Tensorflow

Seccia Ruggiero
PhD candidate in Operations Research
ruggiero.seccia@uniroma1.it

Tensorflow is an open source library for numerical computations, developed by **Google Brain Team**.
Why the name Tensor-flow? because it works considering the flow of tensors.

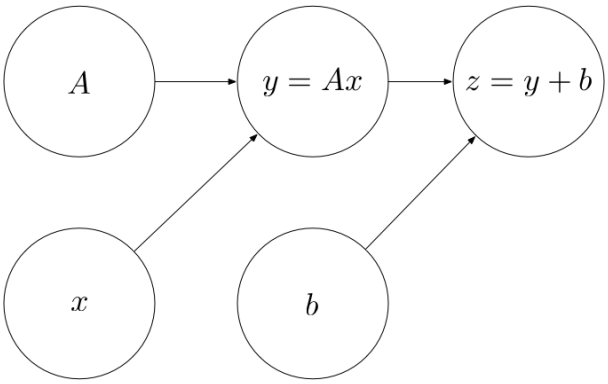
What is a tensor? Nothing more than a multidimensional array

- 1D: a vector
- 2D: a matrix
- 3D: a tensor (also more than 3D)

Tensorflow is based on the implementation of a *computational Graph*

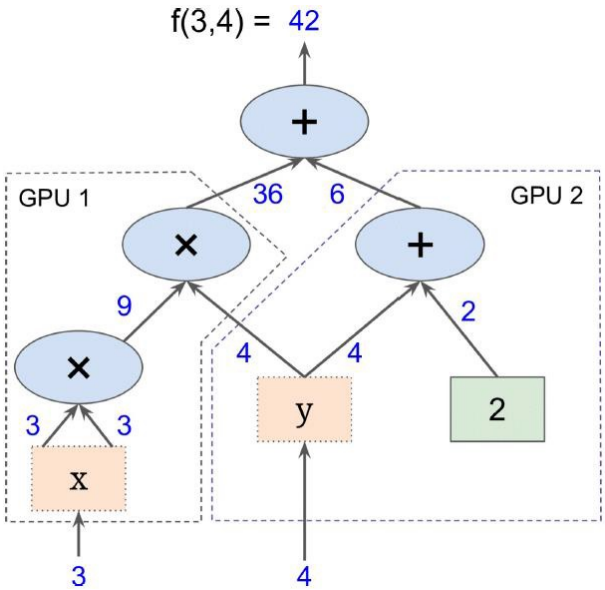
A computational graph is a directed graph where nodes correspond to **operations** or **variables**. Every node in the graph defines a function of the variables.

Let's see an example. The operation $Ax + b$ can be modeled through a computational graph as follows:



Advantages of using a *computational graph*?

- **Automatic differentiation**: derivatives are automatically calculated, you do not need to write the closed formula (particularly useful when the objective function becomes very difficult and nested)
- **Save computations** (run only part of the graph that lead to the values you want)
- Facilitate **distributed computations**, spread the work across multiple CPUs, GPUs, or devices



Tensorflow works in two stages:

- Create the graph: define all the computations of the graph
- Run a Session to execute the command: run the part of the graph you need

Basic of Tensorflow

In [1]:

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

Out[1]:

```
'1.11.0'
```

Tensorflow works with three main types of objects:

- constants
- Variables
- placeholders

Constants

To create a constant we can use tf.constant()

In [2]:

```
a = tf.constant(5)
```

What do we get if we ask to print a?

In [3]:

```
print(a)
```

```
Tensor("Const:0", shape=(), dtype=int32)
```

It prints the object. As I told you before, Tensorflow needs two stages:

- definition of the graph
- running of the part we are interested in

To obtain the value we need to run a session

In [4]:

```
with tf.Session() as sess:
    print(sess.run(a))
```

```
5
```

Let's compute something more difficult: $x = a + b$

In [5]:

```
#definition of the graph
a = tf.constant(5)
b = tf.constant(3)
x=tf.add(a,b)
#run the session
with tf.Session() as sess:
    print(sess.run(x))
```

```
8
```

Other simple operations in Tensorflow are:

Tensorflow Operations

Operation	Description	Operation	Description
tf.add	sum	tf.square	calculates the square
tf.sub	substraction	tf.round	nearest integer
tf.mul	multiplication	tf.sqrt	square root
tf.div	division	tf.pow	calculates the power
tf.mod	module	tf.exp	exponential
tf.abs	absolute value	tf.log	logarithm
tf.neg	negative value	tf.cos	calculates the cosine
tf.inv	inverse	tf.sin	calculates the sine
tf.maximum	returns the maximum	tf.matmul	tensor product
tf.minimum	returns the minimum	tf.transpose	tensor transpose

For more advanced operations, have a look here (https://www.tensorflow.org/api_docs/python/tf/math)

You can also define tensors of zeros/ones/from probability distributions etc...

In [6]:

```
a=tf.zeros([2,2])
b=tf.ones([3,3])
c=tf.random_normal([3,3])
with tf.Session() as sess:
    print('a',sess.run(a))
    print('b',sess.run(b))
    print('c',sess.run(c))
```

```
a [[0. 0.]
 [0. 0.]]
b [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
c [[ 0.4165901   0.62163204 -1.3165778 ]
 [ 0.2636855  -0.6549473   1.1168449 ]
 [-0.28966084  0.69227797  0.41397953]]
```

Variables

Variables are the numerical part of the graph that may change during the computations. Variables have their own class, that is why we *need* the the capital c to create them

In [7]:

```
a = tf.Variable(2, name="scalar")
b = tf.Variable([2,2], name="vector")
c = tf.Variable([[2,2],[2,2]], name="matrix")
```

When working with variables, we need to **initialize them** before running

In [8]:

```
with tf.Session() as sess:
    #initialize all the variables
    sess.run(tf.global_variables_initializer())
    #print the variable
    print(a)
```

```
<tf.Variable 'scalar:0' shape=() dtype=int32_ref>
```

print(a) does not return the value of a.

If we want to obtain the value of a Variable we need to use the *eval()* method

In [9]:

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
    #initialize all the variables
    sess.run(init)
    #print the value of a
    print(a.eval())
```

```
2
```

Placeholder

Values we do not know when we create the graph. In a ML application a placeholder may represent the dataset $\{X, y\}$

Suppose we want to evaluate

$$w^T x = \begin{bmatrix} 5 \\ 5 \\ 5 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

but we do not know *a-priori* the values of x.

We can rely on placeholders!

In [10]:

```
w = tf.constant([5.0, 5.0, 5.0])
x = tf.placeholder(tf.float32, shape=[3]) #specify the shape of the placeholder

w_x = tf.tensordot(x, w, axes=1) #dot product, like np.dot()
```

Now, if $x = [1, 2, 3]$ we can feed values to placeholders by passing them **as a dictionary**

In [11]:

```
with tf.Session() as sess:

    print(sess.run(w_x, {x : [1, 2, 3]}))
```

```
30.0
```

Note that when running something depending on the placeholder, we need to specify the values of the placeholder

if $x = [10, 20, 30]$ we just need to change the values in the dictionary

In [12]:

```
with tf.Session() as sess:

    print(sess.run(w_x, {x : [10, 20, 30]}))
```

```
300.0
```

After having seen the basics of Tensorflow, we can create our first model!

A Multinomial regression model

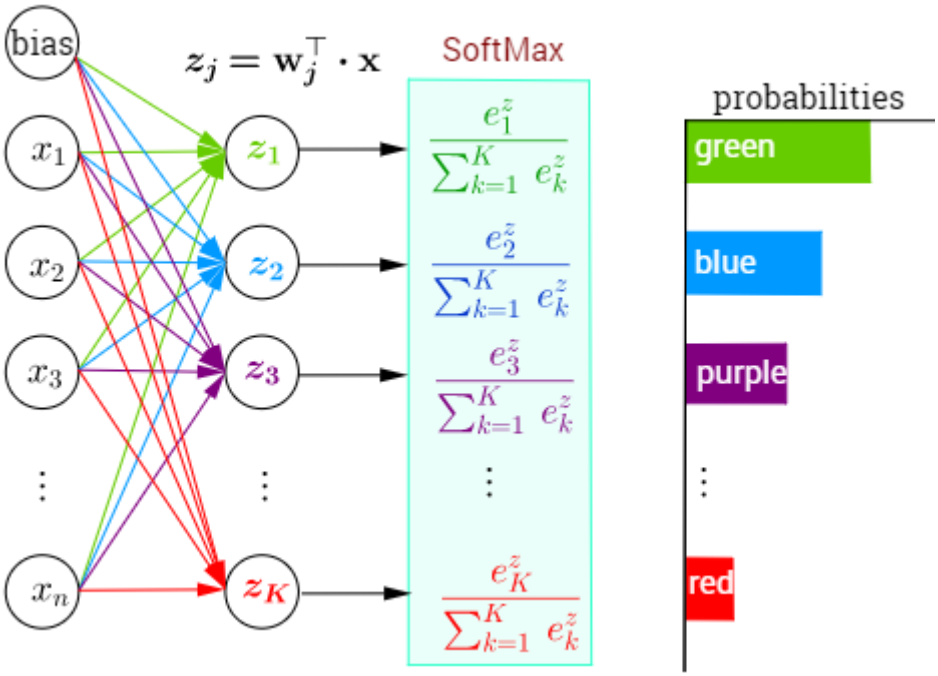
A multinomial regression is a regression problem where the output is a categorical variable with more than two possible labels (K labels)

Example: Given some pictures of handwritten digits, I want to predict the number drawn in each picture ($0, 1, \dots, 9$) (in this case $K = 10$)

Given an instance X^p, first we apply the usual transofrmation $w^T x + b$ and then we apply the softmax (https://deepnotes.io/softmax-crossentropy) operator σ which takes in input a vector $z^p \in R^K$ and returns a new vector $y^p \in R^K$ such that

$$y_k^p \geq 0 \qquad \sum_{k=1}^{K} y_k^p = 1$$

Each component $y_k^p \in (0, 1)$ represents the probability of the instance X^p of belonging to the class k. The predicted class of X^p will be the class with the highest probability



Let's create training and test set

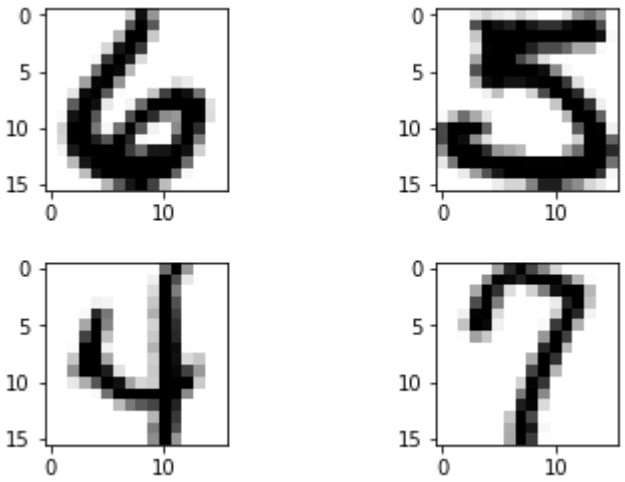
In [14]:

```
import pandas as pd
import numpy as np
train=pd.read_csv('MNIST_Train.csv',header=None, sep= ' ')
test=pd.read_csv('MNIST_Test.csv',header=None, sep= ' ')

#dumb operations to obtain the dataset
X_train=np.array(train.iloc[:,1:-1])
y_train=np.array(train.iloc[:,0])
X_test=np.array(train.iloc[:,1:-1])
y_test=np.array(train.iloc[:,0])

n_sample, n_features = X_train.shape
num_classes=10
```

Let's have a look at our beatiful pics

In [15]:

```
import matplotlib.pyplot as plt
fig, axes = plt.subplots(2, 2)
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i, ax in enumerate(axes.flat):
    ax.imshow(X_train[i].reshape([16,16]), cmap='binary')
plt.show()
```



Since the softmax operator returns a probability for each class, we need to encode our data:

$$y = 3 \rightarrow y_{encoded} = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$$

In [16]:

```
from sklearn.preprocessing import LabelBinarizer
enc = LabelBinarizer()
y_train_encoded = enc.fit_transform(y_train)
y_test_encoded = enc.fit_transform(y_test)
```

In [17]:

```
# let's see the encoding
print(y_train[0:3])
print(y_train_encoded[0:3])
```

```
[6. 5. 4.]
[[0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0]]
```

1) Definition of the graph

- Define the **placeholders** (how we will pass data for training the model)

In [18]:

```
x = tf.placeholder(tf.float32, [None, n_features])
y_true = tf.placeholder(tf.float32, [None, num_classes])
y_true_cls = tf.placeholder(tf.int64, [None])
```

Note that we have not specified the number of rows of the input. **None** means that the dimension may change, it is not fixed! (indeed we will use the same placeholders for both training and test and they have different number of rows/samples)

- Define the **variables**:

$$w \in R^{256 \times 10} \qquad b \in R^{10}$$

In [19]:

```
weights = tf.Variable(tf.random_normal([n_features, num_classes]))
biases = tf.Variable(tf.zeros([num_classes]))
```

We have 2660 variables in our model

- Define the **model**:

In [20]:

```
logits = tf.matmul(x, weights) + biases
```

- Define the **objective function** we want to minimize through the croosentropy (https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html)

(Note we did not define the softmax operator because the tf.nn.softmax_cross_entropy_with_logits_v2 function already does it for us)

In [21]:

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits,
                                                           labels=y_true)

cost = tf.reduce_mean(cross_entropy) #evaluate the mean of the loss
```

- Define the **optimizer**:

In [22]:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5).minimize(cost)
```

- To check the model's ability to generalize, we define also the **accuracy** (which will not be used to train the model but only to check its performance on the test set)

In [23]:

```
y_pred = tf.nn.softmax(logits) #here we need to define the softmax operator
y_pred_cls = tf.argmax(y_pred, axis=1) #get the higher value
correct_prediction = tf.equal(y_pred_cls, y_true_cls) #check if we predicted correctly
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)) #evaluate the accura
cy
```

We can define *a-priori* the placeholder we will use to check the accuracy on the test set

In [24]:

```
feed_dict_test = {x: X_test,
                  y_true: y_test_encoded,
                  y_true_cls: y_test}
```

In order to make the code cleaner, we can define the *optimize* function which runs *numiter* iterations of the optimizer

In [25]:

```
def optimize(num_iter,batch_size=100):
    ''' Performs num_iter of Stochastic Gradient Descent steps
    on the batch_size defined in input '''

    for i in range(num_iter):

        #at each iteration we select a minibatch
        index=np.random.choice(X_train.shape[0],batch_size)
        x_batch, y_true_batch= X_train[index,:],y_train_encoded[index]

        # Put the minibatch into a dictionary
        feed_dict_train = {x: x_batch, y_true: y_true_batch}

        #run the optimizer
        sess.run(optimizer, feed_dict=feed_dict_train)
```

2) Running the graph

In [26]:

```
num_iter=100
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for k in range(25):
        print('Accuracy after ', (k+1)*num_iter , 'iterations: {0:.1%}' .format(sess.ru
n(accuracy, feed_dict=feed_dict_test)))
        optimize(num_iter)
```

```
Accuracy after  100 iterations: 8.5%
Accuracy after  200 iterations: 86.8%
Accuracy after  300 iterations: 89.8%
Accuracy after  400 iterations: 91.9%
Accuracy after  500 iterations: 93.0%
Accuracy after  600 iterations: 93.1%
Accuracy after  700 iterations: 94.1%
Accuracy after  800 iterations: 94.4%
Accuracy after  900 iterations: 94.7%
Accuracy after  1000 iterations: 94.1%
Accuracy after  1100 iterations: 94.5%
Accuracy after  1200 iterations: 95.0%
Accuracy after  1300 iterations: 95.3%
Accuracy after  1400 iterations: 95.5%
Accuracy after  1500 iterations: 95.9%
Accuracy after  1600 iterations: 95.9%
Accuracy after  1700 iterations: 95.6%
Accuracy after  1800 iterations: 95.8%
Accuracy after  1900 iterations: 95.9%
Accuracy after  2000 iterations: 96.5%
Accuracy after  2100 iterations: 96.4%
Accuracy after  2200 iterations: 96.6%
Accuracy after  2300 iterations: 96.7%
Accuracy after  2400 iterations: 96.8%
Accuracy after  2500 iterations: 96.8%
```

Tensorflow is not exactly straightforward. Let's see an high level library which allow us to exploit the power of Tensorflow without sinking in graph computations.

Keras

Keras is an open source library focused on Deep Neural Networks models. Its application is extremely easy and in a couple of lines we can build a **Deep** network.

In [27]:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
```

```
Using TensorFlow backend.
```

We will define a Deep Network for the classification problem of above.

Let's define the model:

Since we are working with *Feedforward* networks, let's create the object (a sequential NN) and then *add* layers:

In [28]:

```
# create model
model = Sequential()
#let's add a layer with 256 neurons and the sigmoid as activation function
# a Dense layer is a classical fully connected layer
model.add(Dense(200, input_dim=256, kernel_initializer='normal', activation='sigmoid'))
#the last layer has many output as the classes we want to distinguish
model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))
```

After haveing built the model we need to define what we want to minimize, the *loss*, and how, the *optimizer*.

In [29]:

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

Similarly to what we did with Sklearn, we can now fit the model on the training set

In [30]:

```
model.fit(X_train, y_train_encoded, epochs=30, batch_size=200)
```

```
Epoch 1/30
7291/7291 [==============================] - 1s 129us/step - loss: 1.6205
- acc: 0.5463
Epoch 2/30
7291/7291 [==============================] - 0s 44us/step - loss: 0.7981 -
acc: 0.8563
Epoch 3/30
7291/7291 [==============================] - 0s 47us/step - loss: 0.4816 -
acc: 0.9039
Epoch 4/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.3495 -
acc: 0.9251
Epoch 5/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.2841 -
acc: 0.9331
Epoch 6/30
7291/7291 [==============================] - 0s 44us/step - loss: 0.2450 -
acc: 0.9394
Epoch 7/30
7291/7291 [==============================] - 0s 42us/step - loss: 0.2180 -
acc: 0.9458
Epoch 8/30
7291/7291 [==============================] - 0s 44us/step - loss: 0.1980 -
acc: 0.9498
Epoch 9/30
7291/7291 [==============================] - 0s 44us/step - loss: 0.1829 -
acc: 0.9546
Epoch 10/30
7291/7291 [==============================] - 0s 44us/step - loss: 0.1715 -
acc: 0.9573
Epoch 11/30
7291/7291 [==============================] - 0s 42us/step - loss: 0.1604 -
acc: 0.9590
Epoch 12/30
7291/7291 [==============================] - 0s 42us/step - loss: 0.1508 -
acc: 0.9627
Epoch 13/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.1430 -
acc: 0.9634
Epoch 14/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.1371 -
acc: 0.9645
Epoch 15/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.1287 -
acc: 0.9680
Epoch 16/30
7291/7291 [==============================] - 0s 44us/step - loss: 0.1235 -
acc: 0.9697
Epoch 17/30
7291/7291 [==============================] - 0s 42us/step - loss: 0.1173 -
acc: 0.9704
Epoch 18/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.1129 -
acc: 0.9716
Epoch 19/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.1066 -
acc: 0.9735
Epoch 20/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.1023 -
acc: 0.9756
Epoch 21/30
```

```
7291/7291 [==============================] - 0s 47us/step - loss: 0.0985 -
acc: 0.9763
Epoch 22/30
7291/7291 [==============================] - 0s 46us/step - loss: 0.0941 -
acc: 0.9772
Epoch 23/30
7291/7291 [==============================] - 0s 44us/step - loss: 0.0898 -
acc: 0.9785
Epoch 24/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.0861 -
acc: 0.9796
Epoch 25/30
7291/7291 [==============================] - 0s 42us/step - loss: 0.0832 -
acc: 0.9805
Epoch 26/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.0794 -
acc: 0.9818
Epoch 27/30
7291/7291 [==============================] - 0s 42us/step - loss: 0.0760 -
acc: 0.9833
Epoch 28/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.0727 -
acc: 0.9831
Epoch 29/30
7291/7291 [==============================] - 0s 43us/step - loss: 0.0707 -
acc: 0.9834
Epoch 30/30
7291/7291 [==============================] - 0s 42us/step - loss: 0.0673 -
acc: 0.9845
```

Out[30]:

```
<keras.callbacks.History at 0x1ffebab5ac8>
```

And then check its performance on the test set

In [31]:

```
# Final evaluation of the model
scores = model.evaluate(X_test, y_test_encoded, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

```
Baseline Error: 1.39%
```