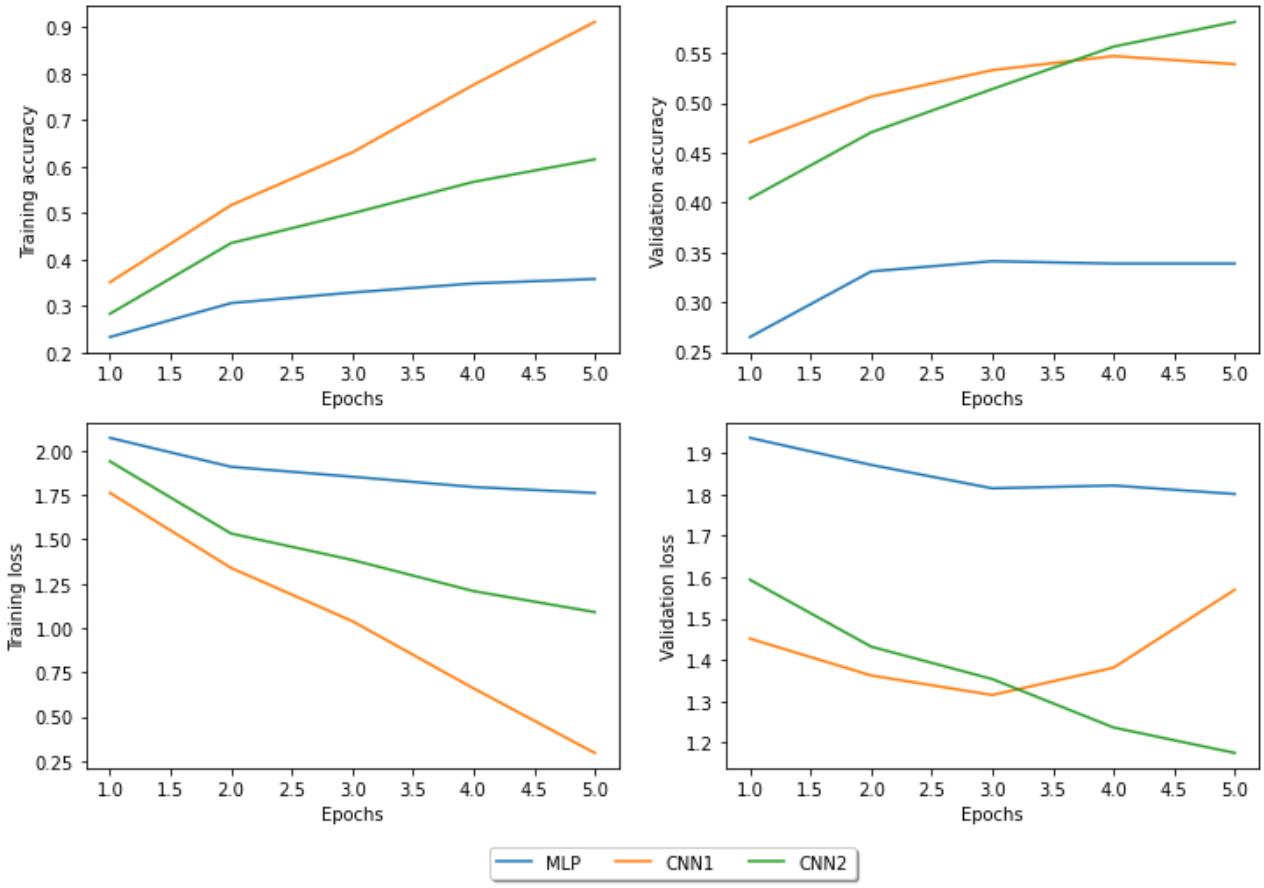
From the bar graph above, it can be observed that MLP gives around 34 percent accuracy on the training set and 34 percent on validation set. MLPs do not perform very well on the image datasets as the spatial information is lost when the image is flattened (matrix to vector) before feeding to the MLP. MLPs react differently to an input (images) and its shifted version — they are not translation invariant. For example, if a picture of a cat appears in the top left of the image in one picture and the bottom right of another picture, the MLP will try to correct itself and assume that a cat will always appear in this section of the image. MLPs use one perceptron for each input and the amount number of weights rapidly becomes unmanageable for large images. It includes too many parameters because it is fully connected. Each node is connected to every other node in next and the previous layer, forming a very dense web — resulting in redundancy and inefficiency.

CNNs, on the other hand perform way better than MLP on Image datasets. The CNN architecture implicitly combines the benefits obtained by a standard neural network training with the convolution operation to efficiently classify images. By performing the convolution operations, the dimensionality of the data shrinks significantly. Hence, the number of parameters to be learned decreases. Hence, the network complexity decreases. CNNs work well with data that has a spatial relationship. The benefit of using CNNs is their ability to develop an internal representation of a two-dimensional image. This allows the model to learn position and scale in variant structures in the data, which is important when working with images. CNN allows parameter sharing, weight sharing so that the filter looks for a specific pattern and is location invariant — can find the pattern anywhere in an image. CNN1 gives much higher training accuracy (> 90) but a moderate validation accuracy (54), the model is overfitting. The model complexity is too high for the given amount of training data and the model starts to memorize rather than approximating the relationship between input and output. CNN2 gives moderate accuracy on both training (65) and validation (55), which means it generalizes well. The performance of CNNs is better than MLP but they still suffer from problems, such as overfitting, among others.

In [58]:
```
history_list2 = [train_MLP, train_CNN1, train_CNN2]
labels2 = ['MLP', 'CNN1', 'CNN2']
compare_models(history_list2, labels2)
```

Out[58]:

	Train Accuracy	Train Loss	Val Accuracy	Val Loss
MLP	35.79	1.7621	33.88	1.8008
CNN1	91.15	0.2962	53.91	1.5696
CNN2	61.56	1.0903	58.15	1.1752

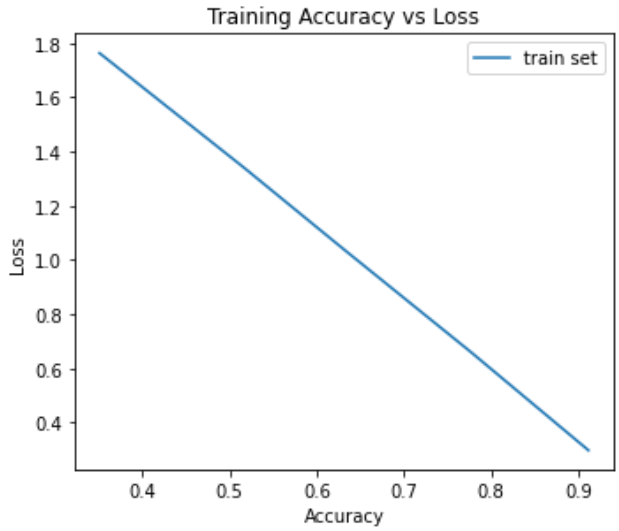


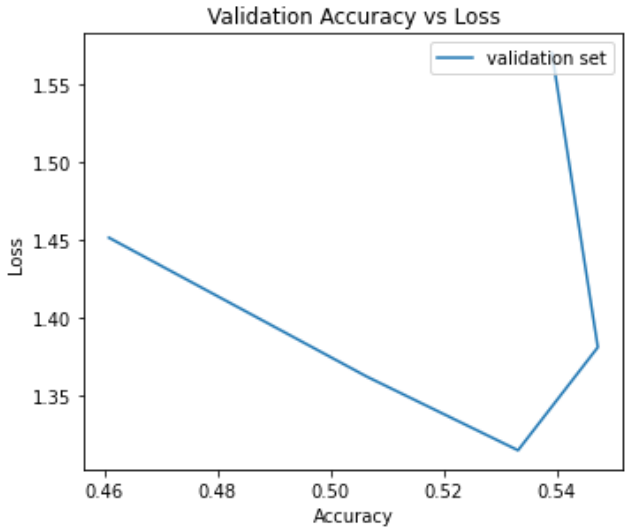
The figure above clearly shows that CNN1 is overfitting. The slope for CNN1 is very steep for training accuracy whereas it is flat for validation data after epoch 3. Validation loss starts to increase after epoch 3 for CNN1. CNN2 gives moderate accuracy on both training (65) and validation (55), which means it generalizes well. If trained for more epochs, CNN2 could higher accuracy for both training and validation sets. MLP performs the worst among the three model. There is no learning happening after epoch 2 for both the sets.

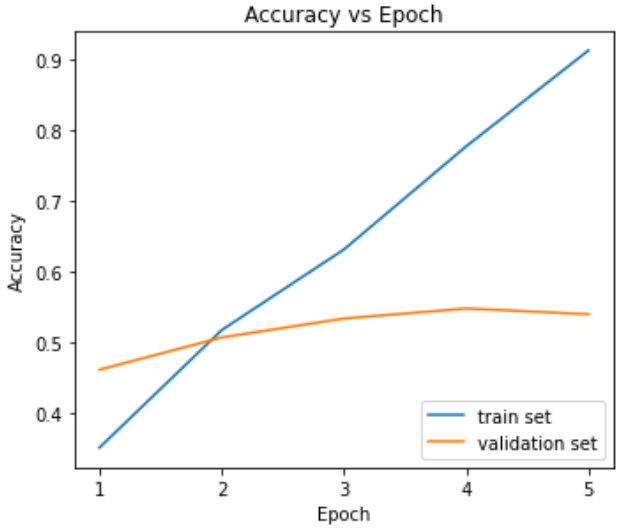
5. COMPARISON OF CNN1 AND CNN2

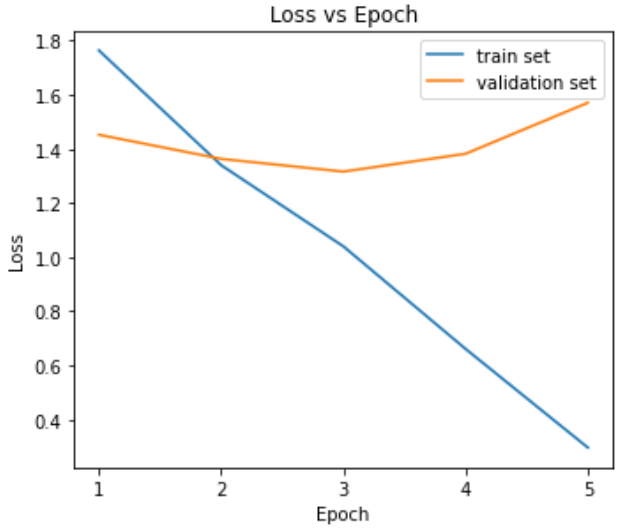
In [60]:

```
# training and validation curves for CNN1
accuracy_loss_plot(train_CNN1)
```



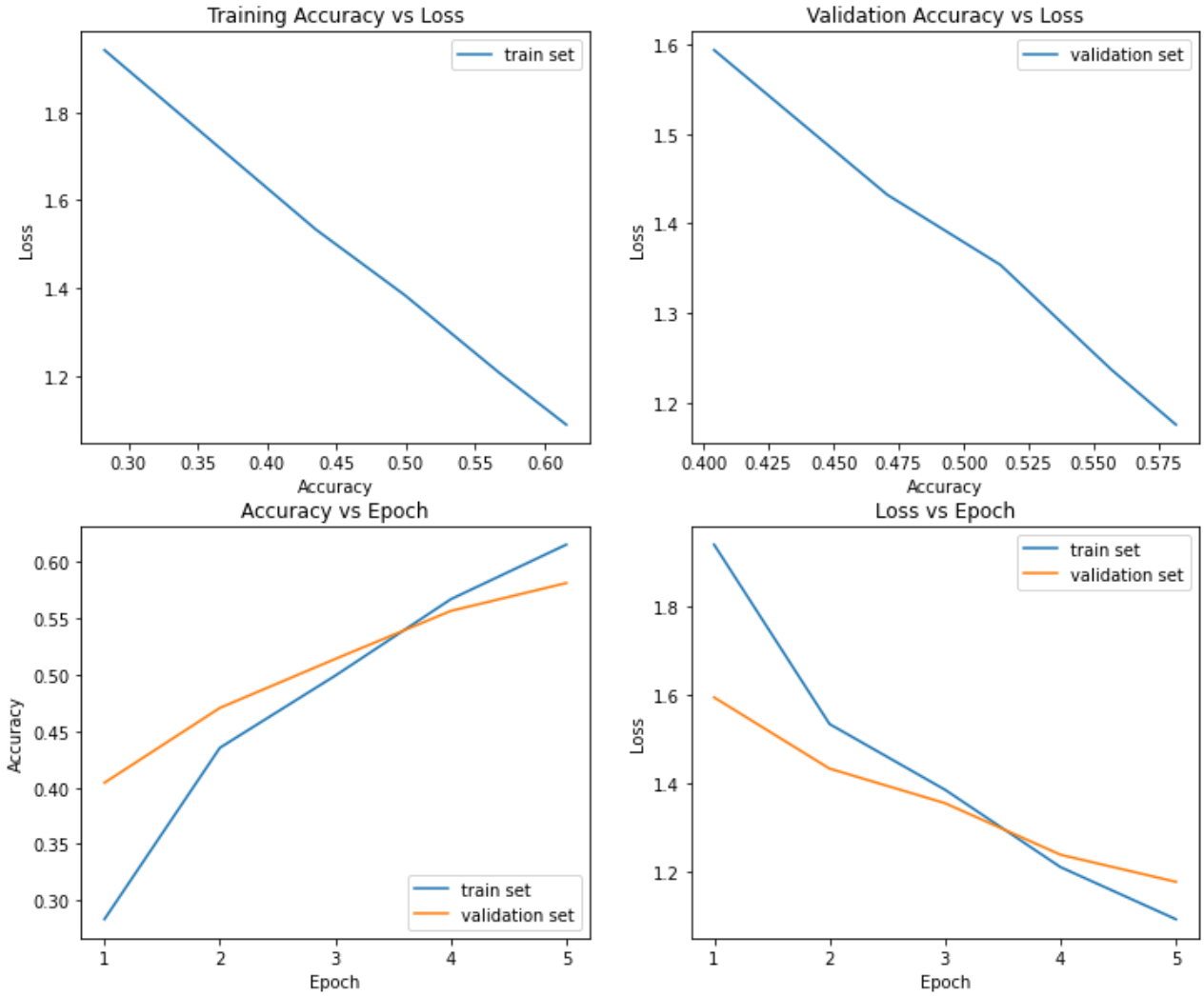






In [61]:

```
# training and validation curves for CNN2
accuracy_loss_plot(train_CNN2)
```



CNN1 gives 92 percent for the training set but 53 percent for the validation set. This means that CNN1 is overfitting and cannot generalize beyond the training set. Overfitting can be simply thought of as fitting patterns that do not exist due to the high number of features or the low number of training examples. CNN1 is memorizing the relationship between input and output for training data rather than approximating and hence does not generalize well. This could be due to the fact that we have 2 fully connected layers after 2 Convolutional layers, hence the model is too powerful for the given amount of training data. The model complexity is too high, as we 25,997,130 trainable parameters. Also, as we are using a subset of whole training set, the model is too complex for the selected training set. Overfitting happens when the dataset is not large enough to accommodate the number of features. CNN2 which is a less complex model as compared to CNN1, with only 1,486,666 trainable parameters, gives 65 percent on the training set and 57 percent on the validation set. This model is able to generalize well as compared to CNN1. We added regularization in the form of dropout (randomly deactivate neurons), hence reducing the overfitting. We also added max pooling in CNN2. Pooling in some sense tries to do feature selection by reducing the dimension of the input. So, by selecting a subset of features, we are less likely to find false patterns.

From the training and validation curves, we observe that CNN1 starts overfitting after the validation accuracy reaches 55 percent as the loss starts increasing very steeply. The loss keeps on decreasing for the training set. The epoch vs accuracy curve shows that for training

set, the accuracy rises steeply from epoch 1 to epoch 2, steadies a bit from epoch 2 to epoch 3 and then continues to rise till epoch 5. However, the slope is not so steep for validation set as the validation accuracy stops increasing after epoch 3. On the other hand, for CNN2 both the training and validation accuracy increase from epoch 1 to epoch 5. The epoch vs accuracy curve shows that for training set, the accuracy rises steeply from epoch 1 to epoch 2, steadies a bit from epoch 2 to epoch 3 and then continues to rise till epoch 5. For the validation set, rises from epoch 1 to epoch 2, does not change much from epoch 2 to epoch 3 and then continues to rise form epoch 3 till epoch 5. The rate of increase of accuracy is almost same for both training and validation from epoch 3 to epoch 5. The loss decreases in a similar fashion as the accuracy increases.

Training Time -

CNN1 – 0.45 minute

CNN2 – 0.30 minute

CNN1 takes approximately 1.5 times the time required to train CNN2. As explained above, CNN2 model has regularization in the form of dropout (randomly deactivate neurons) and pooling layer (a subset of features) which significantly reduces the complexity and reduces the number of trainable parameters from 25,997,130 to 1,486,666. Hence, CNN2 takes significantly less time to train as compared to CNN1.

If the networks were trained for more epochs, CNN2 is still expected to perform as compared to CNN1. CNN1 seems to be too powerful for this amount of training data and has very high complexity. From the epoch vs accuracy graph for CNN1, we can expect the training accuracy to further increase to 98-99 percent, but the validation accuracy would remain almost the same (the slope of curve 0 - horizontal). The model would still overfit. However, looking at the slope of CNN2, the model is expected give higher accuracy for both training and validation sets as the number of epochs are increased. Training the model for 30-40 could give 85-90 percent on the training set and 75- 80 on the validation. Further increasing the number of epochs for CNN2, the model start to overfit. Once important factor contributing to overfitting of CNN1 is small amount of training data as compared to size of validation data. The training data is expected to around 7-8 times the validation data. Hence, more data needs to be used prevent overfitting in CNN1 and CNN2 as well after some point.

6. RECOMMENDATIONS TO IMPROVE THE NETWORK

• **More Training data** - The model can only store so much information. This means that the more training data we feed it, the less likely it is to overfit. The reason is that, as we add more data, the model becomes unable to overfit all the samples, and is forced to generalize to make progress. Training the model on the whole training set instead of 20% randomly sampled data would reduce the chance of overfitting and give better accuracy on both the MLP and CNN models.

• **Reduce architecture complexity** - Another way to reduce overfitting is to lower the capacity of the model to memorize the training data. As such, the model will need to focus on the relevant patterns in the training data, which results in better generalization. This could be done by reducing the number of trainable parameters such that the model simple enough that it does not overfit, but complex enough to learn from the data. This also has the advantage of making the model lighter, train faster and run faster.

• **Adding regularization (mostly dropout or L1/L2)** - One of the most powerful and well-known technique of regularization is to add a penalty to the loss function. The most common are called L1 and L2. With the penalty, the model is forced to make compromises on its weights, as it can no longer make them arbitrarily large. This makes the model more general, which helps combat overfitting. **Dropout** - The idea is to randomly deactivate either neurons (dropout) during the training. This forces the network to become redundant, as it can no longer rely on specific neurons or connections to extract specific features. Once the training is done, all neurons are restored. As observed in CNN2, adding dropout layers after fully connected layers causes 20 percent of the neurons to randomly deactivate reduces the overfitting on the training set and the model is able to generalize well. Adding max pooling layers after convolutional layers reduces the dimension of the input by feature selection and the output has smaller dimension.

• **Early Termination** - The model starts by learning a correct distribution of the data, and, at some point, starts to overfit the data. By identifying the moment where this shift occurs, we can stop the learning process before the overfitting happens. As before, this is done by looking at the training loss and validation loss over time. In CNN1, this shift occurs when the validation accuracy reaches 53 percent, as the validation loss starts increasing after that.

• **Considering alternate activation functions** (ReLU for fully connected layers) - ReLU is more computationally efficient to compute than Sigmoid like functions since ReLU just needs to pick max(0,x) and not perform expensive exponential operations as in Sigmoid. ReLU does not have the vanishing gradient problem. Vanishing gradients lead to very small changes in the weights proportional to the partial derivative of the error function. Although ReLU does have the disadvantage of dying cells (if too many activations get below zero then most of the units(neurons) in network with Relu will simply output zero, in other words, die) which limits the capacity of the network. To overcome this, we can use a variant of ReLU such as leaky ReLU, ELU. We could consider using ReLU in the network for full connected layers as well instead of sigmoid.

• **Batch Normalization and using larger batch size** - Batch normalization makes the input to each layer have zero mean and unit variance. Batch normalization yields faster training, higher accuracy and enable higher learning rates. This suggests that it is the higher learning rate that BN enables, which mediates the majority of its benefits; it improves regularization, accuracy and gives faster convergence. It has a regularizing effect which means you can often remove dropout.

• **Data augmentation & Noise** – We can try to make our data appear as if it was more diverse. To do that, we can use data augmentation techniques so that each time a sample is processed by the model, it is slightly different from the previous time. This will make it harder for the model

to memorize parameters for each sample. We can also add noise to the input and output making the model robust to natural perturbations and this will make the training more diversified. However, the magnitude of the noise should not too be great.

In [1]:
```
import tensorflow as tf
import pandas as pd
import numpy as np
from sklearn.metrics import mean_squared_error
```

In [3]:
```
import matplotlib.pyplot as plt
def loss_plot(history):
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model loss')
    plt.ylabel('loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Val'], loc='upper right')
    #plt.legend(['Train'], loc='upper right')
```

In [25]:
```
data = pd.read_csv('data/q2_dataset.csv')
```

In [9]:
```
data.head()
```

Out[9]:

	Date	Close/Last	Volume	Open	High	Low
0	07/08/20	$381.37	29272970	376.72	381.50	376.36
1	07/07/20	$372.69	28106110	375.41	378.62	372.23
2	07/06/20	$373.85	29663910	370.00	375.78	369.87
3	07/02/20	$364.11	28510370	367.85	370.47	363.64
4	07/01/20	$364.11	27684310	365.12	367.36	363.91

In [8]:
```
data.tail()
```

Out[8]:

	Date	Close/Last	Volume	Open	High	Low
1254	07/15/2015	$126.82	33559770	125.72	127.15	125.58
1255	07/14/2015	$125.61	31695870	126.04	126.37	125.04
1256	07/13/2015	$125.66	41365600	125.03	125.76	124.32
1257	07/10/15	$123.28	61292800	121.94	123.85	121.21
1258	07/09/15	$120.07	78291510	123.85	124.06	119.22

In [ ]:
```
type(data)
```

Out[ ]:
```
pandas.core.frame.DataFrame
```

In [26]:
```
list(data.columns)
```

Out[26]:
```
['Date', ' Close/Last', ' Volume', ' Open', ' High', ' Low']
```

In [27]:
```
data.columns = data.columns.str.strip()
list(data.columns)
```

Out[27]:
```
['Date', 'Close/Last', 'Volume', 'Open', 'High', 'Low']
```

In [28]:
```
data['target']= data['Open']
data['Date'] =pd.to_datetime(data.Date)
data=data.sort_values(by='Date')
data.reset_index(inplace=True, drop=True)
data.head()
```

Out[28]:

	Date	Close/Last	Volume	Open	High	Low	target
0	2015-07-09	$120.07	78291510	123.85	124.06	119.22	123.85
1	2015-07-10	$123.28	61292800	121.94	123.85	121.21	121.94
2	2015-07-13	$125.66	41365600	125.03	125.76	124.32	125.03
3	2015-07-14	$125.61	31695870	126.04	126.37	125.04	126.04
4	2015-07-15	$126.82	33559770	125.72	127.15	125.58	125.72

In [19]:
```
data.head(20)
```

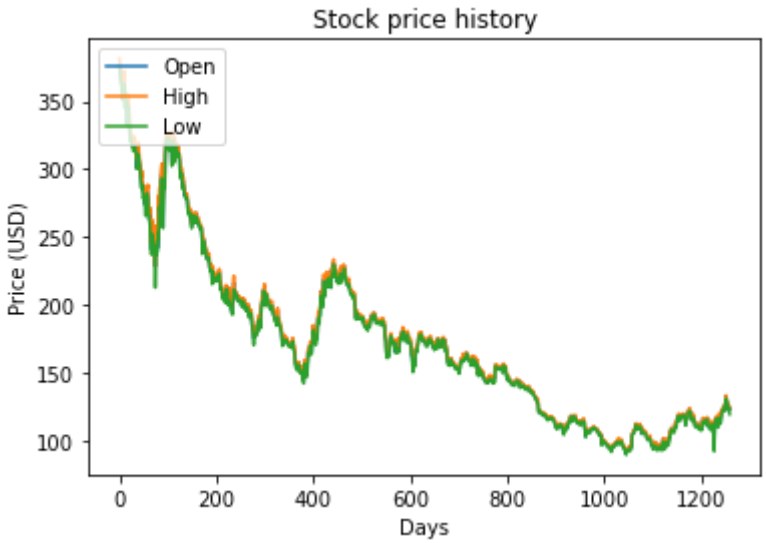
Out[19]:

	Date	Close/Last	Volume	Open	High	Low	target
0	2015-07-09	$120.07	78291510	123.85	124.06	119.22	123.85
1	2015-07-10	$123.28	61292800	121.94	123.85	121.21	121.94
2	2015-07-13	$125.66	41365600	125.03	125.76	124.32	125.03
3	2015-07-14	$125.61	31695870	126.04	126.37	125.04	126.04
4	2015-07-15	$126.82	33559770	125.72	127.15	125.58	125.72
5	2015-07-16	$128.51	35987630	127.74	128.57	127.35	127.74
6	2015-07-17	$129.62	45970470	129.08	129.62	128.31	129.08
7	2015-07-20	$132.07	55204920	130.97	132.97	130.70	130.97
8	2015-07-21	$130.75	73006780	132.85	132.92	130.32	132.85
9	2015-07-22	$125.22	115288400	121.99	125.50	121.99	121.99
10	2015-07-23	$125.16	50832950	126.20	127.09	125.06	126.20
11	2015-07-24	$124.50	42090320	125.32	125.74	123.90	125.32
12	2015-07-27	$122.77	44371580	123.09	123.61	122.12	123.09
13	2015-07-28	$123.38	33570380	123.38	123.91	122.55	123.38
14	2015-07-29	$122.99	36912040	123.15	123.50	122.27	123.15
15	2015-07-30	$122.37	33400950	122.32	122.57	121.71	122.32
16	2015-07-31	$121.30	42832890	122.60	122.64	120.91	122.60
17	2015-08-03	$118.44	69639900	121.50	122.57	117.52	121.50

	Date	Close/Last	Volume	Open	High	Low	target
18	2015-08-04	$114.64	123601900	117.42	117.70	113.25	117.42
19	2015-08-05	$115.40	99202400	112.95	117.44	112.10	112.95

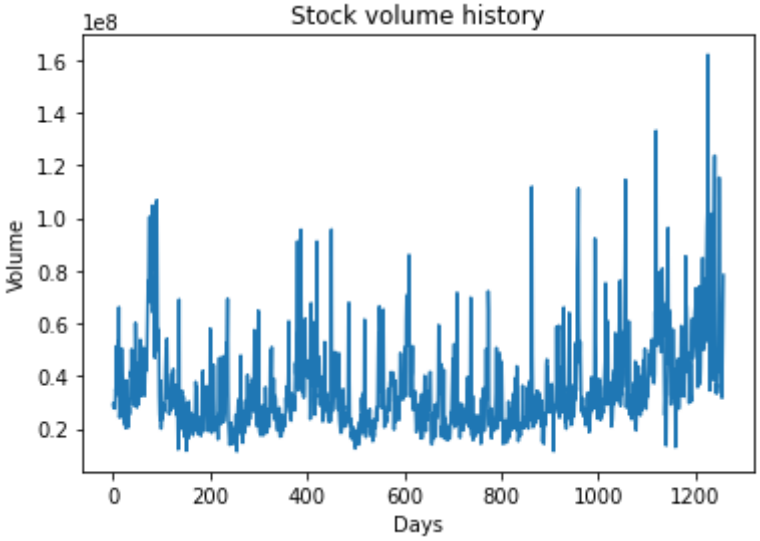
In [ ]:

```
from matplotlib import pyplot as plt
plt.figure()
plt.plot(data["Open"])
plt.plot(data["High"])
plt.plot(data["Low"])
#plt.plot(data["Close"])
plt.title('Stock price history')
plt.ylabel('Price (USD)')
plt.xlabel('Days')
plt.legend(['Open','High','Low'], loc='upper left')
plt.show()
```



In [ ]:

```
plt.figure()
plt.plot(data["Volume"])
plt.title('Stock volume history')
plt.ylabel('Volume')
plt.xlabel('Days')
plt.show()
```



In [29]:

```
#create features using columns from previous 3 days
data['Volume_t-3'] = data.shift(3)['Volume']
data['Volume_t-2'] = data.shift(2)['Volume']
data['Volume_t-1'] = data.shift(1)['Volume']
data['Open_t-3'] = data.shift(3)['Open']
data['Open_t-2'] = data.shift(2)['Open']
data['Open_t-1'] = data.shift(1)['Open']
data['High_t-3'] = data.shift(3)['High']
data['High_t-2'] = data.shift(2)['High']
data['High_t-1'] = data.shift(1)['High']
data['Low_t-3'] = data.shift(3)['Low']
data['Low_t-2'] = data.shift(2)['Low']
data['Low_t-1'] = data.shift(1)['Low']
data['target']= data['Open']
data.head()
```

Out[29]:

	Date	Close/Last	Volume	Open	High	Low	target	Volume_t-3	Volume_t-2	Volume_
0	2015-07-09	$120.07	78291510	123.85	124.06	119.22	123.85	NaN	NaN	N
1	2015-07-10	$123.28	61292800	121.94	123.85	121.21	121.94	NaN	NaN	7829151
2	2015-07-13	$125.66	41365600	125.03	125.76	124.32	125.03	NaN	78291510.0	6129280
3	2015-07-14	$125.61	31695870	126.04	126.37	125.04	126.04	78291510.0	61292800.0	4136560
4	2015-07-15	$126.82	33559770	125.72	127.15	125.58	125.72	61292800.0	41365600.0	3169587

In [30]:

```
data = data.drop(['Close/Last','Volume','Open','High','Low'], axis = 1)
data.head()
```

Out[30]:

	Date	target	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	Higl

	Date	target	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	Higl
0	2015-07-09	123.85	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
1	2015-07-10	121.94	NaN	NaN	78291510.0	NaN	NaN	123.85	NaN	
2	2015-07-13	125.03	NaN	78291510.0	61292800.0	NaN	123.85	121.94	NaN	12
3	2015-07-14	126.04	78291510.0	61292800.0	41365600.0	123.85	121.94	125.03	124.06	12
4	2015-07-15	125.72	61292800.0	41365600.0	31695870.0	121.94	125.03	126.04	123.85	12

In [31]:
```
data.isna().sum()
```

Out[31]:
```
Date          0
target        0
Volume_t-3    3
Volume_t-2    2
Volume_t-1    1
Open_t-3      3
Open_t-2      2
Open_t-1      1
High_t-3      3
High_t-2      2
High_t-1      1
Low_t-3       3
Low_t-2       2
Low_t-1       1
dtype: int64
```

In [32]:
```
#drop columns with null values
data = data.dropna()
data.reset_index(inplace=True, drop=True)
data.head()
```

Out[32]:

	Date	target	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	Hig
0	2015-07-14	126.04	78291510.0	61292800.0	41365600.0	123.85	121.94	125.03	124.06	12
1	2015-07-15	125.72	61292800.0	41365600.0	31695870.0	121.94	125.03	126.04	123.85	12
2	2015-07-16	127.74	41365600.0	31695870.0	33559770.0	125.03	126.04	125.72	125.76	12
3	2015-07-17	129.08	31695870.0	33559770.0	35987630.0	126.04	125.72	127.74	126.37	12
4	2015-07-20	130.97	33559770.0	35987630.0	45970470.0	125.72	127.74	129.08	127.15	12

In [33]:
```
list(data.columns)
```

Out[33]:
```
['Date',
 'target',
 'Volume_t-3',
 'Volume_t-2',
 'Volume_t-1',
 'Open_t-3',
 'Open_t-2',
 'Open_t-1',
 'High_t-3',
 'High_t-2',
 'High_t-1',
 'Low_t-3',
 'Low_t-2',
 'Low_t-1']
```

In [34]:
```
data = data[[
 'Date',
 'Volume_t-3',
 'Volume_t-2',
 'Volume_t-1',
 'Open_t-3',
 'Open_t-2',
 'Open_t-1',
 'High_t-3',
 'High_t-2',
 'High_t-1',
 'Low_t-3',
 'Low_t-2',
 'Low_t-1',
 'target']]
data.head()
```

Out[34]:

	Date	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	High_t-2	Hi
0	2015-07-14	78291510.0	61292800.0	41365600.0	123.85	121.94	125.03	124.06	123.85	1
1	2015-07-15	61292800.0	41365600.0	31695870.0	121.94	125.03	126.04	123.85	125.76	1
2	2015-07-16	41365600.0	31695870.0	33559770.0	125.03	126.04	125.72	125.76	126.37	
3	2015-07-17	31695870.0	33559770.0	35987630.0	126.04	125.72	127.74	126.37	127.15	1
4	2015-07-20	33559770.0	35987630.0	45970470.0	125.72	127.74	129.08	127.15	128.57	1

Dataset Creation

We sorted the dataset in ascending order, since our intention is to predict the opening price from the **previous** three days. Using the pandas shift function which shifts the index by desired number of periods, we were able to create new features by specifying the index that was

needed. For example, to get the Volume from three days prior, we shift by 3 - data.shift(3)['Volume']. This process was repeated for all necessary columns and indices.

In [35]:
```
len(data)
```

Out[35]:
```
1256
```

In [36]:
```
from sklearn.model_selection import train_test_split
#split the data into train and test set
train, test = train_test_split(data, test_size=0.30, random_state=0)
#save the data
train.to_csv('train_data_RNN.csv',index=False)
test.to_csv('test_data_RNN.csv',index=False)
```

In [ ]:
```
type(train)
```

Out[ ]:
```
pandas.core.frame.DataFrame
```

In [38]:
```
train.head()
```

Out[38]:

	Date	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	High_t-2
689	2018-04-09	34581850.0	26750260.0	34949690.0	164.88	172.58	170.97	172.01	174.23
1134	2020-01-14	42621540.0	35217270.0	30521720.0	307.24	310.60	311.64	310.43	312.67
901	2019-02-11	28204640.0	31644240.0	23793830.0	174.65	172.40	168.99	175.57	173.94
579	2017-10-27	17633730.0	21175670.0	16916650.0	156.29	156.91	157.23	157.42	157.55
367	2016-12-23	21337310.0	23724430.0	26043820.0	116.74	116.80	116.35	117.50	117.40

In [40]:
```
data_train = pd.read_csv('train_data_RNN.csv')
data_test = pd.read_csv('test_data_RNN.csv')
```

Preprocessing

Scaling the data

The range of the data is widely varied. The values of Volume are very high and could skew the model. Normalizing data helps the algorithm in converging i.e. to find local/ global minimum efficiently. We utilise the Minmax scaler to keep feature values between 0 and 1.

Scaled values of X are created using the following formula:

X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

X_scaled = X_std * (max - min) + min

We also tried the Standard scaler, however there was no significant difference in training or test loss with this scaler.

Splitting Features and Target

The target is the opening price of the day we wish to predict.

In [41]:
```
#separate features and target
X_train = data_train.drop(['Date','target'], axis = 1)
y_train = data_train['target']
X_test_date = data_test
X_test = data_test.drop(['Date','target'], axis = 1)
y_test = data_test['target']
```

In [ ]:
```
X_train
```

Out[ ]:

	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	High_t-2	High_t
0	34581850.0	26750260.0	34949690.0	164.88	172.58	170.97	172.01	174.23	172.4
1	42621540.0	35217270.0	30521720.0	307.24	310.60	311.64	310.43	312.67	317.0
2	28204640.0	31644240.0	23793830.0	174.65	172.40	168.99	175.57	173.94	170.6
3	17633730.0	21175670.0	16916650.0	156.29	156.91	157.23	157.42	157.55	157.8
4	21337310.0	23724430.0	26043820.0	116.74	116.80	116.35	117.50	117.40	116.5
...	...	...	...	...	...	...	...	...	.
874	20182050.0	20670830.0	15955820.0	189.69	191.78	190.68	192.55	192.43	191.9
875	36487930.0	38016810.0	52954070.0	211.15	216.88	219.05	215.18	220.45	222.3
876	28803760.0	33511990.0	36486560.0	303.22	305.64	308.10	305.17	310.35	317.0
877	35907770.0	25402270.0	21983410.0	151.78	153.80	153.89	153.92	154.72	154.2
878	39824200.0	41464880.0	38116290.0	173.68	167.25	167.81	175.15	170.02	171.7

879 rows × 12 columns

In [42]:
```
X_test
```

Out[42]:

	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	High_t-2	High_t
0	35987630.0	45970470.0	55204920.0	127.74	129.08	130.97	128.57	129.62	132.9
1	35421310.0	25674500.0	24725210.0	145.13	147.17	145.01	147.16	148.28	146.1
2	50278030.0	35678360.0	50061580.0	113.38	113.63	113.25	114.18	114.72	115.5
3	29773430.0	22526310.0	30684390.0	184.28	183.08	186.51	184.99	185.47	191.9

	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	High_t-2	High_t
4	26560420.0	26178840.0	31735810.0	109.51	110.23	109.95	110.73	110.98	110.4
...	...	...	...	...	...	...	...	...	.
372	64678220.0	53168580.0	56157370.0	112.18	111.94	111.07	112.68	112.80	111.9
373	33935720.0	69281360.0	54017920.0	208.76	216.42	213.90	210.16	221.37	218.0
374	24833800.0	25080500.0	20117070.0	145.87	145.50	147.97	146.18	148.49	149.3
375	53812480.0	32503750.0	45247890.0	284.69	277.95	276.28	286.95	281.68	277.2
376	69032740.0	24677880.0	12119710.0	282.23	280.53	284.69	282.65	284.25	284.8

377 rows × 12 columns

In [43]:
```
y_train
```

Out[43]:
```
0      169.88
1      316.70
2      171.05
3      159.29
4      115.59
        ...
874    192.45
875    209.55
876    317.83
877    153.21
878    167.88
Name: target, Length: 879, dtype: float64
```

In [44]:
```
y_test
```

Out[44]:
```
0      132.85
1      144.49
2      116.44
3      191.81
4      108.91
        ...
372    112.02
373    205.53
374    148.82
375    273.61
376    284.82
Name: target, Length: 377, dtype: float64
```

In [45]:
```
X_test_date
```

Out[45]:

	Date	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	High_t-2
0	2015-07-21	35987630.0	45970470.0	55204920.0	127.74	129.08	130.97	128.57	129.62
1	2017-06-28	35421310.0	25674500.0	24725210.0	145.13	147.17	145.01	147.16	148.28

	Date	Volume_t-3	Volume_t-2	Volume_t-1	Open_t-3	Open_t-2	Open_t-1	High_t-3	High_t-2
2	2015-09-25	50278030.0	35678360.0	50061580.0	113.38	113.63	113.25	114.18	114.72
3	2019-06-10	29773430.0	22526310.0	30684390.0	184.28	183.08	186.51	184.99	185.47
4	2016-04-08	26560420.0	26178840.0	31735810.0	109.51	110.23	109.95	110.73	110.98
...	...	...	...	...	...	...	...	...	...
372	2015-12-17	64678220.0	53168580.0	56157370.0	112.18	111.94	111.07	112.68	112.80
373	2019-08-02	33935720.0	69281360.0	54017920.0	208.76	216.42	213.90	210.16	221.37
374	2017-07-17	24833800.0	25080500.0	20117070.0	145.87	145.50	147.97	146.18	148.49
375	2020-04-22	53812480.0	32503750.0	45247890.0	284.69	277.95	276.28	286.95	281.68
376	2019-12-26	69032740.0	24677880.0	12119710.0	282.23	280.53	284.69	282.65	284.25

377 rows × 14 columns

In [46]:
```
#scale the data
#scaling the dataset using minmaxscaler
from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler(feature_range=(0,1))
X_train=scaler.fit_transform(X_train)
X_test=scaler.transform(X_test)
```

In [50]:
```
X_train[:10]
```

Out[50]:
```
array([[0.15426342, 0.12644697, 0.15606877, 0.26949793, 0.29492857,
        0.28369714, 0.28620284, 0.29059167, 0.28161701, 0.27437691,
        0.29461484, 0.27707898],
       [0.20767603, 0.19602151, 0.12662877, 0.78186072, 0.78785714,
        0.77656704, 0.77930961, 0.77786773, 0.7855027 , 0.78971724,
        0.78024251, 0.78358077],
       [0.11189569, 0.16666145, 0.08189741, 0.30466079, 0.29428571,
        0.27675975, 0.29888497, 0.28957094, 0.27527444, 0.30381869,
        0.28840942, 0.27785848],
       [0.04166663, 0.08063982, 0.03617348, 0.23858197, 0.23896429,
        0.23555587, 0.23422749, 0.23188202, 0.23056282, 0.24314969,
        0.23466476, 0.23661553],
       [0.06627179, 0.10158332, 0.0968568 , 0.09623898, 0.09571429,
        0.09232332, 0.09201667, 0.09056351, 0.0865656 , 0.09914735,
        0.09739658, 0.09084789],
       [0.06032861, 0.23632515, 0.11907544, 0.29656289, 0.29867857,
        0.29739673, 0.29019273, 0.29038049, 0.29806587, 0.29944614,
        0.29597004, 0.30067675],
       [0.10418206, 0.2385764 , 0.12959566, 0.31034731, 0.30571429,
        0.29935882, 0.30750597, 0.3035796 , 0.30723122, 0.31161638,
```

```
        0.28815977, 0.29947206],
       [0.1531356 , 0.10127863, 0.09360209, 0.10199748, 0.10714286,
        0.10300971, 0.10177764, 0.10147478, 0.09904164, 0.10475878,
        0.10784593, 0.10406406],
       [0.11052279, 0.34197841, 0.68454705, 0.05398596, 0.04967857,
        0.02102239, 0.04980229, 0.04797438, 0.02453389, 0.05480251,
        0.05149786, 0.02012543],
       [0.04760117, 0.07985845, 0.04767943, 0.1903905 , 0.19325   ,
        0.19175922, 0.18909195, 0.19105276, 0.18881338, 0.19468736,
        0.19222539, 0.19267973]])
```

In [51]:
```
#numpy array conversion
X_train=np.array(X_train)
X_test=np.array(X_test)
```

In [52]:
```
# reshape input to be [samples, time steps, features] which is required for LSTM
X_train =X_train.reshape(X_train.shape[0],X_train.shape[1] , 1)
X_test = X_test.reshape(X_test.shape[0],X_test.shape[1] , 1)
```

In [54]:
```
X_train.shape
```

Out[54]:
```
(879, 12, 1)
```

In [55]:
```
X_test.shape
```

Out[55]:
```
(377, 12, 1)
```

Design Steps

RNNs were not chosen because of the vanishing gradient problem. Long short-term memory (LSTM) is a deep learning system that avoids the vanishing gradient problem. LSTM is normally augmented by recurrent gates called “forget gates”.LSTM prevents backpropagated errors from vanishing or exploding. Instead, errors can flow backwards through unlimited numbers of virtual layers unfolded in space.

Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks introduced in 2014. They are used in the full form and several simplified variants. They have fewer parameters than LSTM, as they lack an output gate.

LSTMs and GRUs take 3D input so data was reshaped. We considered various RNN architectures but the following gave the best performance for our problem.

Step 1: Model Architecture

MODEL 1

LSTM LAYER 1 - 50 units --> Dropout 0.2 --> LSTM LAYER 2 - 50 units --> Dropout 0.2 --> LSTM LAYER 3 - 50 units --> Dropout 0.2 --> Dense Layer - 1 unit

This model uses three LSTM layers. 20 % of the nodes at each layer are unused to avoid overfitting and improve model performance.

MODEL 2

GRU Layer 75 units --> GRU Layer 30 units --> GRU Layer 30 units --> Dropout 0.2 --> Dense Layer - 1 layer

This model uses three GRU layers. 20 % of the nodes at the final GRU layer are unused to avoid overfitting and improve model performance.

Step 2: Optimizers considered

Adagrad - Resulted in poor model performance. Model did not train.

Stochastic Gradient Descents - Resulted in poor model performance. Model did not train.

Adam: Model performed well with this. It is also recommended as the best optimizer for LSTMs as referenced in [1]

Step 3: Number of Epochs

Epochs	Model 1 Training Loss	Model 2 Training Loss
100	13621	18898
256	4851	8759
512	4203	4312
800	1033	2159
1500	189	292

Step 4: Runtime

Model 1 - 13 min 26s for 1500 epochs Model 2 - 12 min 26s for 1500 epochs

Model 2 has a shorter run time, perhaps because of the smaller width in its 2nd and 3rd layer.

Step 5: Loss Metric

Mean Squared Error.

In [56]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import LSTM, SimpleRNN, GRU, Bidirectional
from keras import callbacks
from keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau, C
```

Model #1

In [57]:
```
model = tf.keras.models.Sequential([
    # Shape [batch, time, features] => [batch, time, lstm_units]
```

```
    tf.keras.layers.LSTM(50, return_sequences=True, input_shape=(12,1)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.LSTM(50, return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.LSTM(50),
    tf.keras.layers.Dropout(0.2),
    # Shape => [batch, time, features]
    tf.keras.layers.Dense(units=1, activation='linear')
])
```

In [58]:
```
model.compile(loss='mean_squared_error',optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=
```

In [59]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 12, 50)            10400
_________________________________________________________________
dropout (Dropout)            (None, 12, 50)            0
_________________________________________________________________
lstm_1 (LSTM)                (None, 12, 50)            20200
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 50)            0
_________________________________________________________________
lstm_2 (LSTM)                (None, 50)                20200
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
_________________________________________________________________
dense (Dense)                (None, 1)                 51
=================================================================
Total params: 50,851
Trainable params: 50,851
Non-trainable params: 0
_________________________________________________________________
```

In [60]:
```
%%time
history = model.fit(X_train,y_train,validation_split=0.05,epochs=1500,batch_size
```

```
Epoch 1/1500
14/14 [==============================] - 7s 142ms/step - loss: 33012.8359 - val_
loss: 33627.8398
Epoch 2/1500
14/14 [==============================] - 0s 31ms/step - loss: 31456.1758 - val_l
oss: 31396.8242
Epoch 3/1500
14/14 [==============================] - 0s 30ms/step - loss: 30037.9141 - val_l
oss: 30665.1504
Epoch 4/1500
14/14 [==============================] - 0s 31ms/step - loss: 29497.0449 - val_l
oss: 30262.4941
Epoch 5/1500
14/14 [==============================] - 0s 31ms/step - loss: 29174.7969 - val_l
oss: 29956.0996
Epoch 6/1500
```

```
s: 23.7803
Epoch 1491/1500
14/14 [==============================] - 1s 40ms/step - loss: 192.2093 - val_los
s: 31.8610
Epoch 1492/1500
14/14 [==============================] - 1s 39ms/step - loss: 215.0018 - val_los
s: 25.3716
Epoch 1493/1500
14/14 [==============================] - 1s 43ms/step - loss: 188.5737 - val_los
s: 26.7372
Epoch 1494/1500
14/14 [==============================] - 1s 43ms/step - loss: 165.1457 - val_los
s: 22.6986
Epoch 1495/1500
14/14 [==============================] - 1s 46ms/step - loss: 179.4910 - val_los
s: 24.8382
Epoch 1496/1500
14/14 [==============================] - 1s 40ms/step - loss: 171.4703 - val_los
s: 27.4761
Epoch 1497/1500
14/14 [==============================] - 1s 43ms/step - loss: 182.3687 - val_los
s: 28.1025
Epoch 1498/1500
14/14 [==============================] - 1s 40ms/step - loss: 169.7404 - val_los
s: 30.3197
Epoch 1499/1500
14/14 [==============================] - 1s 38ms/step - loss: 176.8685 - val_los
s: 32.3442
Epoch 1500/1500
14/14 [==============================] - 1s 40ms/step - loss: 189.4727 - val_los
s: 38.6774
CPU times: user 16min 31s, sys: 33 s, total: 17min 4s
Wall time: 13min 26s
```

In [62]:
```
print(history.history["loss"][-1])
```

```
189.47265625
```

In [63]:
```
print('Training MSE for Model 1', model.evaluate(X_train, y_train, verbose=0))
```

```
Training MSE for Model 1 24.01258659362793
```

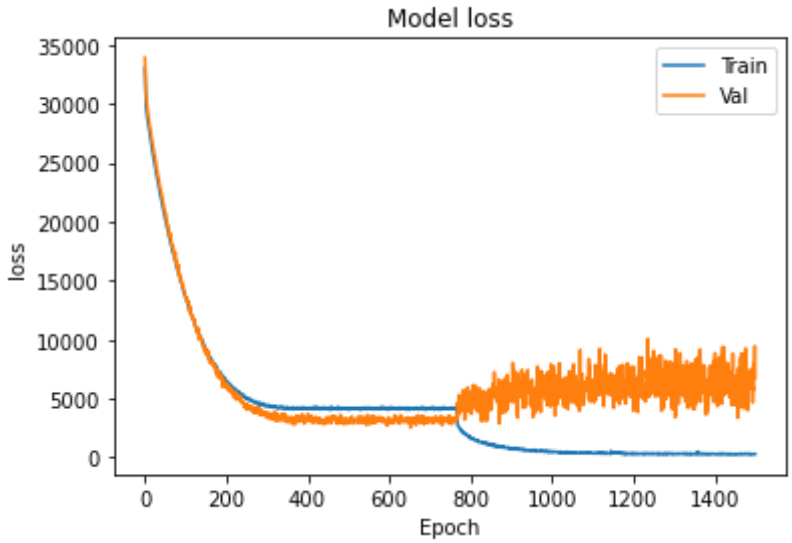
In [83]:
```
model.predict(X_train) - y_train.values
```

Out[83]:
```
array([[   2.95023071, -143.86976929,    1.78023071, ..., -144.99976929,
          19.62023071,    4.95023071],
       [ 147.15701782,    0.33701782,  145.98701782, ...,   -0.79298218,
         163.82701782,  149.15701782],
       [   4.44254028, -142.37745972,    3.27254028, ..., -143.50745972,
          21.11254028,    6.44254028],
       ...,
       [ 143.92276489,   -2.89723511,  142.75276489, ...,   -4.02723511,
         160.59276489,  145.92276489],
       [ -14.04496277, -160.86496277,  -15.21496277, ..., -161.99496277,
           2.62503723,  -12.04496277],
       [   0.6628772 , -146.1571228 ,   -0.5071228 , ..., -147.2871228 ,
          17.3328772 ,    2.6628772 ]])
```

In [64]:

```
#save for the best model
model.save('models/Group3_RNN_model.h5')
```

In [ ]:
```
loss_plot(history)
```



Comment on Model #1 Training Output

- The training and validation losses start at a high value of approximately 35000
- There is a drastic decrease in the first 200 epochs
- The gap between training and validation loss remains steady until about 800 epochs
- The final training loss achieved is approximately 189

In [85]:
```
y_test=np.array(y_test)
y_pred = model.predict(X_test, verbose = 0)
```

In [87]:
```
y_pred[:20]
```

Out[87]:
```
array([[130.02972 ],
       [147.80804 ],
       [114.17725 ],
       [188.1433  ],
       [111.11356 ],
       [114.54795 ],
       [116.98694 ],
       [152.87953 ],
       [116.82906 ],
       [160.26447 ],
       [172.04114 ],
       [248.34659 ],
       [176.55379 ],
       [154.58133 ],
       [190.42662 ],
       [285.1028  ],
       [233.29152 ],
       [333.38208 ],
```

```
       [120.063614],
       [195.56525 ]], dtype=float32)
```

In [88]:
```
#calculate test loss/mse
mean_squared_error(y_pred, y_test)
```

Out[88]:
```
21.196446387090102
```

In [ ]:
```
score = model.evaluate(X_test, y_test, verbose=False)
print('Metric Names',model.metrics_names)
print('Test Score:', score)
```

In [90]:
```
score = model.evaluate(X_train, y_train, verbose=False)
print('Metric Names',model.metrics_names)
print('Training Score:', score)
```

```
Metric Names ['loss']
Training Score: 24.01258659362793
```

In [99]:
```
result_array=pd.DataFrame({'y_test':y_test, 'y_predicted':y_pred.ravel(),'Date':
```

In [101…
```
#result_array = result_array.sort_values(by=['Date'])
result_array=result_array.reset_index(drop=True, inplace=False)
result_array
```

Out[101…

	y_test	y_predicted	Date
0	132.85	130.029724	2015-07-21
1	144.49	147.808044	2017-06-28
2	116.44	114.177254	2015-09-25
3	191.81	188.143295	2019-06-10
4	108.91	111.113564	2016-04-08
...	...	...	...
372	112.02	110.918327	2015-12-17
373	205.53	215.097290	2019-08-02
374	148.82	149.203873	2017-07-17
375	273.61	281.391571	2020-04-22
376	284.82	287.908386	2019-12-26

377 rows × 3 columns

In [103…
```
result_array['Date'] =pd.to_datetime(result_array.Date)
```

In [104…
```
result_array=result_array.sort_values(by='Date')
result_array
```

Out[104…

	y_test	y_predicted	Date
45	125.72	124.952179	2015-07-15
357	127.74	126.632896	2015-07-16
0	132.85	130.029724	2015-07-21
128	125.32	126.351768	2015-07-24
241	123.38	124.806816	2015-07-28
...	...	...	...
49	319.25	326.039917	2020-05-29
34	344.72	347.023743	2020-06-12
76	351.46	345.197784	2020-06-16
236	365.00	348.616974	2020-06-24
232	353.25	348.882324	2020-06-29

377 rows × 3 columns

Comments about y_true/y_pred dataframe

The model has a good output. The predicted values of y and close to the true values. From the 10 values shown above, the largest value of |y_pred - y_true| is 12, although most have a difference of less than 5.

In [105…

```
result_array=result_array.reset_index(drop=True, inplace=False)
```

In [106…

```
result_array.iloc[0:,0:2].plot.line(figsize=(13,8))
plt.xticks(np.arange(0, 377, step=20), result_array["Date"].dt.date.iloc[lambda
plt.xlabel('Date')
plt.ylabel('Opening price')
plt.title('Stock price over time')
```

Out[106…

```
Text(0.5, 1.0, 'Stock price over time')
```

Comments about Stock Price over time plot

The plots of y_test(y_true) and y_pred mostly overlap. The largest gaps in both plots occurs sometime in 2020. This could be due to the Coronavirus pandemic.

Model #2

In [108…

```
model_1 = tf.keras.models.Sequential([
    # Shape [batch, time, features] => [batch, time, lstm_units]
    tf.keras.layers.GRU(75, return_sequences=True, input_shape=(12,1)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.GRU(30, return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.GRU(30),
    tf.keras.layers.Dropout(0.2),
    # Shape => [batch, time, features]
    tf.keras.layers.Dense(units=1)
])
```

In [109…

```
model_1.compile(optimizer='adam', loss='mean_squared_error')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=
```

In [110…

```
model_1.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
```

```
14/14 [==============================] - 1s 36ms/step - loss: 383.2633 - val_los
s: 38.9415
Epoch 1498/1500
14/14 [==============================] - 1s 40ms/step - loss: 364.9526 - val_los
s: 36.0181
Epoch 1499/1500
14/14 [==============================] - 1s 36ms/step - loss: 328.2225 - val_los
s: 44.9635
Epoch 1500/1500
14/14 [==============================] - 1s 40ms/step - loss: 292.2680 - val_los
s: 42.8146
CPU times: user 15min 23s, sys: 37.4 s, total: 16min
Wall time: 12min 26s
```

In [112…

```
print(history.history["loss"][-1])
```

```
292.26800537109375
```

In [113…

```
print('training MSE', model_1.evaluate(X_train, y_train, verbose=0))
```

```
training MSE 89.08911895751953
```

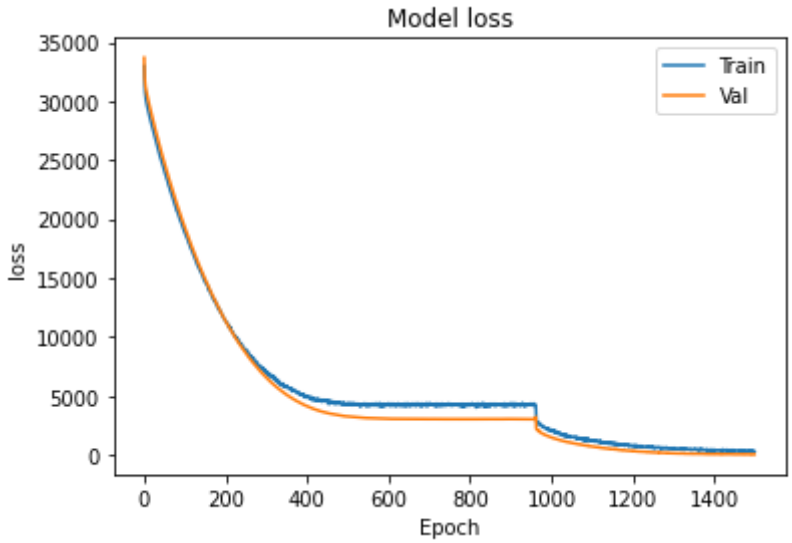
In [114…

```
print(history.history.keys())
```

```
dict_keys(['loss', 'val_loss'])
```

In [115…

```
loss_plot(history)
```



Comment on Model #2 Training Output

- The training and validation losses start at a high value of approximately 35000
- There is a drastic decrease in the first 200 epochs
- The training and validation losses continue to decrease
- The final training loss achieved is approximately 292

In [116…

```
y_test=np.array(y_test)
```

```
y_pred = model_1.predict(X_test, verbose=0)
```

In [117…

```
y_pred[:20]
```

Out[117…

```
array([[125.660934],
       [144.5896  ],
       [110.064964],
       [184.48605 ],
       [107.21594 ],
       [113.757805],
       [112.20062 ],
       [149.78496 ],
       [111.87024 ],
       [156.93988 ],
       [168.75972 ],
       [245.05467 ],
       [173.32153 ],
       [151.28717 ],
       [186.90234 ],
       [283.3356  ],
       [228.92291 ],
       [302.5869  ],
       [116.39398 ],
       [191.75362 ]], dtype=float32)
```

In [118…

```
#calculate test loss/mse
mean_squared_error(y_pred, y_test)
```

Out[118…

```
52.513931808736615
```

In [119…

```
score = model_1.evaluate(X_test, y_test, verbose=False)
print('Metric Names',model_1.metrics_names)
print('Test Score for Model 2:', score)
```

```
Metric Names ['loss']
Test Score for Model 2: 52.5139274597168
```

In [120…

```
score = model_1.evaluate(X_train, y_train, verbose=False)
print('Metric Names',model_1.metrics_names)
print('Training Score for Model 2:', score)
```

```
Metric Names ['loss']
Training Score for Model 2: 89.08911895751953
```

In [121…

```
result_array=pd.DataFrame({'y_test':y_test, 'y_predicted':y_pred.ravel(),'Date':
```

In [ ]:

```
result_array=result_array.reset_index(drop=True, inplace=False)
result_array
```

In [124…

```
result_array['Date'] =pd.to_datetime(result_array.Date)
```

In [125…

```
result_array=result_array.sort_values(by='Date')
```

```
result_array
```

Out[125…

	y_test	y_predicted	Date
45	125.72	120.705017	2015-07-15
357	127.74	122.633720	2015-07-16
0	132.85	125.660934	2015-07-21
128	125.32	123.803505	2015-07-24
241	123.38	120.971870	2015-07-28
...	...	...	...
49	319.25	302.511902	2020-05-29
34	344.72	302.652496	2020-06-12
76	351.46	302.647675	2020-06-16
236	365.00	302.658325	2020-06-24
232	353.25	302.659149	2020-06-29

377 rows × 3 columns

Comments about y_true/y_pred dataframe

The model has a fairly good output, though not as good as Model #1. The predicted values of y are close to the true values. From the 10 values shown above, there are much larger values of |y_pred - y_true| than model #1. Some datapoints have a difference of over 40.

In [126…

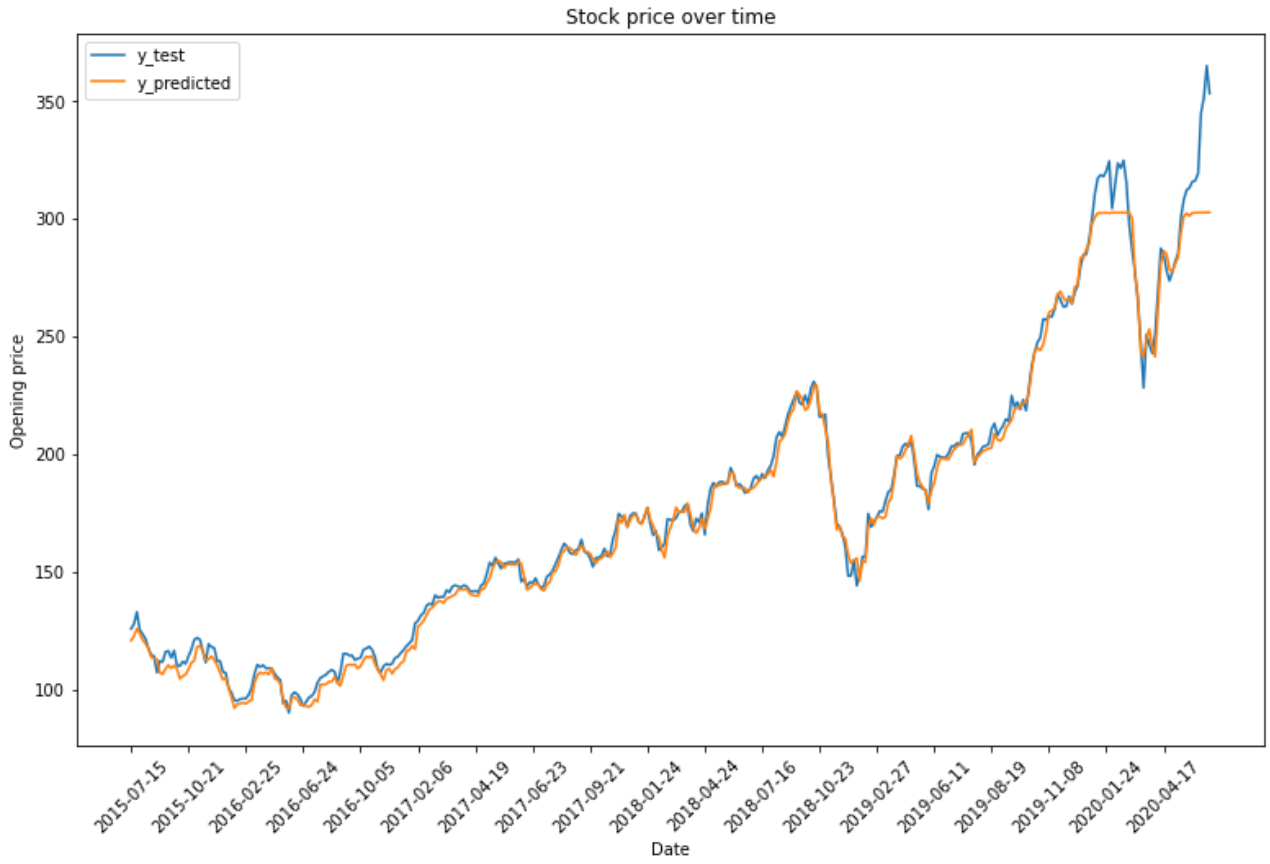
```
result_array=result_array.reset_index(drop=True, inplace=False)
```

In [127…

```
result_array.iloc[0:,0:2].plot.line(figsize=(13,8))
plt.xticks(np.arange(0, 377, step=20), result_array["Date"].dt.date.iloc[lambda
plt.xlabel('Date')
plt.ylabel('Opening price')
plt.title('Stock price over time')
```

Out[127…

```
Text(0.5, 1.0, 'Stock price over time')
```



Comments about Stock Price over time plot

The plots of y_test(y_true) and y_pred mostly overlap. There is significant variation in the plots in 2019 and 2020. The plot for Model #1 seems to be more accurate.

Final Network Architecture

Model #1 is chosen because it has a better performance.

MODEL 1

LSTM LAYER 1 - 50 units --> Dropout 0.2 --> LSTM LAYER 2 - 50 units --> Dropout 0.2 --> LSTM LAYER 3 - 50 units --> Dropout 0.2 --> Dense Layer - 1 unit

This model uses three LSTM layers. 20 % of the nodes at each layer are unused to avoid overfitting and improve model performance.

Optimizer - **Adam**

Loss Metric - **Mean Squared Error**

Activation Function in Dense layer - **Linear**

Batch Size - **64**

Number of Epochs - **1500**

The model utilises Early Stopping in order to converge faster and avoid overfitting.

Effect of Adding More Features

After increasing the features to 40 i.e (using data from the latest 10 days) we observed the following:

- The model trained for a longer time with the same number of epochs.
- The model performance was significantly improved. The training loss was approximately 15 using Model #1 as compared to 189 using Model #1 with 12 Features
- The final plot of predicted values against true values in the test set are almost identical.
- External Resources suggest that the prices and volumes are not the best features for stock prediction. Return value is suggested to be a better input.

References

1 "LSTM Optimizer Choice ?" https://deepdatascience.wordpress.com/2016/11/18/which-lstm-optimizer-to-use/

In [ ]:

NLP

July 19, 2021

1 ECE 657 ASSIGNMENT 2: Problem 3

1.1 NLP (Classification of IMDB Movie Reviews)

Jubilee Imhanzenobe: 20809735
Olohireme Ajayi: 20869827
Harnoor Singh: 20870613

[1]:
```
# import required packages
import pandas as pd
from sklearn.metrics import confusion_matrix, f1_score, accuracy_score
from sklearn.model_selection import train_test_split
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, LSTM
from gensim.models import Word2Vec, KeyedVectors
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
import numpy as np
from utils import *
```

[2]:
```
# Importing the training and testing datasets
# extract data if it hasnt already been extracted and saved as csv
if not 'train_data_NLP.csv' in os.listdir("data"):
    X_train, y_train = load_data()
    train_data = pd.DataFrame([X_train, y_train]).transpose()
    train_data.columns = ["Review", "Rating"]

    X_test, y_test = load_data(False)
    test_data = pd.DataFrame([X_test, y_test]).transpose()
    test_data.columns = ["Review", "Rating"]

    # saving the datasets as csv files for ease of loading
    train_data.to_csv("data/train_data_NLP.csv", index = False)
    test_data.to_csv("data/test_data_NLP.csv", index = False)
```

[3]:
```
# loading the dataset from saved file
train_data = pd.read_csv("data/train_data_NLP.csv")
```

1.1.1 Data Preprocessing

We first preprocess the reviews to make them ready for our NLP embedding techniques.

During preprocessing, all special characters were removed and each review was tokenized i.e separated into a list containing the individual words.

Next, the words were stemmed using potterstemmer to reduce every word to its root word. The Porter stemming is a process for removing the commoner morphological and inflexional endings from words in English. It is mainly used as part of the term normalisation process usually done when setting up Information Retrieval systems.

Finally, the tokenized words were joined back together to get a preprocessed review.

[4]:
```
""" Data Preprocessing """
# preprocessing the reviews in the training data
train_data['Token'] = train_data['Review'].apply(preprocess_sentence)
```

[5]:
```
# Before preprocessing
train_data['Review'][0]
```

[5]:
```
'Bromwell High is a cartoon comedy. It ran at the same time as some other
programs about school life, such as "Teachers". My 35 years in the teaching
profession lead me to believe that Bromwell High\'s satire is much closer to
reality than is "Teachers". The scramble to survive financially, the insightful
students who can see right through their pathetic teachers\' pomp, the pettiness
of the whole situation, all remind me of the schools I knew and their students.
When I saw the episode in which a student repeatedly tried to burn down the
school, I immediately recalled … at … High. A classic line:
INSPECTOR: I\'m here to sack one of your teachers. STUDENT: Welcome to Bromwell
High. I expect that many adults of my age think that Bromwell High is far
fetched. What a pity that it isn\'t!'
```

[6]:
```
# After preprocessing
train_data['Token'][0]
```

[6]:
```
'bromwel high cartoon comedi ran time program school life teacher year teach
profess lead believ bromwel high satir much closer realiti teacher scrambl
surviv financi insight student see right pathet teacher pomp petti whole situat
remind school knew student saw episod student repeatedli tri burn school immedi
recal high classic line inspector im sack one teacher student welcom bromwel
high expect mani adult age think bromwel high far fetch piti isnt'
```

1.1.2 Model 1 (Using Count Vectorizer)

Count Vectorizer is an embedding method for textual data it is also called Term Frequency Embedding. It transforms a given text into a vector on the basis of the frequency (count) of each word that occurs in the entire text. It creates a sparse matrix with each unique word being represented by a column in the matrix and the value of each element is the frequency of the particular word in the text sample.

A 5000 feature vectorization was chosen so that is large enough to properly represent the large sized corpus yet small enough for good running efficiency.

A 5 layer ANN (1 input, 3 hidden, 1 output) was used in the classification of the embedded movie reviews. Due to the large input dimensions, dropout layers were added between hidden layers to prevent overfitting of the network to the training set. The network was also trained for 30 epochs to observe the learning pattern of the network and a batch size of 128 was used.

[7]:
```
epochs = 30
```

[8]:
```
# creating the word embedding using count vectorizer
X, CV = word_vectorizer(train_data.Token, "CountVectorizer", 5000)
y = train_data.Rating

# splittint the data into training and validatuion set
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.10,␣
 ↪random_state=0)
```

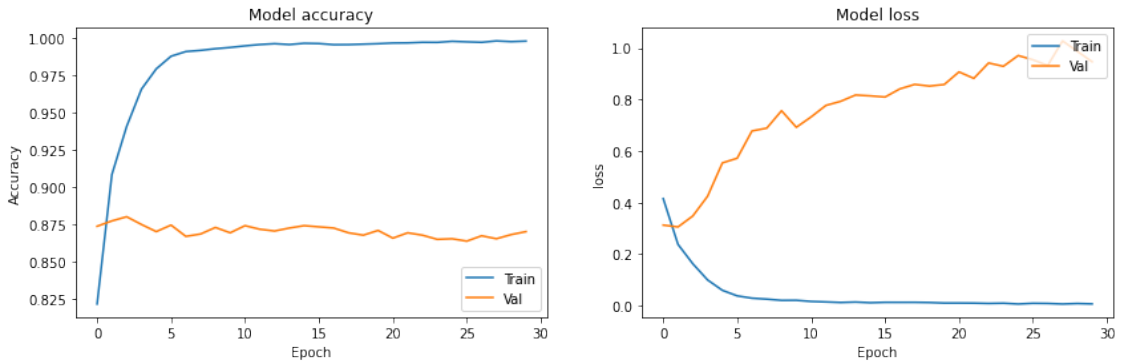
[9]:
```
""" Building the ANN model """
model = Sequential()

# Input - Layer
model.add(Dense(128, activation = "relu", input_shape=(X_train.shape[1], )))

# Hidden - Layers
model.add(Dropout(0.3, seed=None))
model.add(Dense(32, activation = "relu"))
model.add(Dropout(0.2, seed=None))
model.add(Dense(16, activation = "relu"))

# Output- Layer
model.add(Dense(1, activation = "sigmoid"))
model.summary()

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

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               640128
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                4128
_________________________________________________________________
dropout_1 (Dropout)          (None, 32)                0
```

```
_________________________________________________________________
dense_2 (Dense)              (None, 16)                528
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 17
=================================================================
Total params: 644,801
Trainable params: 644,801
Non-trainable params: 0
_________________________________________________________________
```

[10]:
```
%%capture
history = model.fit(X_train,y_train,
                    epochs = epochs,
                    validation_data = (X_val, y_val),
                    verbose = 1, # print result every epoch
                    batch_size = 128)
```

[11]:
```
accuracy_loss_plot(history)
```



[12]:
```
print_final_result(history)
```

```
 ******************** Training set Evaluation ********************
Final Train accuracy:  0.9981
Final Train loss:  0.0064

 ******************** Validation set Evaluation ********************
Final Val accuracy:  0.87
Final Val loss:  0.9468
```

1.1.3 Model 2 (Using TF-IDF Vectorizer)

Term Frequency-Inverse Document Frequency Vectorizer (TF-IDF) is similar to Count Vectorizer in that it embeds words based on the frequency of a word in the corpus but it also provides a numerical representation of the importance of the words. The term frequency (TF) refers to the

count of a word in a sentence and the document frequency (DF) refers to the number of documents in the corpus that contain the particular word.

A 5000 feature vectorization was chosen so that is large enough to properly represent the large sized corpus yet small enough for good running efficiency.

A 5 layer ANN (1 input, 3 hidden, 1 output) was used in the classification of the embedded movie reviews. Due to the large input dimensions, dropout layers were added between hidden layers to prevent overfitting of the network to the training set. The network was also trained for 30 epochs to observe the learning pattern of the network and a batch size of 128 was used.

[13]:
```
# creating the word embedding using count vectorizer
X2, TFIDF = word_vectorizer(train_data.Token, "TFIDF", 5000)

# splittint the data into training and validatuion set
X_train2, X_val2, y_train2, y_val2 = train_test_split(X2, y, test_size=0.10,␣
 ↪random_state=0)
```

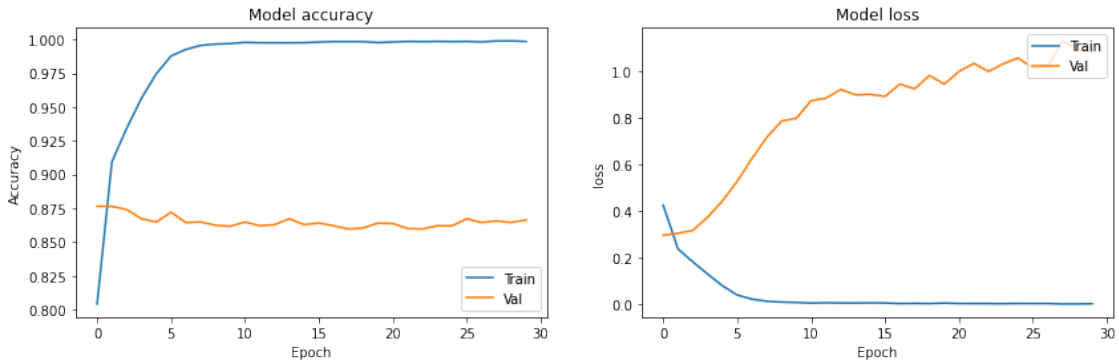
[14]:
```
""" Building the ANN model """
model2 = Sequential()

# Input - Layer
model2.add(Dense(128, activation = "relu", input_shape=(X_train2.shape[1], )))

# Hidden - Layers
model2.add(Dropout(0.3, seed=None))
model2.add(Dense(32, activation = "relu"))
model2.add(Dropout(0.2, seed=None))
model2.add(Dense(16, activation = "relu"))

# Output- Layer
model2.add(Dense(1, activation = "sigmoid"))
model2.summary()

#Compiling the ANN
model2.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 128)               640128
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 32)                4128
_________________________________________________________________
dropout_3 (Dropout)          (None, 32)                0
```

```
_________________________________________________________________
dense_6 (Dense)              (None, 16)                528
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 17
=================================================================
Total params: 644,801
Trainable params: 644,801
Non-trainable params: 0
_________________________________________________________________
```

[15]:
```
%%capture
history2 = model2.fit(X_train2,y_train2,
                      epochs = epochs,
                      validation_data = (X_val2, y_val2),
                      verbose = 1, # print result every epoch
                      batch_size = 128)
```

[16]:
```
accuracy_loss_plot(history2)
```



[17]:
```
print_final_result(history2)
```

```
 ******************** Training set Evaluation ********************
Final Train accuracy:  0.9986
Final Train loss:  0.004

 ******************** Validation set Evaluation ********************
Final Val accuracy:  0.8664
Final Val loss:  1.0852
```

1.1.4 Preprocessing for Word2Vec and Doc2Vec

For Word2Vec and Doc2Vec, the the reviews were preprocessed slightly differently. Like before, all special characters were removed and each review was tokenized i.e separated into a list containing then individual words. Finally, the words were Lemmatized to reduce every word to its root word.

Lemmatization like Porter Stemmimg aims to reduce words to their root words but lemmatization considers the vocabulary and morphological analysis of words while aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma . e.g if the word **saw** is stemmed, the result may be **s** whereas if it is lemmatized the result will be **see** or **saw** depending on whether the use of the token was as a verb or a noun.

The lemmatized tokens of the reviews were then used to get the Word2Vec feature embedding.

1.1.5 Model 3 (Using Word2Vec)

Word2Vec unlike the previously used vectorizers actually embeds the meaning of a word by inferring it from its context. It uses a linguistic technique which uses the context in which a word appears to infer its meaning. Word2Vec only gets the embeddings of words and is usually very good for finding relationships between words but it doesn't embed sentences.

Here, we got the sentence embeddings by averaging the embedding of every word in the sentence or review.

[18]:
```
sentences = train_data["Review"].apply(w2v_preprocess)
w2v_model = Word2Vec(sentences=sentences,
                     vector_size=100,
                     window=5,
                     min_count=5,
                     workers=8)
```

[19]:
```
w2v_model.wv.most_similar('review', topn=10)
```

[19]:
```
[('comment', 0.9242464900016785),
 ('imdb', 0.8326272368431091),
 ('reviewers', 0.760803759098053),
 ('reviewer', 0.7367420792579651),
 ('summary', 0.7364093661308289),
 ('user', 0.7301311492919922),
 ('post', 0.7195536494255066),
 ('negative', 0.7160252332687378),
 ('website', 0.7016050219535828),
 ('critics', 0.7012093663215637)]
```

The cell above shows the ability of Word2Vec to find similarity between words. It shows a high similarity between **review**, **comments**, **reviewer**, **summary**, **post**, **opinion** which shows some level of semantical correctness in English grammar.

[20]:
```
df_train = get_sentence_embedding(sentences, w2v_model)
X3 = df_train.iloc[:].values
```

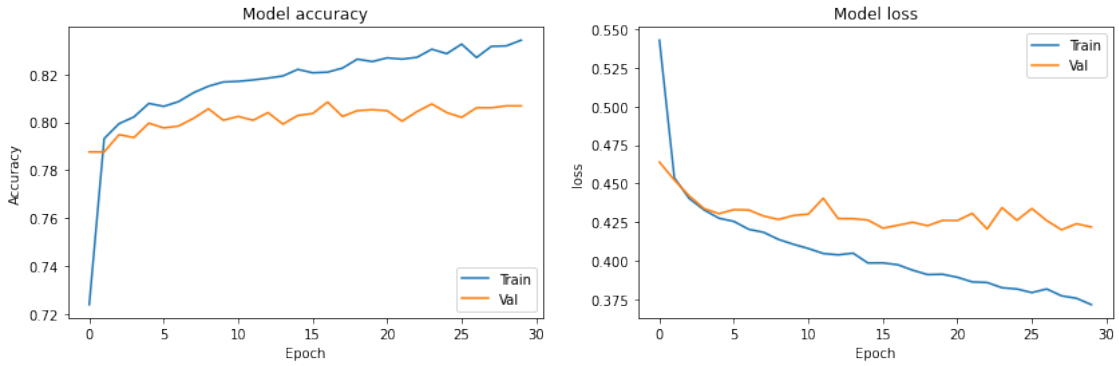
[21]:
```
# splitting the data into training and validatuion set
X_train3, X_val3, y_train3, y_val3 = train_test_split(X3, y, test_size=0.10,␣
↪random_state=0)
```

[22]:
```
""" Building the ANN model """
model3 = Sequential()

# Input - Layer
model3.add(Dense(128, activation = "relu", input_shape=(X_train3.shape[1], )))

# Hidden - Layers
model3.add(Dropout(0.3, seed=None))
model3.add(Dense(32, activation = "relu"))
model3.add(Dropout(0.2, seed=None))
model3.add(Dense(16, activation = "relu"))

# Output- Layer
model3.add(Dense(1, activation = "sigmoid"))
model3.summary()

#Compiling the ANN
model3.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 128)               12928
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_9 (Dense)              (None, 32)                4128
_________________________________________________________________
dropout_5 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_10 (Dense)             (None, 16)                528
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 17
=================================================================
Total params: 17,601
Trainable params: 17,601
Non-trainable params: 0
_________________________________________________________________
```

[23]:
```
%%capture
history3 = model3.fit(X_train3,y_train3,
                      epochs=epochs,
                      validation_data=(X_val3, y_val3),
                      verbose=1, # print result every epoch
                      batch_size=128)
```

[24]:
```
accuracy_loss_plot(history3)
```



[25]:
```
print_final_result(history3)
```

```
 ******************** Training set Evaluation ********************
Final Train accuracy:  0.8342
Final Train loss:  0.3716

 ******************** Validation set Evaluation ********************
Final Val accuracy:  0.8068
Final Val loss:  0.4219
```

1.1.6 Model 4 (Using Doc2Vec)

Unlike Word2Vec that trains on a single word, Doc2Vec trains on texts of variable length and with Doc2Vec, relationships between sentences or documents can be measured. Doc2Vec is a generalized extension of Word2Vec that is applied to a document as a whole instead of individual words. This model was developed by Le and Mikolov. It aims to create a numerical representation of a document rather than a word (Le & Mikolov, 2014). Doc2Vec operates on the logic that the meaning of a word also depends on its context and also the document that it occurs in. The vectors generated by Doc2Vec can be used for finding similarities between documents.
The documents have to be tagged in the Doc2Vec training. The tagging can be done according to authors or topics or using any other criteria. In this model, the reviews were tagged serially. This method of tagging was chosen because the aim of the embeddings is to perform classification and we want the model to be robust and have low generalization error.

[26]:
```
""" Using Doc2Vec """
tagged_data = [TaggedDocument(d, [i]) for i, d in enumerate(sentences)]

d2v_model = Doc2Vec(tagged_data, vector_size = 100, window = 5, min_count = 5,␣
 ↪epochs = 20, workers=8)
embed = sentences.apply(d2v_model.infer_vector)
```

```
X4 = np.stack( embed, axis=0)
```

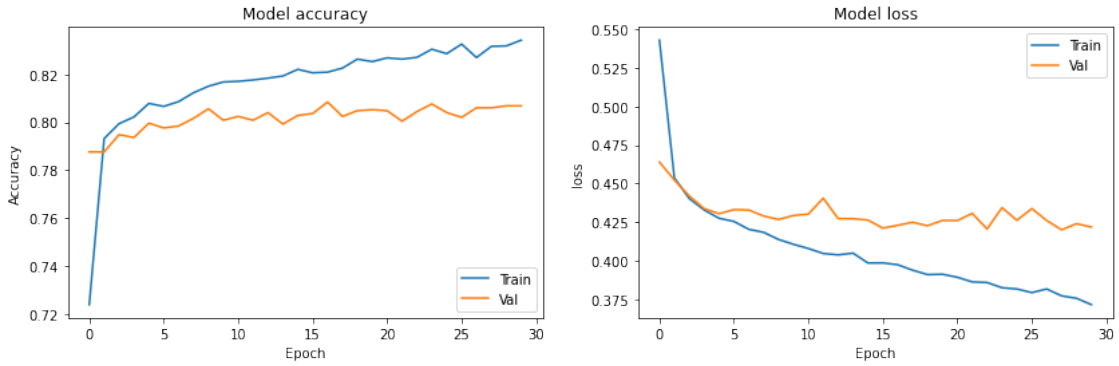
[27]:
```
# splitting the data into training and validatuion set
X_train4, X_val4, y_train4, y_val4 = train_test_split(X4, y, test_size=0.10,␣
 ↪random_state=0)
```

[28]:
```
model4 = Sequential()

# Input - Layer
model4.add(Dense(128, activation = "relu", input_shape=(X_train4.shape[1], )))

# Hidden - Layers
model4.add(Dropout(0.3, seed=None))
model4.add(Dense(32, activation = "relu"))
model4.add(Dropout(0.2, seed=None))
model4.add(Dense(16, activation = "relu"))

# Output- Layer
model4.add(Dense(1, activation = "sigmoid"))
model4.summary()

#Compiling the ANN
model4.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 128)               12928
_________________________________________________________________
dropout_6 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_13 (Dense)             (None, 32)                4128
_________________________________________________________________
dropout_7 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_14 (Dense)             (None, 16)                528
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 17
=================================================================
Total params: 17,601
Trainable params: 17,601
Non-trainable params: 0
_________________________________________________________________
```

[29]:
```
%%capture
history4 = model4.fit(X_train4,y_train4,
                      epochs=epochs,
                      validation_data=(X_val4, y_val4),
                      verbose=1, # print result every epoch
                      batch_size=128)
```

[30]:
```
accuracy_loss_plot(history3)
```



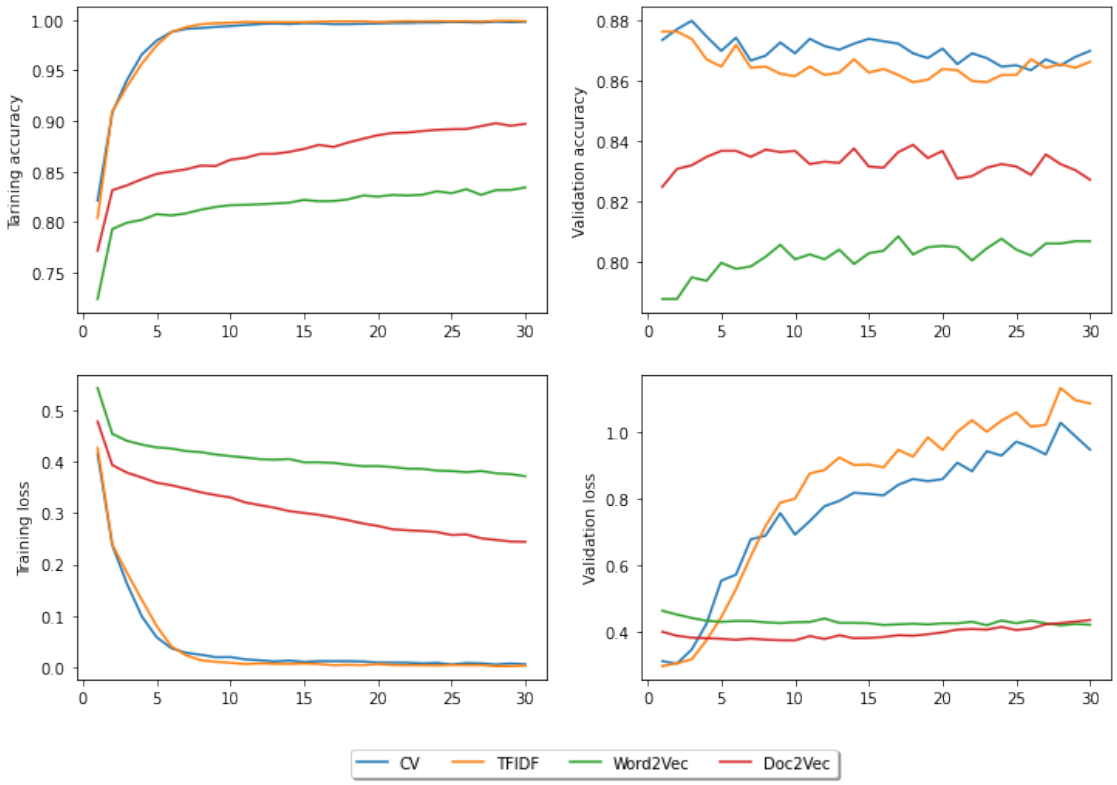
[31]:
```
print_final_result(history4)
```

```
 ********************* Training set Evaluation ********************
Final Train accuracy:  0.8972
Final Train loss:  0.2443

 ********************* Validation set Evaluation ********************
Final Val accuracy:  0.8272
Final Val loss:  0.4361
```

1.1.7 Comparing the performance of the models

[32]:
```
history_list = [history, history2, history3, history4]
labels = ["CV", "TFIDF", "Word2Vec", "Doc2Vec"]
```

[33]:
```
result = compare_models(history_list, labels, epochs)
```



From the plots, it can be seen that Count Vectorizer and TFIDF performed better than Word2Vec and Doc2vec in terms of training accuracy and validation accuracy. Although Count Vectorizer and TFIDF had lower training losses compared to Word2Vec and Doc2Vec, both of them had higher validation losses that Word2Vec and Doc2Vec. The contradiction in validation loss is actually due to the difference in dimensionality between CV/TFIDF (5000) and Word2Vec/Doc2Vec (100).

Also, as the number of epoch increased beyond 10, the training accuracy, training loss and validation accuracy for Count Vectorizer and TFID remained considerably constant but the validation loss for both embedding methods seemed to increase slightly.

Also, Word2Vec and Doc2Vec had lower generalization error compared to CV and TFIDF.

[34]:
```
result
```

[34]:

	Train Accuracy	Train Loss	Val Accuracy	Val Loss
CV	99.81	0.0064	87.00	0.9468
TFIDF	99.86	0.0040	86.64	1.0852
Word2Vec	83.42	0.3716	80.68	0.4219
Doc2Vec	89.72	0.2443	82.72	0.4361

The table above shows the final evaluation of the different embedding methods used after 30 epochs. From the results, CV and TFIDF have comparatively similar performances on the training and validation set but CV seemed to perform slightly better on the validation set and as such will be used for evaluating the test set. On the other hand, Doc2Vec preformed better than Word2Vec

on the training and validation sets> This in indicative of their design as Word2Vec embeds words individually and is better for finding similarity and relationships between words unlike Doc2Vec which can embed entire documents and find similarity and relationships between documents.

1.1.8 Evaluating models on the test set

[35]:
```
# Loading the test set data
test_data = pd.read_csv("data/test_data_NLP.csv")
```

Model 1 (Count Vectorizer)

[36]:
```
# preprocessing the reviews in the training data
test_data['Token'] = test_data['Review'].apply(preprocess_sentence)

# Vectorizing using Bag of Words Model
X_test = CV.transform(test_data.Token).toarray()
y_test = test_data.Rating

# predicting the test set results
y_pred = (model.predict(X_test) >= 0.5).astype("int32")

accuracy = accuracy_score(y_pred, y_test)
F_score = f1_score(y_test, y_pred, average = "binary")
CM = confusion_matrix(y_pred, y_test)

print("Test accuracy: ", round(accuracy * 100, 2))
print("Test F_score: ", round(F_score, 4))
print("Test confusion matrix: \n", CM)
```

```
Test accuracy:  85.59
Test F_score:  0.8534
Test confusion matrix:
 [[10914  2017]
 [ 1586 10483]]
```

Model 2 (TFIDF)

[37]:
```
# Vectorizing using Bag of Words Model
X_test = TFIDF.transform(test_data.Token).toarray()

# predicting the test set results
y_pred = (model2.predict(X_test) > 0.5).astype("int32")

accuracy2 = accuracy_score(y_pred, y_test)
F_score2 = f1_score(y_test, y_pred, average = "binary")
CM2 = confusion_matrix(y_pred, y_test)

print("Test accuracy: ", round(accuracy2 * 100, 2))
print("Test F_score: ", round(F_score2, 4))
```

```
print("Test confusion matrix: \n", CM2)
```

```
Test accuracy:  84.72
Test F_score:  0.846
Test confusion matrix:
 [[10682  2003]
 [ 1818 10497]]
```

Model 3 (Word2Vec)

[38]:
```
# preprocess the reviews
test_sentences = test_data["Review"].apply(w2v_preprocess)

# get Word2Vec embeddings of test data
df_test = get_sentence_embedding(sentences, w2v_model)
X_test = df_test.iloc[:].values

# predicting the test set results
y_pred = (model3.predict(X_test) > 0.5).astype("int32")

accuracy3 = accuracy_score(y_pred, y_test)
F_score3 = f1_score(y_test, y_pred, average = "binary")
CM3 = confusion_matrix(y_pred, y_test)

print("Test accuracy: ", round(accuracy * 100, 2))
print("Test F_score: ", round(F_score, 4))
print("Test confusion matrix: \n", CM)
```

```
Test accuracy:  85.59
Test F_score:  0.8534
Test confusion matrix:
 [[10914  2017]
 [ 1586 10483]]
```

Model 4 (Doc2Vec)

[39]:
```
# get Doc2Vec embeddings of test data
test_embed = test_sentences.apply(d2v_model.infer_vector)
X_test = np.stack(test_embed, axis=0)

# predicting the test set results
y_pred = (model4.predict(X_test) > 0.5).astype("int32")

accuracy4 = accuracy_score(y_pred, y_test)
F_score4 = f1_score(y_test, y_pred, average = "binary")
CM4 = confusion_matrix(y_pred, y_test)

print("Test accuracy: ", round(accuracy * 100, 2))
print("Test F_score: ", round(F_score, 4))
```

```
print("Test confusion matrix: \n", CM)
```

```
Test accuracy:  85.59
Test F_score:  0.8534
Test confusion matrix:
 [[10914  2017]
 [ 1586 10483]]
```

On the test set, Word2Vec, Doc2Vec and CV all had the same accuracy **85.59**. TFIDF on the other hand had an accuracy of **84.72**. This butresses the previous point about the generalization error of CV and TFIDF being greater than that of Word2Vec and Doc2Vev