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

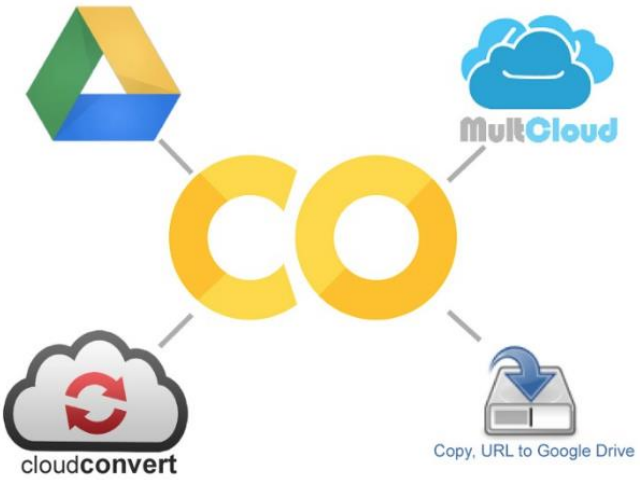


Figure 1- Google colab and connection to other tools

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:

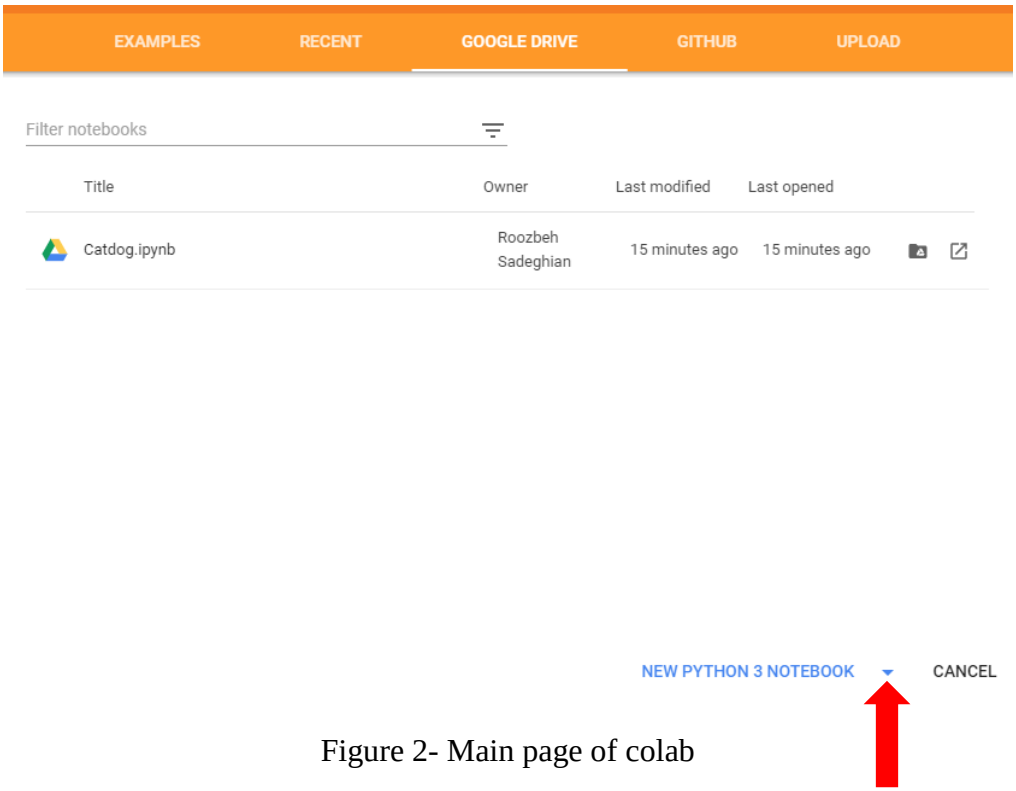


Figure 2- Main page of colab

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:

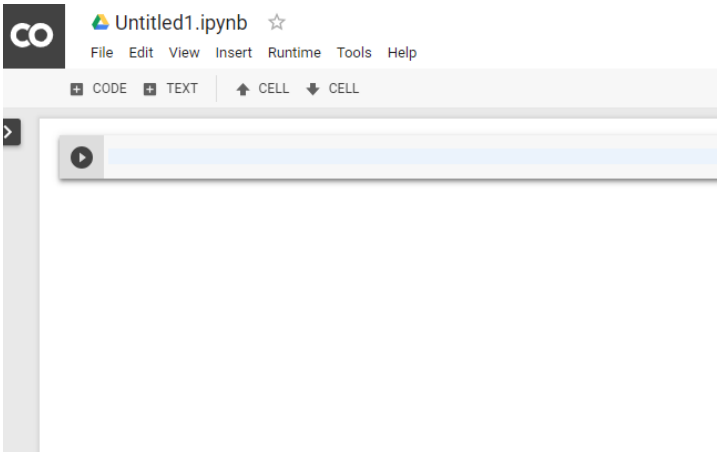


Figure 3- Colab coding environment

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

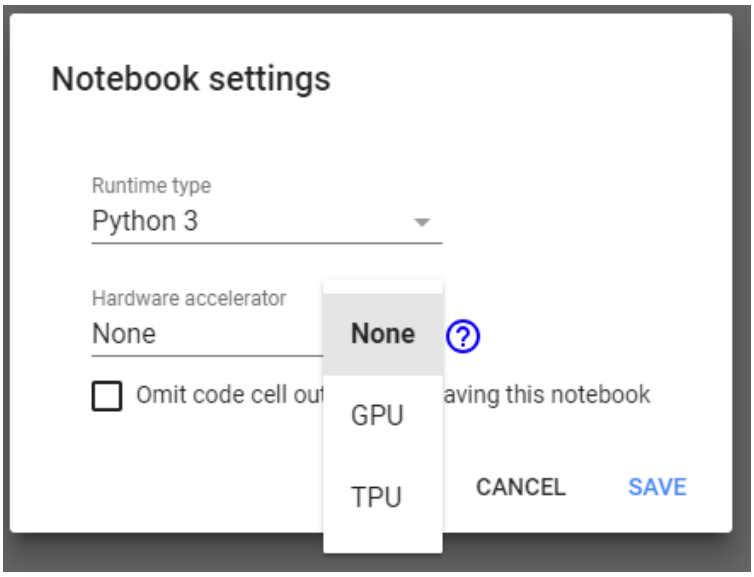


Figure 4- Available resources

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:

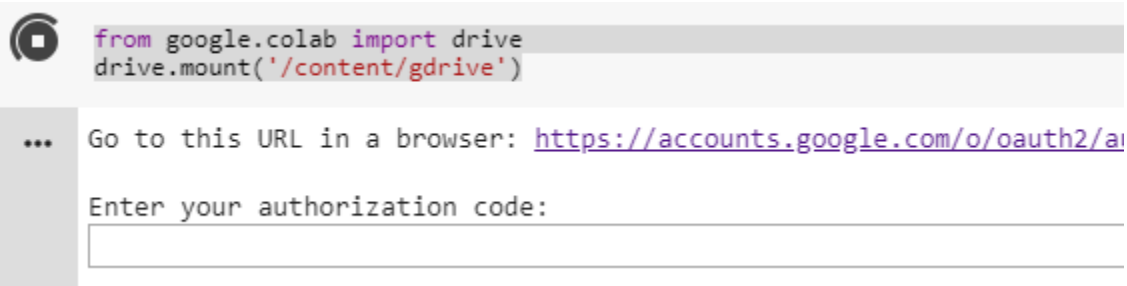
```
Show System RAM Memory:


MemTotal:       13335212 kB


Show Devices:

[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 17586487045871342804
, name: "/device:XLA_CPU:0"
device_type: "XLA_CPU"
memory_limit: 17179869184
locality {
}
incarnation: 735060478082093336
physical_device_desc: "device: XLA_CPU device"
, name: "/device:XLA_GPU:0"
device_type: "XLA_GPU"
memory_limit: 17179869184
locality {
}
incarnation: 5016494282645131613
physical_device_desc: "device: XLA_GPU device"
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 10962786714
locality {
  bus_id: 1
  links {
  }
}
incarnation: 5433205508330369511
physical_device_desc: "device: 0, name: Tesla K80, pci bus id: 0000:00:04.0, compute capability: 3.7"
]
```

It looks we have a Tesla K80 GPU. With 13GB of RAM which is good for our experiments.

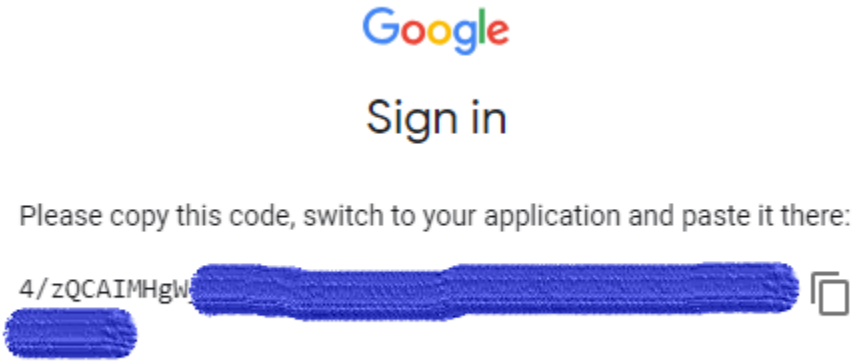
We can share a google drive into Colab environment using:

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

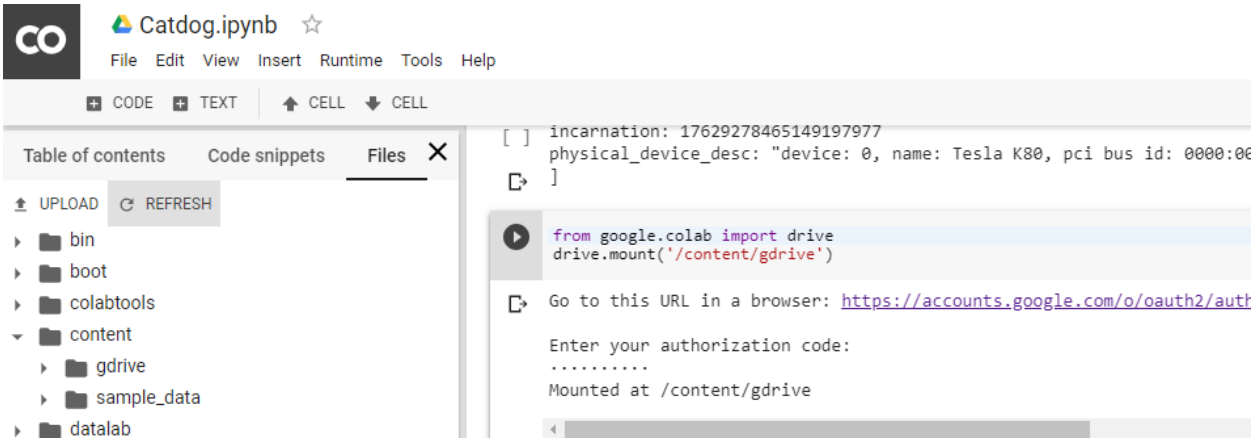
This will show you a link, click on it and login to your google account:



Copy the provided link back into authorization code of Colab.



This will copy your address under “content/gdraive”. You can have access to this file from left hand side of the environment.



Now, if you want to load dataset as we did in class you may use this command:

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale = 1./255, shear_range = 0.2,
zoom_range = 0.2, horizontal_flip = True)

training_set = train_datagen.flow_from_directory('/content/gdrive/My
Drive/Colab Notebooks/train/', target_size = (50, 50),
batch_size = 32, class_mode = 'binary')
```

This is the only part that looks tricky.

Q1- Now run the code provided in class (Keras on cat dog classification) and see the result. Do you think speed is faster?

There are other resources like Amazon (AWS), Microsoft (Azure) or Floydhub but unfortunately, they are not free.

Q2- 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 CPU:

```
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: 116 seconds

Step2. Implement handwritten recognition in Tensorflow using CNN

In this section, we want to rebuild the model that we designed using Keras. This time we will use Tensorflow.

Stat with loading the MNIST into python environment:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot = True)
batch_size = 128
import matplotlib.pyplot as plt
import random as ran
import numpy as np
# Functions that can define the size of train and test sets
train_X = mnist.train.images.reshape(-1, 28, 28, 1)
test_X = mnist.test.images.reshape(-1,28,28,1)

train_y = mnist.train.labels
test_y = mnist.test.labels
```

Now define the placeholders:

```
x = tf.placeholder("float", [None, 28,28,1])
y = tf.placeholder("float", [None, 10])
```

We can define the convolution and max pooling layers as functions. This can help us to easily modify the parameters easier.

```
def conv2d(x, W, b, strides=1):
    # Conv2D wrapper, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)

def maxpool2d(x, k=2):
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k,
1],padding='SAME')
```

We want to use following structure for the CNN. We want to have 3 convolutional layer in which each layer has a maxpooling layer with size of 2x2.

- First Conv layer has 32-3x3 filters
- Second Conv layer has 64-3x3 filters
- Third Conv layer has 128-3x3 filters

The fully connected layer has one layer of 128 nodes and the last layer which is the output one.

This structure can be implemented as following:

```
weights = {

    'wc1': tf.get_variable('W0', shape=(3,3,1,32),
initializer=tf.contrib.layers.xavier_initializer()),
    'wc2': tf.get_variable('W1', shape=(3,3,32,64),
initializer=tf.contrib.layers.xavier_initializer()),
    'wc3': tf.get_variable('W2', shape=(3,3,64,128),
initializer=tf.contrib.layers.xavier_initializer()),
    'wd1': tf.get_variable('W3', shape=(4*4*128,128),
initializer=tf.contrib.layers.xavier_initializer()),
    'out': tf.get_variable('W6', shape=(128,10),
initializer=tf.contrib.layers.xavier_initializer()),
}
biases = {
    'bc1': tf.get_variable('B0', shape=(32),
initializer=tf.contrib.layers.xavier_initializer()),
    'bc2': tf.get_variable('B1', shape=(64),
initializer=tf.contrib.layers.xavier_initializer()),
    'bc3': tf.get_variable('B2', shape=(128),
initializer=tf.contrib.layers.xavier_initializer()),
    'bd1': tf.get_variable('B3', shape=(128),
initializer=tf.contrib.layers.xavier_initializer()),
    'out': tf.get_variable('B4', shape=(10),
initializer=tf.contrib.layers.xavier_initializer()),
}
```

Conv_net() function receives all the above mentioned functions and generates the network.

```
def conv_net(x, weights, biases):

    # here we call the conv2d function we had defined above and pass the
input image x, weights wc1 and bias bc1.
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    # Max Pooling (down-sampling), this chooses the max value from a 2*2
matrix window and outputs a 14*14 matrix.
    conv1 = maxpool2d(conv1, k=2)

    # Convolution Layer
    # here we call the conv2d function we had defined above and pass the
input image x, weights wc2 and bias bc2.
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    # Max Pooling (down-sampling), this chooses the max value from a 2*2
matrix window and outputs a 7*7 matrix.
    conv2 = maxpool2d(conv2, k=2)

    conv3 = conv2d(conv2, weights['wc3'], biases['bc3'])
```

```
    # Max Pooling (down-sampling), this chooses the max value from a 2*2
matrix window and outputs a 4*4.
    conv3 = maxpool2d(conv3, k=2)


    # Fully connected layer
    # Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(conv3, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    # Output, class prediction
    # finally we multiply the fully connected layer with the weights and add
a bias term.
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
```

The optimization algorithm can be designed as below:

```
learning_rate = 0.001

pred = conv_net(x, weights, biases)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred,
labels=y))

optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

#Here you check whether the index of the maximum value of the predicted image
is equal to the actual labelled image. and both will be a column vector.
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))

#calculate accuracy across all the given images and average them out.
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Finally, we can run the following code to start the algorithm:

```
init = tf.global_variables_initializer()

epochs = 100

with tf.Session() as sess:
    sess.run(init)
    train_loss = []
    test_loss = []
    train_accuracy = []
    test_accuracy = []
    summary_writer = tf.summary.FileWriter('./Output', sess.graph)
```

```
    for i in range(epochs):
        for batch in range(len(train_X)//batch_size):
            batch_x =
train_X[batch*batch_size:min((batch+1)*batch_size,len(train_X))]
            batch_y =
train_y[batch*batch_size:min((batch+1)*batch_size,len(train_y))]
            # Run optimization op (backprop).
                # Calculate batch loss and accuracy
            opt = sess.run(optimizer, feed_dict={x: batch_x,
                                                 y: batch_y})
            loss, acc = sess.run([cost, accuracy], feed_dict={x: batch_x,
                                                              y: batch_y})
        print("Iter " + str(i) + ", Loss= " + \
                      "{:.6f}".format(loss) + ", Training Accuracy= " + \
                      "{:.5f}".format(acc))
        print("Optimization Finished!")

        # Calculate accuracy for all 10000 mnist test images
        test_acc,valid_loss = sess.run([accuracy,cost], feed_dict={x:
test_X,y : test_y})
        train_loss.append(loss)
        test_loss.append(valid_loss)
        train_accuracy.append(acc)
        test_accuracy.append(test_acc)
        print("Testing Accuracy:","{:.5f}".format(test_acc))
    summary_writer.close()
```

This is part of the result that you will see:

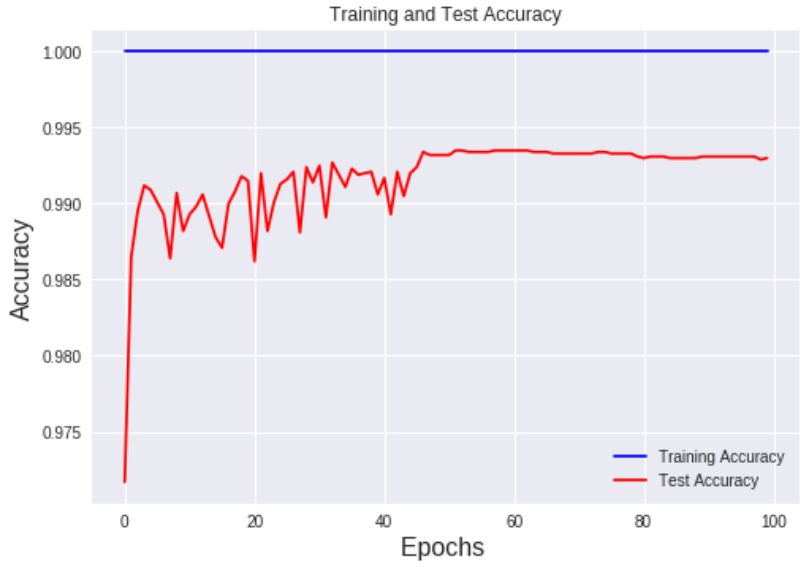
```
Iter 25, Loss= 0.000020, Training Accuracy= 1.00000
Optimization Finished!
Testing Accuracy: 0.98840
Iter 26, Loss= 0.000054, Training Accuracy= 1.00000
Optimization Finished!
Testing Accuracy: 0.98890
Iter 27, Loss= 0.000305, Training Accuracy= 1.00000
Optimization Finished!
Testing Accuracy: 0.99140
Iter 28, Loss= 0.000024, Training Accuracy= 1.00000
Optimization Finished!
Testing Accuracy: 0.99120
Iter 29, Loss= 0.000025, Training Accuracy= 1.00000
Optimization Finished!
Testing Accuracy: 0.99150
```

You can see the loss and accuracy graphs using following commands:

```
import matplotlib.pyplot as plt
```

```
plt.plot(range(len(train_loss)), train_loss, 'b', label='Training loss')
plt.plot(range(len(train_loss)), test_loss, 'r', label='Test loss')
plt.title('Training and Test loss')
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.legend()
plt.figure()
plt.show()
```


```
plt.plot(range(len(train_loss)), train_accuracy, 'b', label='Training
Accuracy')
plt.plot(range(len(train_loss)), test_accuracy, 'r', label='Test Accuracy')
plt.title('Training and Test Accuracy')
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.legend()
plt.figure()
plt.show()
```



Q3- Do you think dropout can help the model?

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 Moodle. This data includes labeled overviews from Amazon, Yelp and IMDB. 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

filepath_dict = {'yelp':   'data/sentiment labelled
sentences/yelp_labelled.txt',
                 'amazon': 'data/sentiment labelled
sentences/amazon_cells_labelled.txt',
                 'imdb':   'data/sentiment labelled
sentences/imdb_labelled.txt'}

df_list = []
for source, filepath in filepath_dict.items():
    df = pd.read_csv(filepath, names=['sentence', 'label'], sep='\t')
    df['source'] = source  # Add another column filled with the source name
    df_list.append(df)

df = pd.concat(df_list)
```

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

The result will be:

```
sentence    Wow... Loved this place.
label                              1
source                          yelp
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 "yelp" dataset for more analysis.

```
from sklearn.model_selection import train_test_split

df_yelp = df[df['source'] == 'yelp']

sentences = df_yelp['sentence'].values
y = df_yelp['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
```

```
<750x1714 sparse matrix of type '<class 'numpy.int64'>'
        with 7368 stored elements in Compressed Sparse Row format>
```

It shows 750 samples which are the number of training samples. Each sample has 1714 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:

```
from keras.models import Sequential
from 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'])

history = model.fit(X_train, y_train, epochs=100, validation_data=(X_test,
y_test), batch_size=10)
loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))
loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy:  {:.4f}".format(accuracy))
```

```
Epoch 98/100
750/750 [==============================] - 0s 293us/step - loss: 3.2084e-04 - acc: 1.0000 - val_loss: 0.7736 - val_acc: 0.784
0
Epoch 99/100
750/750 [==============================] - 0s 283us/step - loss: 3.0789e-04 - acc: 1.0000 - val_loss: 0.7801 - val_acc: 0.784
0
Epoch 100/100
750/750 [==============================] - 0s 288us/step - loss: 2.9406e-04 - acc: 1.0000 - val_loss: 0.7846 - val_acc: 0.784
0
Training Accuracy: 1.0000
Testing Accuracy:  0.7840
```

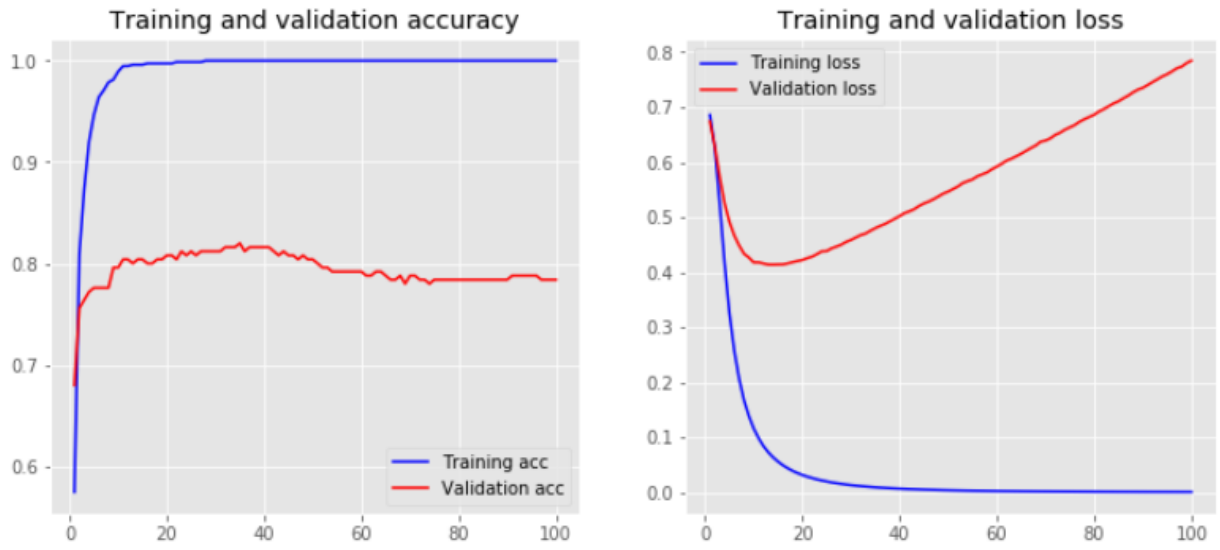
Let's draw the learning curves. We can use following function to draw learning curves.

```
import matplotlib.pyplot as plt
plt.style.use('ggplot')

def plot_history(history):
    acc = history.history['acc']
    val_acc = history.history['val_acc']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    x = range(1, len(acc) + 1)

    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(x, acc, 'b', label='Training acc')
    plt.plot(x, val_acc, 'r', label='Validation acc')
    plt.title('Training and validation accuracy')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(x, loss, 'b', label='Training loss')
    plt.plot(x, val_loss, 'r', label='Validation loss')
    plt.title('Training and validation loss')
    plt.legend()
    plt.show()
```

If you want to see the graphs, you can call the function using:

```
plot_history(history)
```

And here is what you will see:



Q4- Explain these graphs. If you see any issue, suggest a solution to resolve it.

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[2])
print(X_train[2])
```

```
Of all the dishes, the salmon was the best, but all were great.
[11, 43, 1, 171, 1, 283, 3, 1, 47, 26, 43, 24, 22]
```

The indexing is ordered after the most common words in the text, which you can see by the word the 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', 'sad']:
    print('{}: {}'.format(word, tokenizer.word_index[word]))
```

```
the: 1
all: 43
happy: 320
sad: 450
```

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, :])
```

```
[  1  10   3 282 739  25   8 208  30  64 459 230  13   1 124   5 231   8
  58   5  67   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   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:

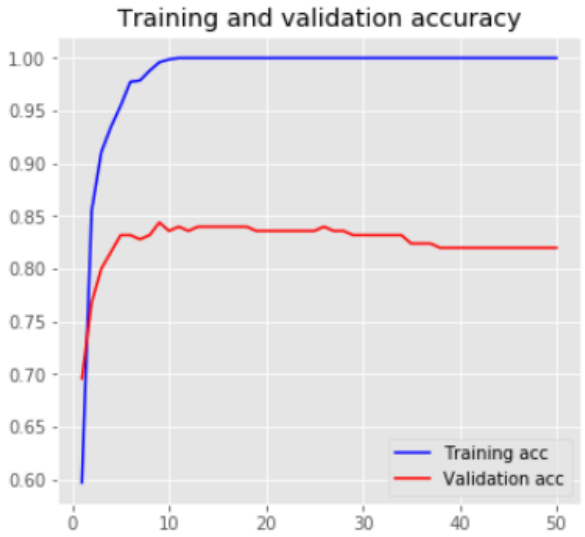
We can now start training the model:

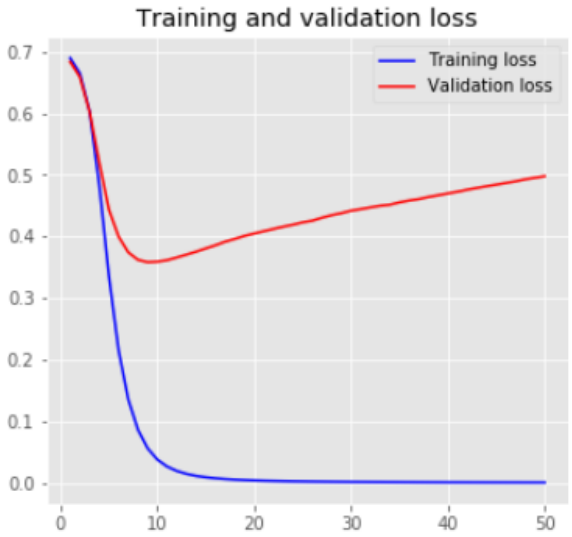
```
from keras.models import Sequential
from keras import layers

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()
```

And evaluate the model with:

```
history = model.fit(X_train, y_train,
                    epochs=50,
                    verbose=False,
                    validation_data=(X_test, y_test),
                    batch_size=10)
loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))
loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy:  {:.4f}".format(accuracy))
plot_history(history)
```

```
Training Accuracy: 1.0000
Testing Accuracy:  0.8200
```





Q5- How do you interpret these results?

Q6- What is your recommendation to improve the accuracy?