# ICP10

**Answer 1)** The three mistake in given code are as below

**1.** Input data shape or dimension (**input_dim**) is not provided in input layer definition:
It's really necessary to define and provide the input dimension while defining the input layer of any NN model. If the neural network don't know the correct input dimension then it won't be properly able to process data and do the predictions.
The input dimension should be equal to the number of columns of the train dataset after performing train test split, which is 2000.
Corrected code is as below.

```
# Defining input_dim
# input dimension should be equal to the number of columns of the train dataset after performing train test split,
# which is 2000.
input_dim = np.prod(X_train.shape[1:])
# Number of features
print('input_dim: ',input_dim)
```

Output

```
input_dim:  2000
```

As seen in above output the input dimension = 2000
Lets use input_dim now in input layer of our model.

```
# Creating Model
model = Sequential()
# Adding dense layer of 300 units and activation= relu
model.add(layers.Dense(300,input_dim= input_dim, activation='relu'))
```

**2.** In the given code the 5 neurons are defined while it should be **3**:
The neurons in the output layer should be **3** as there are three classes [**pos, neg, unsup**] in the column(label) of dataset which is the target.
**3.** The activation function in the output layer should be **softmax** instead of sigmoid.
The activation = 'softmax' needs to be used in the definition of output layers as it works best for the multi class classification since there are 3 classifications in the target column.

The corrected codes after correcting 2nd and 3rd mistakes are as below.

```
# The neurons in the output layer should be 3 as there are three classes [pos, neg, unsup],
# In the Column label of the dataset,which is the Target.
# Changing the activation function to softmax as it works best for the multi class classification
model.add(layers.Dense(3, activation='softmax'))
```

After compiling and fitting the model lets calculate accuracy and loss.

```
# Compile the model
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['acc'])
# fit the model
history=model.fit(X_train,y_train, epochs=5, verbose=True, validation_data=(X_test,y_test), batch_size=256)

# Evaluating the result on test data and get the loss and accuracy values
[test_loss, test_acc] = model.evaluate(X_test,y_test)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_loss, test_acc))
```

Output:

```
Evaluation result on Test Data : Loss = 1.0825130573272705, accuracy = 0.5069599747657776
```

## Answer 2)

**Purpose:** Using embedded layer in NN model created previously and check the loss and accuracy results.

**Explanation:**

Import the necessary libraries. Here we are also importing the Tokenizer library to create tokens out of sentences of input data.

```
import warnings
warnings.filterwarnings('ignore')
# Imported the necessary libraries and created our environment
# Keras is a high-level library.
# Sequential model is a linear stack of layers.
from keras.models import Sequential
from keras import layers
# Importing Tokenizer to tokenize our data
from keras.preprocessing.text import Tokenizer
# Importing pandas library as we will be doing dataframe manipulation
import pandas as pd
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
# Embedding that will implement the embedding layer
# Flatten to reshape the arrays
from keras.layers import Embedding, Flatten
# pad_sequences that will be used to pad the sentence sequences to the same length
from keras.preprocessing.sequence import pad_sequences
```

Next read the input CSV file into dataframes.

```
#Next is importing 'csv' file into dataframe.
# We are using pd.read_csv() method of Pandas for that. It creates a Dataframe from a csv file.
# Reading the Data
df = pd.read_csv('imdb_master.csv',encoding='latin-1')
# Printing the head of the data
print(df.head())
```

**Output**

```
   Unnamed: 0  type  ... label         file
0           0  test  ...   neg      0_2.txt
1           1  test  ...   neg  10000_4.txt
2           2  test  ...   neg  10001_1.txt
3           3  test  ...   neg  10002_3.txt
4           4  test  ...   neg  10003_3.txt
```

Now we will format the data. We will extract the Sentences and labels from input data frame.

```
# Taking sentences and labels
# .values returns a Numpy array instead of a Pandas Series
df = df[df['label']!='unsup']
sentences = df['review'].values
y = df['label'].values
```

Then we will create tokens out of extracted sentences.

```
# tokenizing data
tokenizer = Tokenizer(num_words=2000)
tokenizer.fit_on_texts(sentences)
```

**Embedding layers:**

Now we will be using Embedding layer for our model. The embedding layers takes the previously calculated integers and maps them to dense vector of the embedding.
We will need to define below parameters for that.

- **input_dim**: the size of the vocabulary
- **output_dim**: the size of the dense vector
- **input_length**: the length of the sequence

Before we can use embedding layers, lets first prepare data for embedding layers.
Calculate the max_review_len for passing it to input_length of embedding layer. Similarly get the vocab_size to be passed as input_dim.

Then get the vocabulary of data and store the sentences. We will use pad_sequences to pad the sentence sequences so as make them of the same length

```
# Preparing the data for embedding layer
max_review_len= max([len(s.split()) for s in sentences])
vocab_size= len(tokenizer.word_index)+1
#getting the vocabulary of data
sentences = tokenizer.texts_to_sequences(sentences)
# padding which indicates whether to add the zeros before or after the sequence.
# # pad_sequences that will be used to pad the sentence sequences to the same length
padded_docs= pad_sequences(sentences,maxlen=max_review_len)
# LabelEncoder to normalize labels.
# Encoding the Target Column
le = preprocessing.LabelEncoder()
y = le.fit_transform(y)
```

Let's add the Embedding layers. We will pass the parameters we set above.
We also need to flatten the embedding layer before passing it to the dense layer so as to reshape it.

```
# In this test_size = 0.25 means 25% od data for testing and 75% of data for training
X_train, X_test, y_train, y_test = train_test_split(padded_docs, y, test_size=0.25, random_state=1000)

# Creating a Model
model1 = Sequential()
# Adding embedding layers in keras
model1.add(Embedding(vocab_size, 50, input_length=max_review_len))
#  flatten the embedding layer before passing it to the dense layer.
model1.add(Flatten())
```

Now let's define the hidden dense layers and output layers and then compile-fit the model.

```
# Adding dense layer of 300 units and activation= relu
model1.add(layers.Dense(300, activation='relu',input_dim=max_review_len))
# The neurons in the output layer should be 3 as there are three classes [pos, neg, unsup],
# In the Column label of the dataset,which is the Target.
# Changing the activation function to softmax as it works best for the multi class classification
model1.add(layers.Dense(3, activation='softmax'))
# Compile the model
model1.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['acc'])
# fit the model
history1=model1.fit(X_train,y_train, epochs=5, verbose=True, validation_data=(X_test,y_test), batch_size=256)
```

As our model is ready lets evaluate it for loss and accuracy.

```
# Evaluating the result on test data and get the loss and accuracy values
[test_loss, test_acc] = model1.evaluate(X_test, y_test)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_loss, test_acc))
```

**Output:**

```
11904/12500 [===========================>..] - ETA: 0s
11968/12500 [===========================>..] - ETA: 0s
12000/12500 [===========================>..] - ETA: 0s
12064/12500 [===========================>..] - ETA: 0s
12128/12500 [===========================>.] - ETA: 0s
12192/12500 [===========================>.] - ETA: 0s
12256/12500 [===========================>.] - ETA: 0s
12320/12500 [===========================>.] - ETA: 0s
12384/12500 [===========================>.] - ETA: 0s
12448/12500 [===========================>.] - ETA: 0s
12500/12500 [==============================] - 15s 1ms/step
Evaluation result on Test Data : Loss = 0.3931850618481636, accuracy = 0.8632799983024597
OMP: Info #250: KMP_AFFINITY: pid 19044 tid 16296 thread 8 bound to OS proc set 0
Actual Value is:  0 Predicted Value is:  [0]
```

| | |
|---|---|
| **Loss for previous model:** | 1.0825130573272705 |
| **Loss for model after adding embedded layer:** | 0.3931850618481636 |
| **Accuracy for previous model:** | 0.5069599747657776 |
| **Accuracy for model after adding embedded layer:** | 0.8632799983024597 |

**Observation:** As we can see for above comparision Loss is reduced while accuracy is increased after adding the Embedding layers.

## Answer 3)

**Purpose:** Applying the model created previously on 20_newsgroup data set.

**Explanation:** As we are using almost similar program as used above only describing the part where changes are necessary for using 20_newsgroup data.

Import the 20_newsgroup library.

```
# Embedding that will implement the embedding layer
# Flatten to reshape the arrays
from keras.layers import Flatten
from keras import layers
# fetch_20newsgroups. Specify a download and cache folder for the datasets
from sklearn.datasets import fetch_20newsgroups
```

Taking 2 categories ['alt.atheism','sci.space']

```
categories = ['alt.atheism','sci.space']
```

Fetch the 20_newsgroup data to train dataset.
20newsgroup is standard text classification dataset which is collection of app. 20,000 newsgroups documents.

```
# 20newsgroup is standard text classification dataset which is collection of app. 20,000 newsgroups documents.
# Create train dataset from 20newsgroup
newsgroups_train =fetch_20newsgroups(subset='train', shuffle=True)
```

Extract data from train dataset.

```
# Extract data from newsgroups train data set.
sentences = newsgroups_train.data
y = newsgroups_train.target
print(np.unique(y))
```

Output:

```
[0 1]
```

Let's format data for our model. Crate the required input for Embedded layer. Then compile, train and fit the model.

```
# tokenizing data
tokenizer = Tokenizer(num_words=2000)
tokenizer.fit_on_texts(sentences)
# Preparing the data for embedding layer
max_review_len= max([len(s.split()) for s in sentences])
vocab_size= len(tokenizer.word_index)+1
#getting the vocabulary of data
sentences = tokenizer.texts_to_sequences(sentences)
# padding which indicates whether to add the zeros before or after the sequence.
# # pad_sequences that will be used to pad the sentence sequences to the same length
padded_docs= pad_sequences(sentences,maxlen=max_review_len)
# LabelEncoder to normalize labels.
# Encoding the Target Column
le = preprocessing.LabelEncoder()
y = le.fit_transform(y)
# Splitting the data into test and train
# In this test_size = 0.25 means 25% od data for testing and 75% of data for training
X_train, X_test, y_train, y_test = train_test_split(padded_docs, y, test_size=0.25, random_state=1000)
```

Changing the number of neuron to 2 as we have only two labels Pos and Neg

```
# create a model
model2 = Sequential()
# Adding embedding layers in keras
model2.add(Embedding(vocab_size, 50, input_length=max_review_len))
#  flatten the embedding layer before passing it to the dense layer.
model2.add(Flatten())
# Adding dense layer of 300 units and activation= relu
model2.add(layers.Dense(300, activation='relu',input_dim=max_review_len))
# Adding dense layer of 20 units and activation= softmax
model2.add(layers.Dense(2, activation='softmax'))
# compile the model
model2.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['acc'])
# fit the model
historynew=model2.fit(X_train,y_train, epochs=5, verbose=True, validation_data=(X_test,y_test), batch_size=256)
```

Finally we will evaluate the model to get loss and accuracy.

```
# Evaluating the result on test data and get the loss and accuracy values
[test_loss, test_acc] = model2.evaluate(X_test, y_test)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_loss, test_acc))
```

Output

```
 32/269 [==>............................] - ETA: 0s
 64/269 [======>.......................] - ETA: 0s
 96/269 [=========>....................] - ETA: 0s
128/269 [=============>................] - ETA: 0s
160/269 [================>.............] - ETA: 0s
192/269 [====================>.........] - ETA: 0s
224/269 [=======================>......] - ETA: 0s
256/269 [===========================>..] - ETA: 0s
269/269 [==============================] - 1s 3ms/step
Evaluation result on Test Data : Loss = 2.256505791582583, accuracy = 0.5947955250740051
```

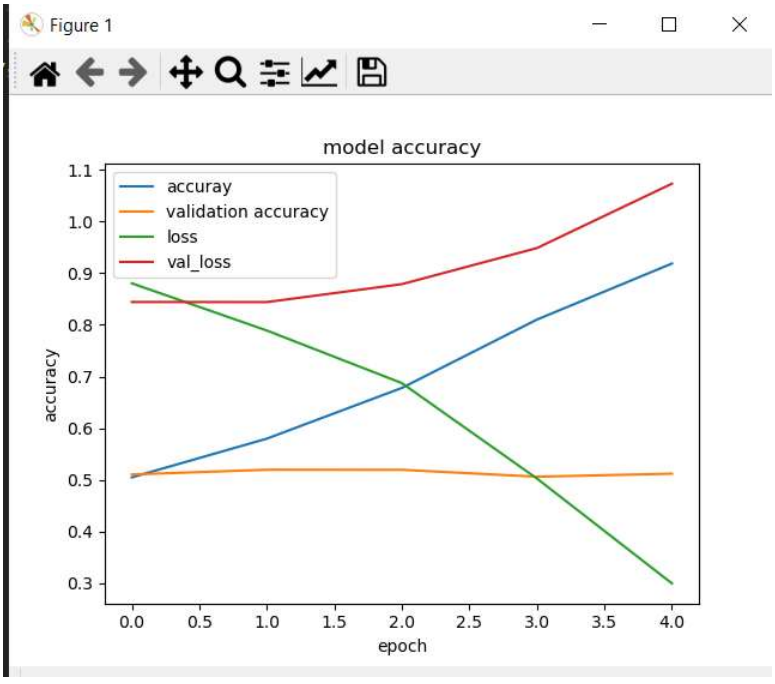
## BONUS ANSWERS 1) : Graphs:

**1)** Plotting the loss and accuracy using history object for program in Answer 1

```
# matplotlib is a plotting library which we have used for our graph
import matplotlib.pyplot as plt
# Plotting history for accuracy
plt.plot(history.history['acc'])
# Plotting history for val_accuracy
plt.plot(history.history['val_acc'])
# Plotting History for loss
plt.plot(history.history['loss'])
# Plotting History for val_loss
plt.plot(history.history['val_loss'])
# Printing Title
plt.title('model accuracy')
# y label as loss
plt.ylabel('accuracy')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'accuray', 'validation accuracy','loss','val_loss'
plt.legend(['accuray', 'validation accuracy','loss','val_loss'], loc='upper left')
# To show the graph
plt.show()
```

**OUTPUT:**

Below graph shows the output of accuracy, val_accuracy, loss and val_loss.

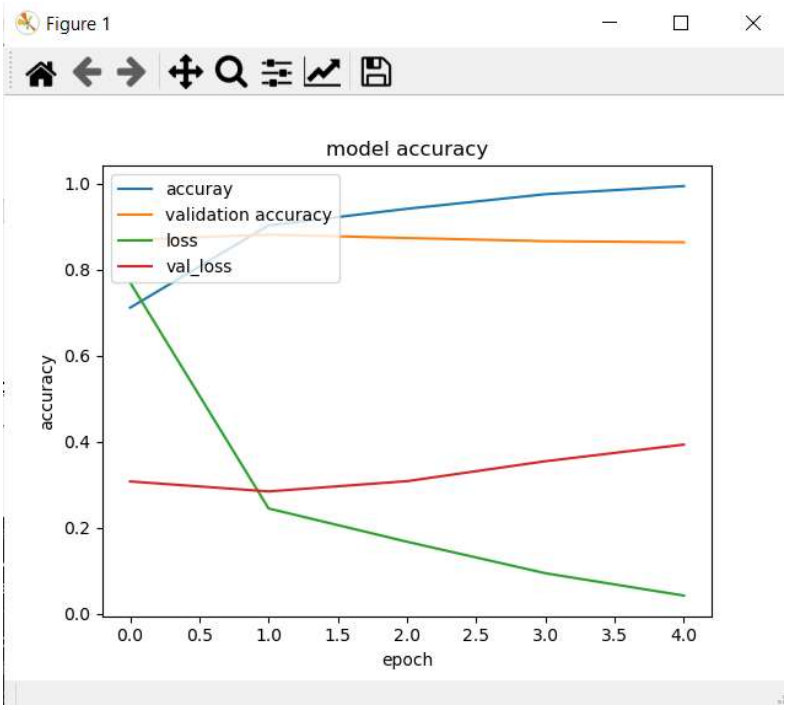


**2)** Plotting the loss and accuracy using history object for program in Answer 2

```
# matplotlib is a plotting library which we have used for our graph
import matplotlib.pyplot as plt
# Plotting history for accuracy
plt.plot(history1.history['acc'])
# Plotting history for val_accuracy
plt.plot(history1.history['val_acc'])
# Plotting History for loss
plt.plot(history1.history['loss'])
# Plotting History for val_loss
plt.plot(history1.history['val_loss'])
# Printing Title
plt.title('model accuracy')
# y label as loss
plt.ylabel('accuracy')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'accuray', 'validation accuracy','loss','val_loss'
plt.legend(['accuray', 'validation accuracy','loss','val_loss'], loc='upper left')
# To show the graph
plt.show()
```

**OUTPUT:**

Below graph shows the output of accuracy, val_accuracy, loss and val_loss.

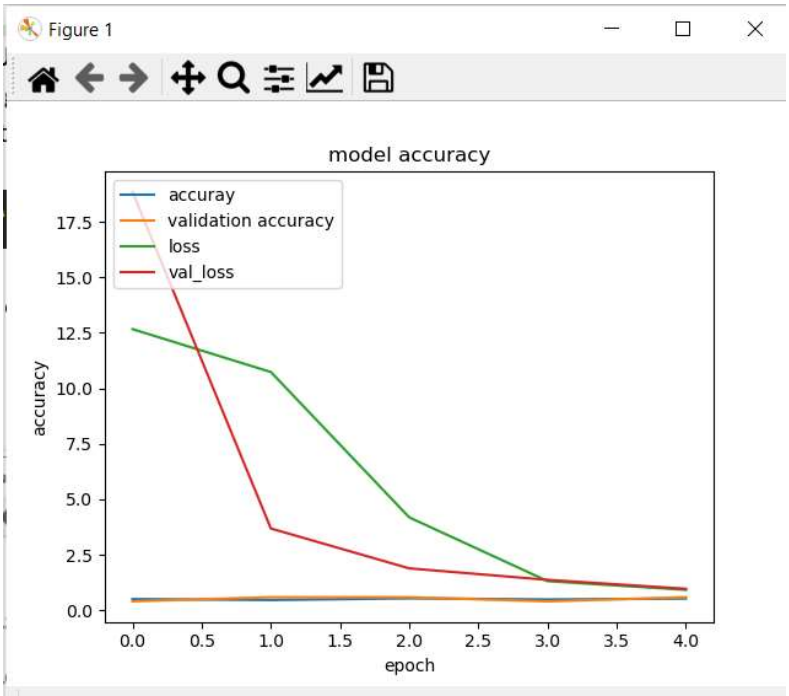


**3)** Plotting the loss and accuracy using history object for program in Answer 3.

```
# matplotlib is a plotting library which we have used for our graph
import matplotlib.pyplot as plt
# Plotting history for accuracy
plt.plot(historynew.history['acc'])
# Plotting history for val_accuracy
plt.plot(historynew.history['val_acc'])
# Plotting History for loss
plt.plot(historynew.history['loss'])
# Plotting History for val_loss
plt.plot(historynew.history['val_loss'])
# Printing Title
plt.title('model accuracy')
# y label as loss
plt.ylabel('accuracy')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'accuray', 'validation accuracy','loss','val_loss'
plt.legend(['accuray', 'validation accuracy','loss','val_loss'], loc='upper left')
# To show the graph
plt.show()
```

**OUTPUT:**

Below graph shows the output of accuracy, val_accuracy, loss and val_loss



**BONUS ANSER 2) Actual vs Prediction value:**

**1) Values for Program in answer 1:**

Predicting over one sample of data and checking the prediction for that.

```
print('Actual Value is: ',y_test[4],'Predicted Value is: ',model.predict_classes(X_test[[4],:]))
```

Below output shows the actual value which is 2 and Predicted value which is 2

```
Actual Value is:  2 Predicted Value is:  [2]
```

**2) Values for Program in answer 2:**

Predicting over one sample of data and checking the prediction for that.

```
print('Actual Value is: ',y_test[5],'Predicted Value is: ',model1.predict_classes(X_test[[5],:]))
```

Below output shows the actual value which is 2 and Predicted value which is 2

```
Actual Value is:  0 Predicted Value is:  [0]
```

**3) Values for Program in answer 3:**

Predicting over one sample of data and checking the prediction for that.

```
print('Actual Value is: ',y_test[2],'Predicted Value is: ',model2.predict_classes(X_test[[2],:]))
```

Below output shows the actual value which is 2 and Predicted value which is 2

```
Actual Value is:  0 Predicted Value is:  [1]
```

As accuracy is low for this model (0.59) output prediction is not always accurate.