Laboratory #2 Tensorflow and CNN

Table of Contents

One of the main reasons in recent year's breakthrough of DNN is the power of new super computers specially with introducing the GPUs. In this lab we will first have a review on a tool that can be used as a GPU tool and then will continue the discussion of Tensorflow that we started in lab1.

Step1. GPU

Google colab is an environement that allows developers to have access to an interactive IDE. There are some advantages of this colab. For example,
You have access to both Python 2 and 3.
You have access to CPU, GPU, and TPU
You can write linux commands in IDE environment
Most of the required libraries are pre-installed
You have access to cloud for storing and retrieving your data

Figure 1- Google colab and connection to other tools

You can either start coding in a notebook or upload from github ipython notebook. Let's start from writing a code from scratch. Here are the steps:
1- Go to https://colab.research.google.com. This is the page that you will see:

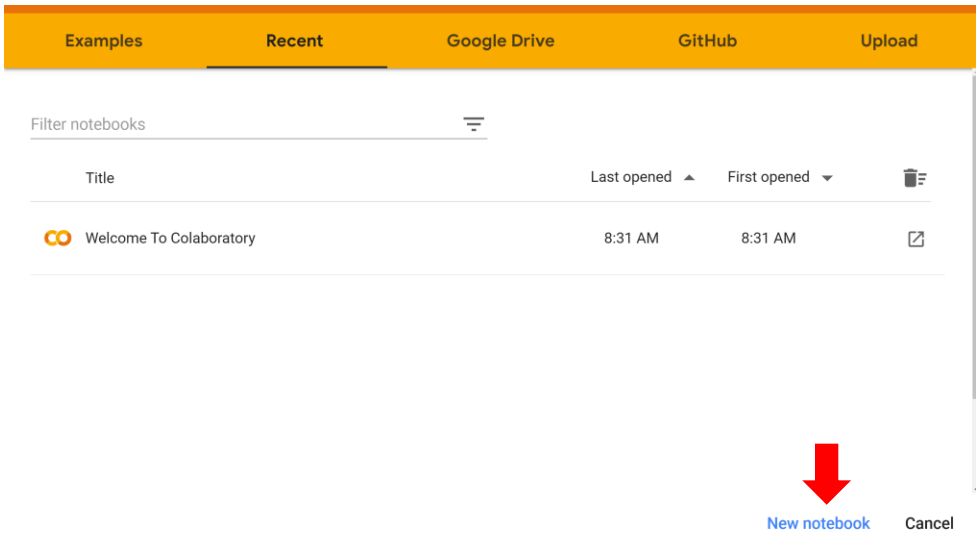


Figure 2- Main page of colab

2- You may go to "GOOGLE DRIVE" tab to store your code on google drive. Click on arrow next to "NEW PYTHON NOTEBOOK 3" to choose the version of the language that you want to use. This is the environment that you will see which is very similar to Python notebook:

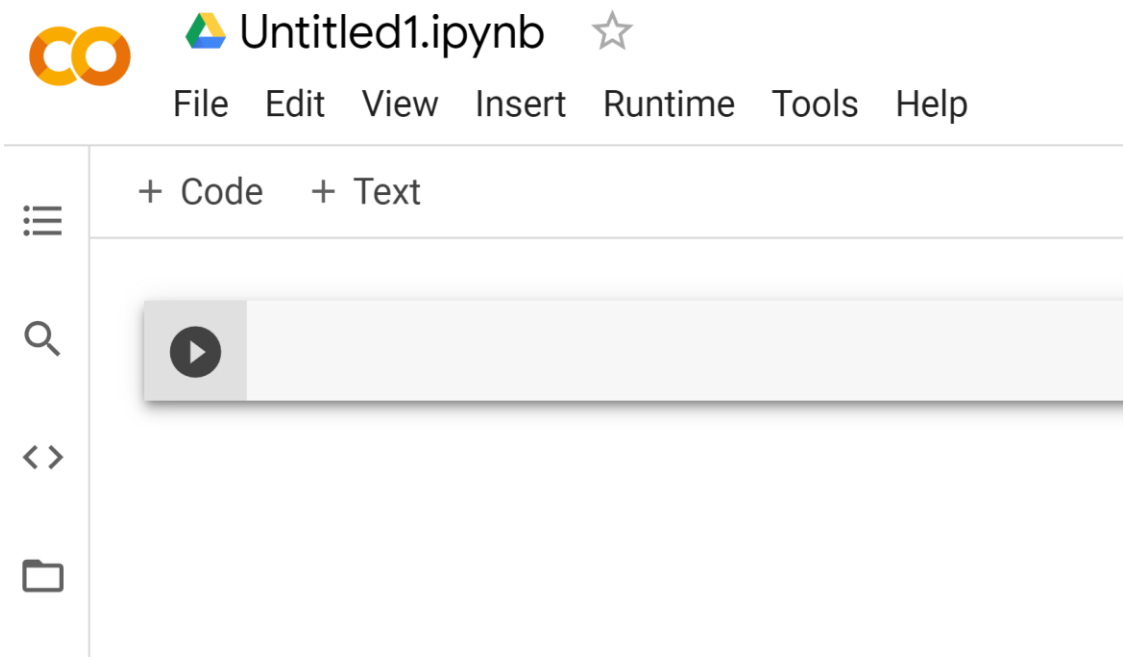


Figure 3- Colab coding environment

Before start actual coding, click on the "Runtime\Change runtime type" to choose between available sources.

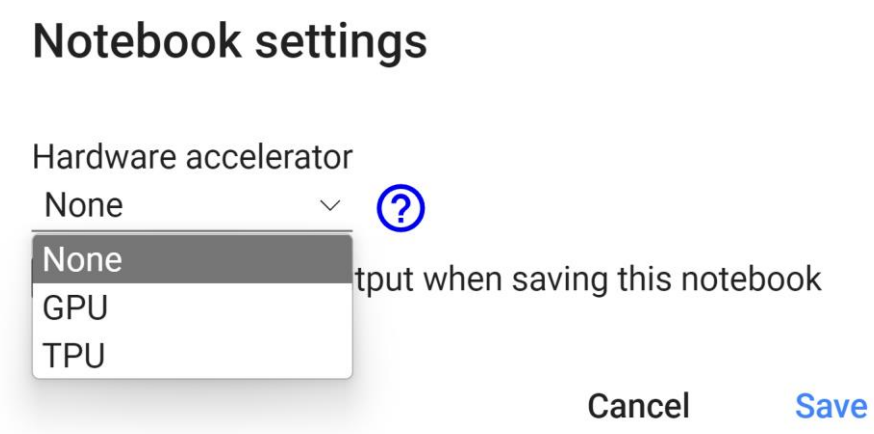


Figure 4- Available resources

Here again you can choose between available types of python. As you see here, you can choose to run your code on GPU rather than CPU to speed up your computations. You can also click on the name on top of the page and changed it to your desired name.

If you like to see the power of GPU resources, you may type following commands in the cell:

```
from tensorflow.python.client import device_lib

print("Show System RAM Memory:\n\n")
!cat /proc/meminfo | egrep "MemTotal*"


print("\n\nShow Devices:\n\n"+str(device_lib.list_local_devices()))
```

To run the command, click on the arrow key on the left hand side of the cell. This will show you a page like this:

```
Show System RAM Memory:


MemTotal:       13302920 kB


Show Devices:

[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 10900234812275313976
xla_global_id: -1
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 14465892352
locality {
  bus_id: 1
  links {
  }
}
incarnation: 1781117250928914076
physical_device_desc: "device: 0, name: Tesla T4, pci bus id: 0000:00:04.0, compute capability: 7.5"
xla_global_id: 416903419
]
```

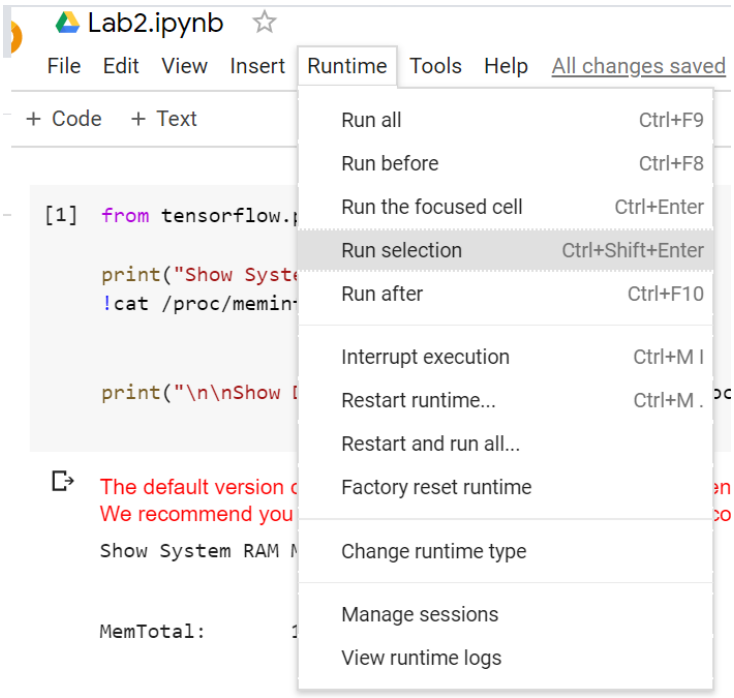
It looks we have a Tesla T4 GPU. With 16GB of RAM which is good for our experiments.
We need Tensorflow ver. 2.7 for this lab so first check the version of Tensorflow using:

```
import tensorflow as tf
tf.__version__
```

If you see any other versions, you can upgrade it using:

```
!pip install q tensorflow-gpu==2.7.0
```

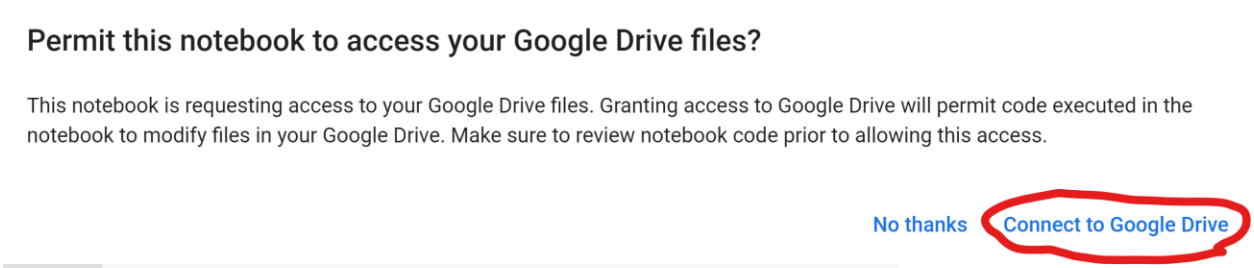
And restart the code from here:



We can share a google drive into Colab environment using:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

This will show you a permit notification, click on "connect to Google Drive" and login to your google account:



Now we can load codes our datasets from the google drive.
There are other resources like Amazon (AWS), Microsoft (Azure) or Floydhub but unfortunately, they are not free.
For start we want to compare running a code on your machine and on your cloud system.

Q1- A Keras code is provided for running hand written recognition on both GPU and CPU. Run the code on colab and your own machine and compare the results.

You may use following code to see the speed of the code:
For running the code on your own machine:

```
import time
start = time.time()
%run Address/to/file/mnist_cnn
end = time.time()
print(end - start)
```

Run time on my computer: 2469 seconds
For Colab:

```
import time
start = time.time()
!python3 "Adress/to/drive/mnist_cnn.py"
end = time.time()
print(end - start)
```

Run time on GPU: 75 seconds

Step2. Implement handwritten recognition in Tensorflow using CNN

In this section, we want to implement the code which is provided in the attached file in Colab and analyze each section of the code.
We start with loading the dataset:

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras import backend as K

batch_size = 128
num_classes = 10
epochs = 12

# input image dimensions
img_rows, img_cols = 28, 28

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

This general code will convert the add the number of channels to the dataset. Remember that it is a key factor that should be attached to dataset in order to analyze the data using CNN.

```
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
```

```
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
```

Continue the pre-processing with:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

Now is the time to design the model.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),activation='relu',
input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Q2- Explain the way that this model is designed. Talk about all the layers and their functionality.

Now to evaluate the model, we can write:

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_split=0.2)

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

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Epoch 1/12
375/375 [==============================] - 8s 17ms/step - loss: 0.2732 - accuracy: 0.9165 - val_loss: 0.0677 - val_accuracy: 0.9797
Epoch 2/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0972 - accuracy: 0.9712 - val_loss: 0.0494 - val_accuracy: 0.9859
Epoch 3/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0716 - accuracy: 0.9779 - val_loss: 0.0421 - val_accuracy: 0.9871
Epoch 4/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0607 - accuracy: 0.9817 - val_loss: 0.0432 - val_accuracy: 0.9874
Epoch 5/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0487 - accuracy: 0.9855 - val_loss: 0.0390 - val_accuracy: 0.9890
Epoch 6/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0417 - accuracy: 0.9872 - val_loss: 0.0438 - val_accuracy: 0.9883
Epoch 7/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0381 - accuracy: 0.9874 - val_loss: 0.0413 - val_accuracy: 0.9889
Epoch 8/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0338 - accuracy: 0.9894 - val_loss: 0.0385 - val_accuracy: 0.9893
Epoch 9/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0316 - accuracy: 0.9896 - val_loss: 0.0355 - val_accuracy: 0.9910
Epoch 10/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0281 - accuracy: 0.9904 - val_loss: 0.0415 - val_accuracy: 0.9898
Epoch 11/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0267 - accuracy: 0.9910 - val_loss: 0.0376 - val_accuracy: 0.9908
Epoch 12/12
375/375 [==============================] - 6s 15ms/step - loss: 0.0253 - accuracy: 0.9911 - val_loss: 0.0412 - val_accuracy: 0.9902
Test loss: 0.031212283298373222
Test accuracy: 0.9921000003814697
```

Q3- Design the learning curve and talk about what you see.

Step3. Text mining using CNN

The majority of this part of lab is coming from Realpython website.

3.1. Pre-processing:

For this part of the lab we want to see one of the other applications of CNN which is text mining. The dataset is downloaded from the Sentiment Labelled Sentences Data Set from [the UCI Machine Learning Repository](). It is also uploaded on Canvas. This data includes labeled overviews from Amazon, Yelp and IMDB. We will work only part of the dataset which is Amazon reviews. Each review is marked as 0 for negative comment or 1 for positive sentiment. Run following code to see one of the results:

```
import pandas as pd

df = pd.read_csv('gdrive/My Drive/data/amazon_cells_labelled.txt', names=[
'sentence', 'label'], sep='\t')
```

We can print one of the dataset values to see inside the dataframe.

```
print(df.iloc[0])
```

The result will be:

```
sentence    So there is no way for me to plug it in here i...
label                                                       0
source                                                 amazon
Name: 0, dtype: object
```

The way that the dataset is labeled is taught in Sentimental analysis course. The collection of texts (corpus) is analyzed and the frequency of particular word is counted. Then, it is compared to a dictionary to see if it is positive or negative. *Feature vector* is a vector that contains all the vocabulary words plus their count. Let's see how these vectors are generated. Let's think of the sentences that we have as following vector named sentences:

```
sentences = ['John likes ice cream', 'John hates chocolate.']
```

CountVectorizer from *scikit-learn* library can take these sentences and make this feature vector. This is how it works:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(min_df=0, lowercase=False)
vectorizer.fit(sentences)
vectorizer.vocabulary_
```

```
{'John': 0, 'chocolate': 1, 'cream': 2, 'hates': 3, 'ice': 4, 'likes': 5}
```

This vocabulary serves also as an index of each word. Now, you can take each sentence and get the word occurrences of the words based on the previous vocabulary. The vocabulary consists of all five words in our sentences, each representing one word in the vocabulary. When you take the previous two sentences and transform them with the CountVectorizer you will get a vector representing the count of each word of the sentence:

```
vectorizer.transform(sentences).toarray()
```

```
array([[1, 0, 1, 0, 1, 1],
       [1, 1, 0, 1, 0, 0]], dtype=int64)
```

Now, you can see the resulting feature vectors for each sentence based on the previous vocabulary. For example, if you take a look at the first item, you can see that both vectors have a 1 there. This means that both sentences have one occurrence of John, which is in the first place in the vocabulary. This is called Bag of Words (BOW) model.

Let's get back to our own problem and load "Amazon" dataset for more analysis.

```
from sklearn.model_selection import train_test_split


sentences = df['sentence'].values
y = df['label'].values

sentences_train, sentences_test, y_train, y_test =
train_test_split(sentences, y, test_size=0.25, random_state=1000)
```

We can again use BOW strategy to create vectorized sentences.

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
vectorizer = CountVectorizer()
vectorizer.fit(sentences_train)

X_train = vectorizer.transform(sentences_train)
X_test  = vectorizer.transform(sentences_test)
X_train
```

```
<750x1546 sparse matrix of type '<class 'numpy.int64'>'
        with 6817 stored elements in Compressed Sparse Row format>
```

It shows 750 samples which are the number of training samples. Each sample has 1546 dimensions which is the size of the vocabulary.
Just as a side note, we really don't need to always use fancy algorithms. For example here even using a logistic regression model, gives us a reasonable result:

```
from sklearn.linear_model import LogisticRegression

classifier = LogisticRegression()
classifier.fit(X_train, y_train)
score = classifier.score(X_test, y_test)

print("Accuracy:", score)
```

```
Accuracy: 0.796
```

Now, we can implement a normal DNN. Before training we need to convert the sentences into arrays.

```
X_train = vectorizer.transform(sentences_train).toarray()
X_test = vectorizer.transform(sentences_test).toarray()

from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

input_dim = X_train.shape[1]  # Number of features

model = Sequential()
model.add(layers.Dense(10, input_dim=input_dim, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

hist = model.fit(X_train, y_train, epochs=100, validation_split=0.2 ,
batch_size=10)
loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Test Accuracy: ",accuracy*100)
```

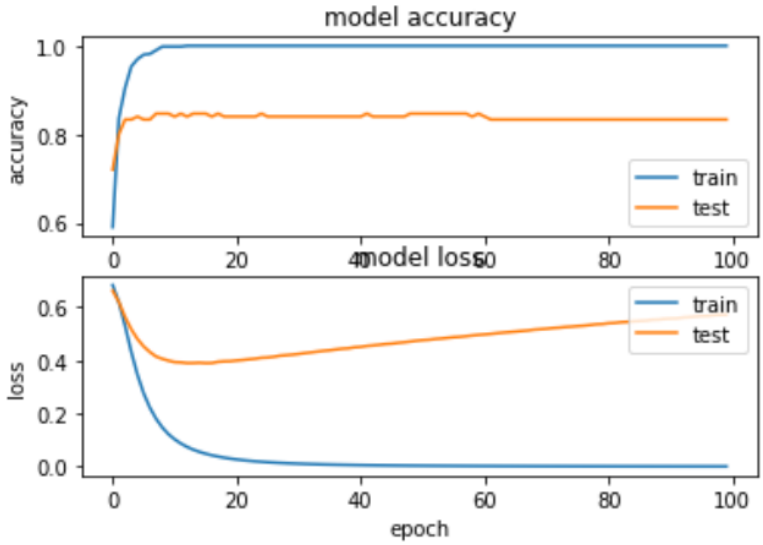
```
Epoch 99/100
60/60 [==============================] - 0s 3ms/step - loss: 3.4720e-04 - accuracy: 1.0000 - val_loss: 0.5728
Epoch 100/100
60/60 [==============================] - 0s 3ms/step - loss: 3.3433e-04 - accuracy: 1.0000 - val_loss: 0.5746
Test Accuracy:  77.60000228881836
```

Note- In case the code generated an error about type of array, convert the variables into numpy array format using:

X_train = vectorizer.transform(sentences_train).toarray()
X_test = vectorizer.transform(sentences_test).toarray()

Let's draw the learning curves.

And here is what you will see:



Q4- Explain these graphs. If you see any issue, suggest a solution to resolve it. Make the model by creating 3 hidden layers (first one 200 nodes, second one 100 nodes and last one 50 nodes and after each step, add dropout of 0.2 and report the accuracy.
If you don't see a huge improvement, don't worry we are not done with the model yet.

3.2. Embedded word:

Text is considered a form of sequence data similar to time series data that you would have in weather data or financial data. In the previous BOW model, you have seen how to represent a whole sequence of words as a single feature vector. Now you will see how to represent each word as vectors. There are various ways to vectorize text, such as:

- Words represented by each word as a vector
- Characters represented by each character as a vector
- N-grams of words/characters represented as a vector (N-grams are overlapping groups of multiple succeeding words/characters in the text)

If you want to learn more about the algorithm, you may read this website:

https://medium.com/@krishnakalyan3/a-gentle-introduction-to-embedding-567d8738372b

Data pre-processing steps:

```
from keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(sentences_train)

X_train = tokenizer.texts_to_sequences(sentences_train)
X_test = tokenizer.texts_to_sequences(sentences_test)

vocab_size = len(tokenizer.word_index) + 1  # Adding 1 because of reserved 0
index

print(sentences_train[3])
print(X_train[3])
```

```
This is the phone to get for 2005.... I just bought my S710a and all I can say is WOW!
[7, 5, 1, 9, 8, 92, 11, 676, 2, 59, 101, 10, 677, 3, 32, 2, 71, 225, 5, 449]
```

The indexing is ordered after the most common words in the text, which you can see by the word this having the index 1. It is important to note that the index 0 is reserved and is not assigned to any word. This zero index is used for padding, which I'll introduce in a moment.

Unknown words (words that are not in the vocabulary) are denoted in Keras with word_count + 1 since they can also hold some information. You can see the index of each word by taking a look at the word_index dictionary of the Tokenizer object:

```
for word in ['the', 'all', 'happy']:
     print('{}: {}'.format(word, tokenizer.word_index[word]))
```

```
the: 1
all: 32
happy: 86
```

With CountVectorizer, we had stacked vectors of word counts, and each vector was the same length (the size of the total corpus vocabulary). With Tokenizer, the resulting vectors equal the length of each text, and the numbers don't denote counts, but rather correspond to the word values from the dictionary tokenizer.word_index.
We can add a parameter to identify how long each sequence should be.

```
from keras.preprocessing.sequence import pad_sequences

maxlen = 100

# Pad variables with zeros
X_train = pad_sequences(X_train, padding='post', maxlen=maxlen)
X_test = pad_sequences(X_test, padding='post', maxlen=maxlen)
print(X_train[0, :])
```

```
[  7  24   5  16   4 137 148   6 223 315   2  71 224   8   1 673 111 444
  18 316  11 445   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0]
```

3.3. Model training:

We can now start training the model:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

embedding_dim = 50

model = Sequential()
model.add(layers.Embedding(input_dim=vocab_size,
                           output_dim=embedding_dim,
                           input_length=maxlen))
model.add(layers.GlobalMaxPool1D())
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model.summary()
```

This shows a summary of model.

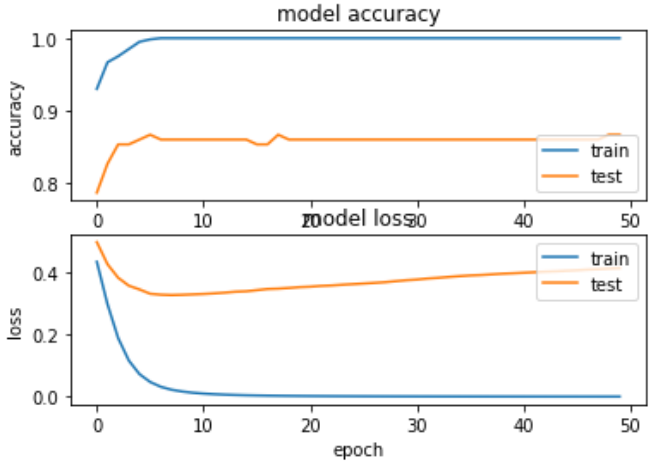
```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 100, 50)           78700
_________________________________________________________________
global_max_pooling1d (Global (None, 50)                0
_________________________________________________________________
dense_10 (Dense)             (None, 10)                510
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 11
=================================================================
Total params: 79,221
Trainable params: 79,221
Non-trainable params: 0
```

And train and evaluate the model with:

```
hist = model.fit(X_train, y_train,
                 epochs=50,
                 validation_split=0.2,
                 batch_size=10)
loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
```

```
print("Accuracy: ",accuracy)
```

Accuracy = 81%



Q5- How do you interpret these results?
Q6- What is your recommendation to improve the accuracy? Implement your idea.

For final submission, you may use following code to convert the ipynb to html:

```
%shell jupyter nbconvert --to html '//content/gdrive/My Drive/Colab Notebooks/filename.ipynb'
```