How to build & run your first deep learning network in TensorFlow

Jordi Torres & Maurici Yagües

Barcelona, July 2016

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

DAY 2

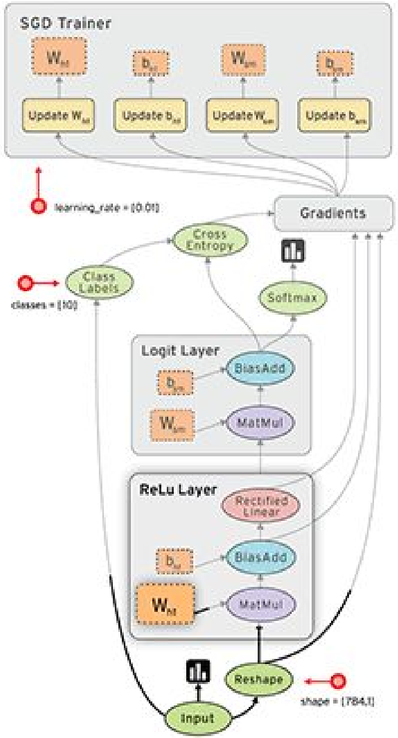
Basic data structures in TensorFlow

Case study: Clustering in TensorFlow

Parallelism and Distribution in TensorFlow

Core TensorFlow concepts (reminder)

- **Graph**: A TensorFlow computation, represented as a dataflow graph
- **Operation**: a graph node that performs computation on tensors
- **Tensor**: a handle to one of the outputs of an operation



Tensor: Core TensorFlow data structure (reminder)

- **Constants**
- **Placeholders**: must be fed with data on execution
- **Variables:** a modifiable tensor that lives in TensorFlow's graph of interacting operations
- **Session:** encapsulates the environment in which operation objects are executed, and Tensor objects are evaluated

Basic data type: Tensor

- Tensor can be considered a dynamically-sized multidimensional data arrays
- Main types and their equivalent in Python:

Type in TensorFlow	Type in Python	Description
DT_FLOAT	tf.float32	Floating point of 32 bits
DT_INT16	tf.int16	Integer of 16 bits
DT_INT32	tf.int32	Integer of 32 bits
DT_INT64	tf.int64	Integer of 64 bits
DT_STRING	tf.string	String
DT_BOOL	tf.bool	Boolean

Basic data type: Tensor

- Each tensor has a rank, which is the number of its dimensions.

- For example, the following tensor (defined as a list in Python) has rank 2:

```
t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Basic data type: Tensor

- TensorFlow (documentation) uses three types of naming conventions

Shape	Rank	Dimension Number
[]	0	0-D
[D0]	1	1-D
[D0, D1]	2	2-D
[D0, D1, D2]	3	3-D
…	…	…
[D0, D1, ... Dn]	n	n-D

Transformations

- Tensors can be manipulated with a series of transformations that supply the TensorFlow package

Operation	Description
tf.shape	To find a shape of a *tensor*
tf.size	To find the size of a *tensor*
tf.rank	To find a rank of a *tensor*
tf.reshape	To change the shape of a *tensor* keeping the same elements contained
tf.squeeze	To delete in a *tensor* dimensions of size 1
tf.expand_dims	To insert a dimension to a *tensor*

Transformations (cont.)

Operation	Description
tf.slice	To remove a portions of a *tensor*
tf.split	To divide a *tensor* into several tensors along one dim
tf.tile	To create a new *tensor* replicating it multiple times
tf.concat	To concatenate *tensors* in one dimension
tf.reverse	To reverse a specific dimension of a *tensor*
tf.transpose	To transpose dimensions in a *tensor*
tf.gather	To collect portions according to an index

Tensor transformations

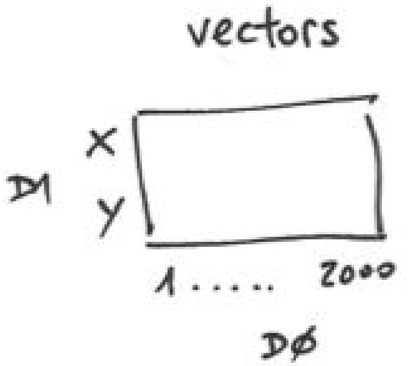
- For example, suppose that we want to extend an array of 2x2000 (a 2D tensor) to a cube (3D tensor).
- We can use the **`tf.expand_dims`** function, which allows us to insert a dimension to a tensor (the dimensions start at zero):

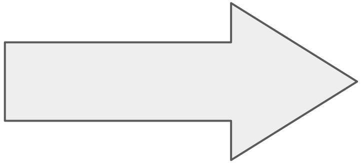
```
vectors = tf.constant(vectors_set)
extended_vectors = tf.expand_dims(vectors, 0)
```

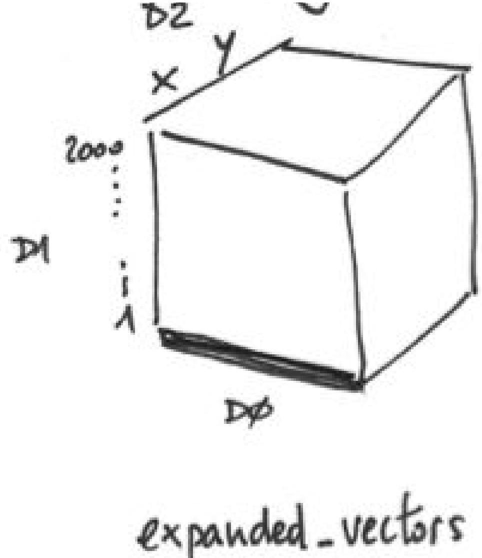
Tensor transformations

- For example, suppose that we want to extend an array of 2x2000 (a 2D tensor) to a cube (3D tensor).
- We can use the **`tf.expand_dims`** function, which allows us to insert a dimension to a tensor (the dimensions start at zero):

```
vectors = tf.constant(vectors_set)
extended_vectors = tf.expand_dims(vectors, 0)
```







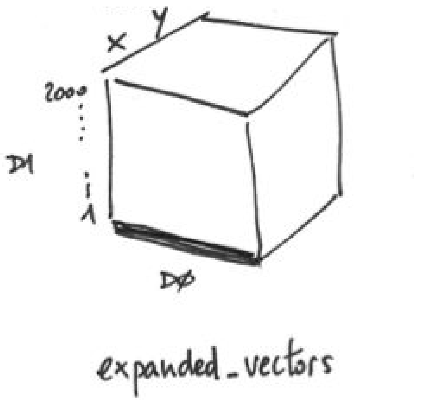
Tensor shape

- If we obtain the shape of this tensor with the `.get_shape()` operation:

```
print expanded_vectors.get_shape()
```

- we can see that there is no associated size:

```
TensorShape([Dimension(1), Dimension(2000), Dimension(2)])
```



Data storage in TensorFlow

There are three main ways of obtaining data on a TensorFlow program:

1. From data files.
2. Data preloaded as constants or variables.
3. Those provided by Python code.

Data storage in TensorFlow

1. **From data files:** example **`input_data.py`**

```
SOURCE_URL = 'http://yann.lecun.com/exdb/mnist/'
TRAIN_IMAGES = 'train-images-idx3-ubyte.gz'

local_file = maybe_download(TRAIN_IMAGES, train_dir)
train_images = extract_images(local_file)
```

```
def maybe_download(filename, work_directory):
  """Download the data from Yann's website, unless it's already here."""
  if not os.path.exists(work_directory):
    os.mkdir(work_directory)
  filepath = os.path.join(work_directory, filename)
  if not os.path.exists(filepath):
    filepath, _ = urllib.request.urlretrieve(SOURCE_URL + filename, filepath)
    statinfo = os.stat(filepath)
    print('Successfully downloaded', filename, statinfo.st_size, 'bytes.')
  return filepath
```

```
def extract_images(filename):
  """Extract the images into a 4D uint8 numpy array [index, y, x, depth]."""
  print('Extracting', filename)
  with gzip.open(filename) as bytestream:
    magic = _read32(bytestream)
    if magic != 2051:
      raise ValueError(
          'Invalid magic number %d in MNIST image file: %s' %
          (magic, filename))
    num_images = _read32(bytestream)
    rows = _read32(bytestream)
    cols = _read32(bytestream)
    buf = bytestream.read(rows * cols * num_images)
    data = numpy.frombuffer(buf, dtype=numpy.uint8)
    data = data.reshape(num_images, rows, cols, 1)
    return data
```

Data storage in TensorFlow

2. **Data preloaded as a:**
    - Constant using **`tf.constant(…)`**
    - Variable using **`tf.Variable`**(…)

TensorFlow package offers different operations that can be used to generate constants and variables:

Constants generation

Operation	Description
tf.zeros_like	Creates a tensor with all elements initialized to 0
tf.ones_like	Creates a tensor with all elements initialized to 1
tf.fill	Creates a tensor with all elements initialized to a scalar value given as argument
tf.constant	Creates a tensor of constants with the elements listed as an arguments

Variable Tensor random generation

Operation	Description
tf.random_normal	Random values with a normal distribution
tf.truncated_normal	Random values with a normal distribution but eliminating those values whose magnitude is more than 2 times the standard deviation
tf.random_uniform	Random values with a uniform distribution
tf.random_shuffle	Randomly mixed tensor elements in the first dimension
tf.set_random_seed	Sets the random seed

Data storage in TensorFlow

3. **Provided by Python code:**

   The call is **tf.`placeholder()`**, which includes arguments with the type of the elements and the shape of the tensor, and optionally a name.

```
import tensorflow as tf

a = tf.placeholder("float")
b = tf.placeholder("float")

y = tf.mul(a, b)

sess = tf.Session()
print sess.run(y, feed_dict={a: 3, b: 3})
```

*With calls in `Session.run()` or `Tensor.eval()` this tensor is populated with the data specified in the **feed_dict** parameter*

Case study: K-means algorithm

- K-means: Unsupervised algorithm which solves the clustering problem
    - Its procedure follows a simple and easy way to classify a given data set through a certain number of clusters (**assume k clusters**).
    - Data points inside a cluster are homogeneous and heterogeneous to peer groups, that means that all the elements in a subset are more similar to each other than with the rest.
    - The result of the algorithm is a set of K dots, called **centroids**, which are the focus of the different groups obtained, and the tag that represents the set of points that are assigned to only one of the K clusters. All the points within a cluster are closer in distance to the centroid than any of the other centroids.

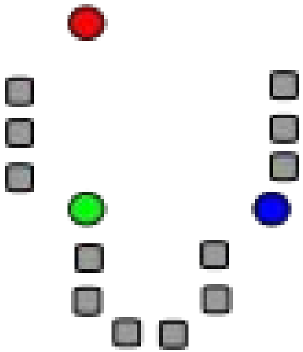
Case study: K-means algorithm

- K-means:
  - Computationally expensive problem (NP-hard problem)
  - Using algorithms that converge rapidly in a local optimum by heuristics
  - The most commonly used algorithm uses an iterative refinement technique.
- Steps:
  1. Initial step: **determines an initial set of K centroids.**
  2. Allocation step: **assigns each observation to the nearest group.**
  3. Update step: **calculates the new centroids for each new group.**
  4. Steps 2 and 3 are repeated until convergence has been reached.

k-means algorithm: visual example

1. Initial step: **determines an initial set of K centroids.**

   k initial "means" (in this case *k* = 3) are randomly generated within the data domain (shown in color)

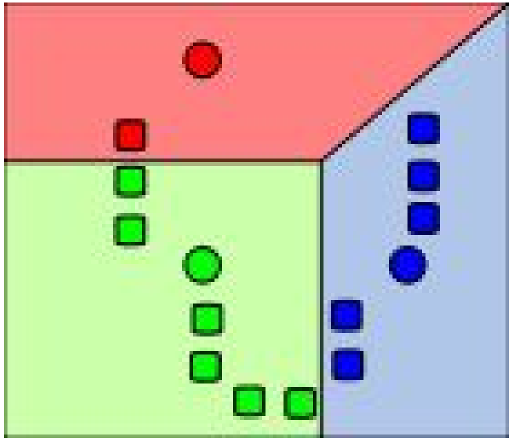


Source: https://en.wikipedia.org/wiki/K-means_clustering

k-means algorithm: visual example

2. Allocation step: **assigns each observation to the nearest group.**

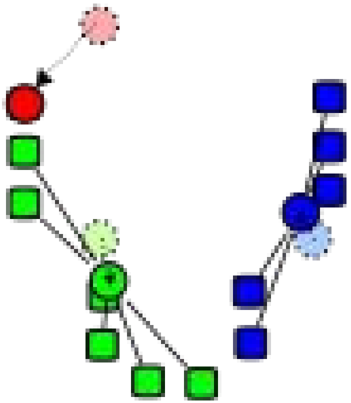
   k clusters are created by associating every observation with the nearest mean. The partitions here represent Voronoi diagram generated by the means



k-means algorithm: visual example

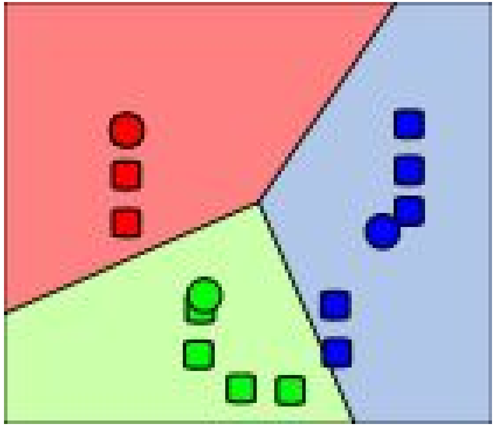
3. Update step: **calculates the new centroids for each new group.**

   The centroid of each of the *k* clusters becomes the new mean. *k* clusters are created by associating every observation with the nearest mean



k-means algorithm: visual example

4. Steps 2 and 3 are repeated until convergence has been reached



- As it is a heuristic algorithm, there is no guarantee that it will converge to the global optimum, and the result may depend on the initial clusters
- As the algorithm is usually very fast, it is common to run it multiple times with different starting conditions

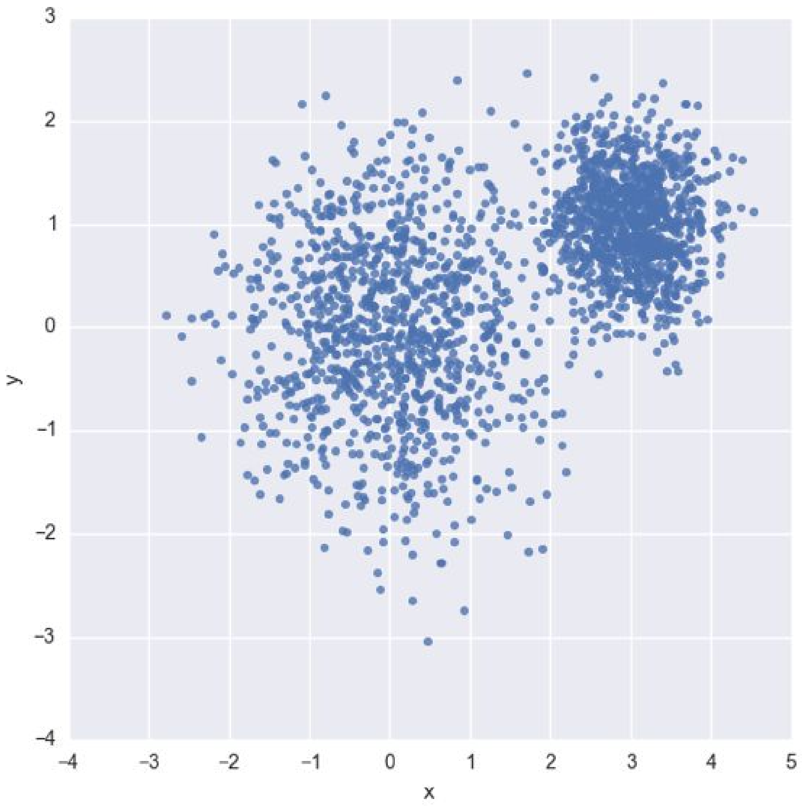
K-means: one testbed example

- *Generate 2000 points in a 2D space in a random manner (following 2 normal distributions)*

```
num_puntos = 2000
conjunto_puntos = []

for i in xrange(num_puntos):
  if np.random.random() > 0.5:
    conjunto_puntos.append([np.random.normal(0.0, 0.9),
                            np.random.normal(0.0, 0.9)])
  else:
    conjunto_puntos.append([np.random.normal(3.0, 0.5),
                            np.random.normal(1.0, 0.5)])
```

K-means: one testbed example




```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

df = pd.DataFrame({"x": [v[0] for v in vectors_set],
                   "y": [v[1] for v in vectors_set]})
sns.lmplot("x", "y", data=df, fit_reg=False, size=6)
plt.show()
```

K-means: one testbed example (step by step)

1. **Initial step: determines an initial set of K centroids**

- The first thing to do is move all our data to tensors:

```
vectors = tf.constant(vectors_set)
```

- Randomly choose K observations from the input data as centroids

```
k = 4
centroides = tf.Variable(tf.slice(tf.random_shuffle(vectors),[0,0],
[k,-1]))
```

(*)centroides.get_shape() ➞ TensorShape([Dimension(4), Dimension(2)])

K-means: one testbed example (step by step)

2. Allocation step: assigns each observation to the nearest group

- Calculate, for each point, its closest centroid by the *Squared Euclidean Distance*:

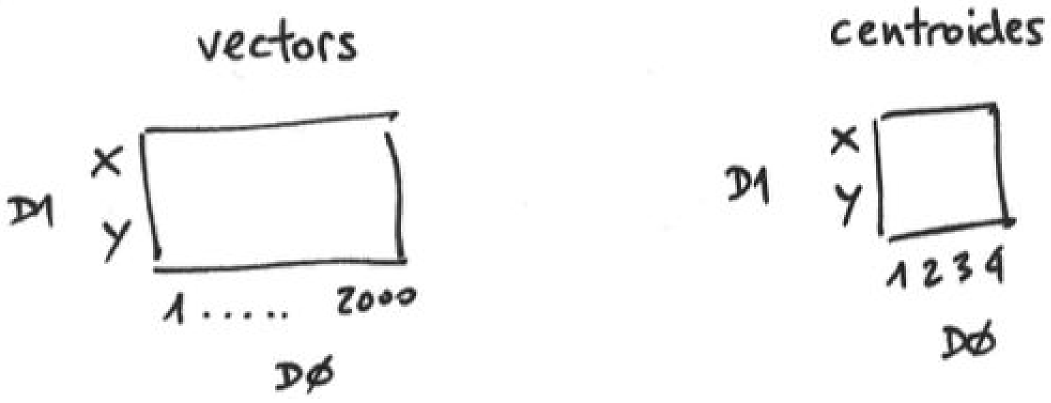
$$d^2(vector, centroide) = (vector_x - centroide_x)^2 + (vector_y - centroide_y)^2$$

- TensorFlow code ?

K-means: one testbed example (step by step)

(2. Allocation step: assigns each observation to the nearest group)

- **`tf.sub(vectors, centroides)`** is the main function used.
    - note that, although the two subtract tensors have both 2 dimensions, they have different sizes in one dimension (2000 vs 4 in dimension D0), which, in fact, also represent different things.



K-means: one testbed example (step by step)

- To fix this problem we could use some of the functions discussed before, for instance **`tf.expand_dims`** in order to insert a dimension in both tensors.

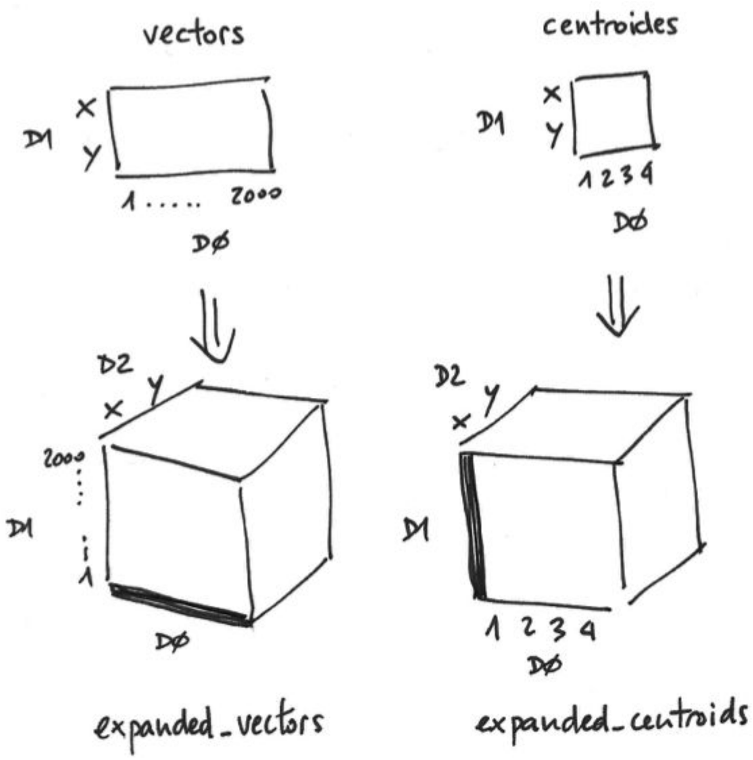
- The aim is to extend both tensors from 2 dimensions to 3 dimensions to make the sizes match in order to perform a subtraction:

```
expanded_vectors = tf.expand_dims(vectors, 0)
expanded_centroides = tf.expand_dims(centroides, 1)
```

K-means: one testbed example (step by step)

`tf.expand_dims`

- inserts one dimension in each tensor; in the first dimension (D0) of vectors tensor,
- and in the second dimension (D1) of centroids tensor.



K-means: one testbed example (step by step)

Note: there are dimensions that have not been able to determinate the sizes of those dimensions:

```
print expanded_vectors.get_shape()
print expanded_centroides.get_shape()

TensorShape([Dimension(1), Dimension(2000), Dimension(2)])
TensorShape([Dimension(4), Dimension(1), Dimension(2)])
```

(*) With 1 it is indicating a no assigned size.

TensorFlow allows **broadcasting**, and therefore the **`tf.sub`** function is able to discover for itself how to do the subtraction of elements between the two tensors.

K-means: one testbed example (step by step)

$$d^2(vector, centroide) = (vector_x - centroide_x)^2 + (vector_y - centroide_y)^2$$

- TensorFlow code ?

```
expanded_vectors = tf.expand_dims(vectors, 0)
expanded_centroides = tf.expand_dims(centroides, 1)

diff=tf.sub(expanded_vectors, expanded_centroides)
sqr= tf.square(diff)
```

*(*) tensor diff shape is **TensorShape([Dimension(4), Dimension(2000), Dimension(2)])** where D0 indicates the centroid , D1 the subtraction value and D2 each x,y point. sqr have the same shape.*

K-means: one testbed example (step by step)

2. Allocation step: assigns each observation to the nearest group.

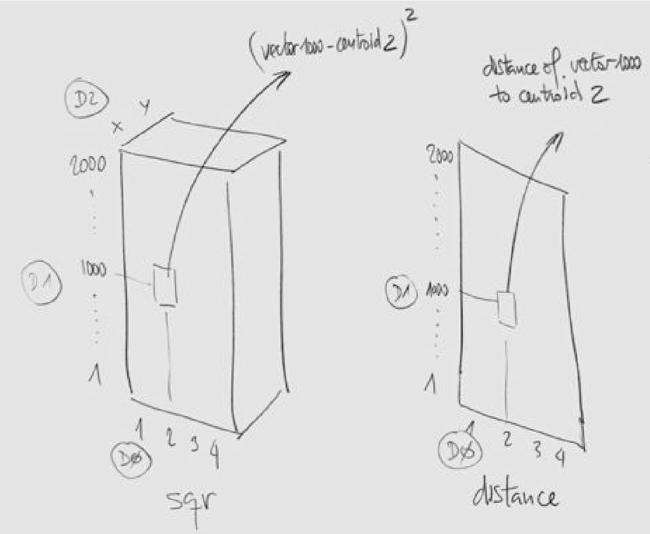
- Code that calculates, for each point, its closest centroid:

```
distances = tf.reduce_sum(sqr, 2)
assignments = tf.argmin(distances, 0)
```

K-means: one testbed example (step by step)

```
distances = tf.reduce_sum(sqr, 2)
```

- The **`distance`** tensor has already reduced one dimension, the one indicated as a parameter in tf.reduce_sum function TensorShape ([Dimension(4), Dimension(2000)])



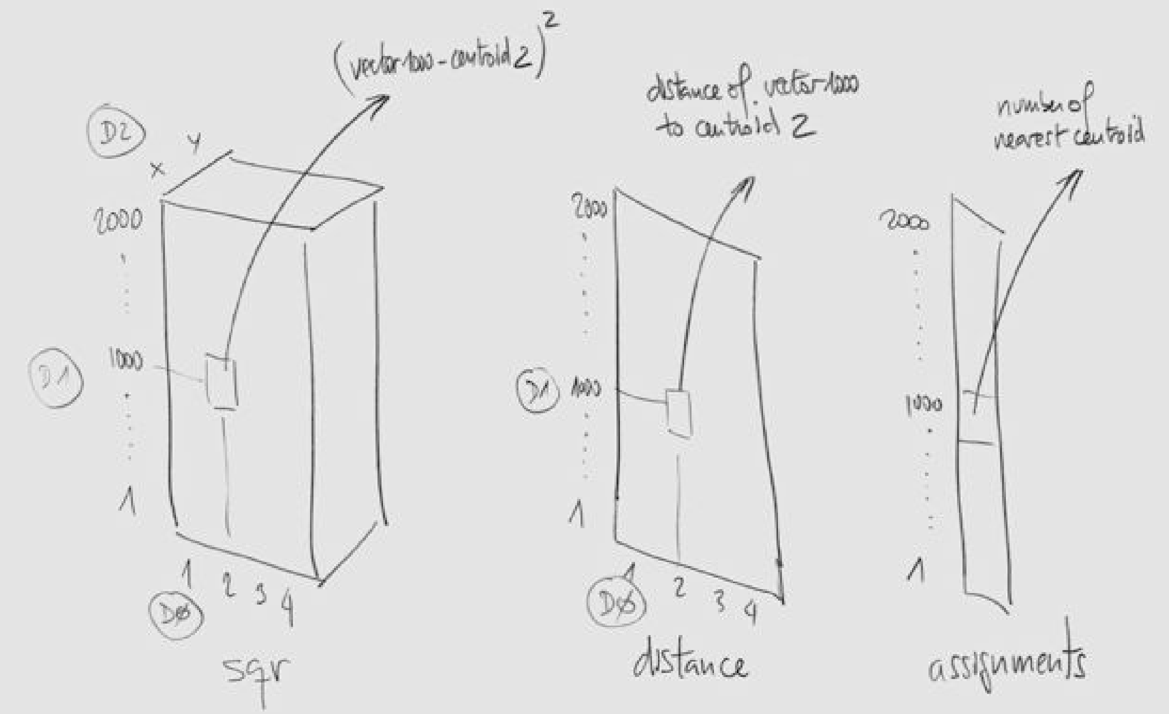
K-means: one testbed example (step by step)

```
assignments = tf.argmin(distances, 0)
```

- The assignation is achieved with tf.argmin, which returns the index with the minimum value of the tensor dimension (in our case D0, which remember that was the centroid). TensorShape([Dimension(2000)])

K-means: one testbed example (step by step)

```
distances = tf.reduce_sum(sqr, 2)
assignments = tf.argmin(distances, 0)
```



Tensor operations

Operation	Description
tf.reduce_sum	Computes the sum of elements along one dimensions of a tensor
tf.reduce_prod	Computes the product of elements along one dimensions of a tensor
tf.reduce_min	Computes the minimum of elements along one dimensions of a tensor
tf.reduce_max	Computes the maximum of elements along one dimensions of a tensor
tf.reduce_mean	Computes the mean of elements along one dimensions of a tensor

Tensor operations

Operation	Description
tf.argmin	Returns the index of the element with the minimum value along tensor dimension
tf.argmax	Returns the index of the element with the maximum value of the tensor dimension

K-means: one testbed example (step by step)

```
diff=tf.sub(expanded_vectors, expanded_centroides)
sqr= tf.square(diff)
distances = tf.reduce_sum(sqr, 2)
assignments = tf.argmin(distances, 0)
```

- Alternative TensorFlow code used in my book (*)

```
assignments = tf.argmin(tf.reduce_sum(tf.square(tf.sub
(expanded_vectors,
              expanded_centroides)), 2), 0)
```

(*) Github, (2016) Shawn Simister. [Online]. Available at: https://gist.github.com/narphorium/d06b7ed234287e319f18

K-means: one testbed example (step by step)

3. Update step: calculates the new centroids for each new group

- We create a tensor that contains the result of the concatenation of the k tensors that correspond to the mean value of every point that belongs to each k cluster (*):

```
means = tf.concat(0, [tf.reduce_mean(tf.gather(vectors, tf.reshape
(tf.where( tf.equal(assignments, c)),[1,-1])), reduction_indices=[1])
for c in xrange(k)])
```

(*) Github, (2016) Shawn Simister. [Online]. Available at: https://gist.github.com/narphorium/d06b7ed234287e319f18

(brief explanation)

- With `tf.equal` we can obtain a **boolean tensor** (*Dimension(2000))* that indicates (with *true* value) the positions where the *assignment tensor* match with the *K cluster*, which, at the time, we are calculating the average value of the points.

- With `tf.where` is constructed a tensor (*Dimension(1) x Dimension(2000))* with the position where the values *true* are on the *boolean tensor* received as a parameter. (i.e. a list of the position of these)

- With `tf.reshape` is constructed a tensor *(Dimension(2000) x Dimension(1))* with the index of the points inside *vectors* tensor that belongs to this *c cluster*

- With `tf.gather` is constructed a *tensor (Dimension(1) x Dimension(2000))* which gathers the coordinates of the points that form the *c cluster*

- With `tf.reduce_mean` it is constructed a tensor *(Dimension(1) x Dimension(2))* that contains the average value of all points that belongs to the cluster *c*

K-means: one testbed example

4. Graph Execution

```
update_centroides = tf.assign(centroides, means)

init_op = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init_op)

for step in xrange(100):
   _, centroid_values, assignment_values = sess.run([update_centroides,
                     centroides, assignments])
```

K-means: one testbed example

```
vectors = tf.constant(conjunto_puntos)
k = 4
centroides = tf.Variable(tf.slice(tf.random_shuffle(vectors),[0,0],[k,-1]))

expanded_vectors = tf.expand_dims(vectors, 0)
expanded_centroides = tf.expand_dims(centroides, 1)

assignments = tf.argmin(tf.reduce_sum(tf.square(tf.sub(expanded_vectors,
              expanded_centroides)), 2), 0)

means = tf.concat(0, [tf.reduce_mean(tf.gather(vectors, tf.reshape(tf.where( tf.equal
(assignments, c)),[1,-1])), reduction_indices=[1]) for c in xrange(k)])

update_centroides = tf.assign(centroides, means)

init_op = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init_op)

for step in xrange(100):
   _, centroid_values, assignment_values = sess.run([update_centroides,
                  centroides, assignments])
```

K-means: one testbed example

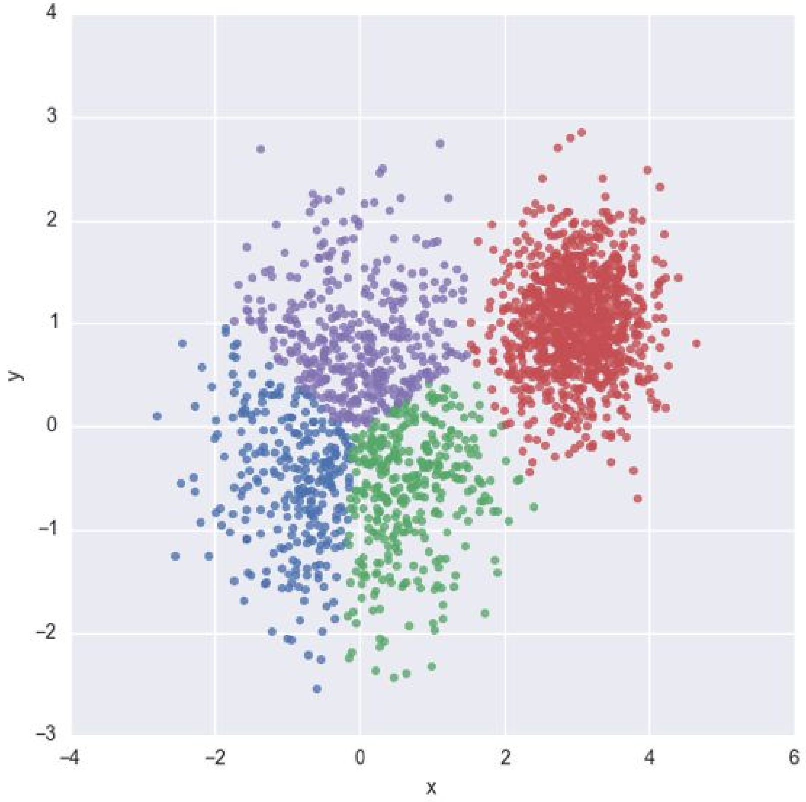
- We can use a simple print to know the centroids:

```
print centroid_values
```

- The output is as it follows:

```
[[ 2.99835277e+00 9.89548564e-01]
[ -8.30736756e-01 4.07433510e-01]
[ 7.49640584e-01 4.99431938e-01]
[ 1.83571398e-03 -9.78474259e-01]]
```

Case study: k-means




```
data = {"x": [], "y": [], "cluster": []}

for i in xrange(len(assignment_values)):
  data["x"].append(conjunto_puntos[i][0])
  data["y"].append(conjunto_puntos[i][1])
  data["cluster"].append(assignment_values[i])

df = pd.DataFrame(data)
sns.lmplot("x", "y", data=df,
           fit_reg=False, size=6,
           hue="cluster", legend=False)

plt.show()
```

Some words about Parallelism in TensorFlow

- The TensorFlow package, appearing in November 2015, was ready to run on servers with available GPUs and executing the training operation simultaneously in them
  - Requires the CudaToolkit and CUDNN packages

- In February 2016, an update added the capability to distribute and parallelize the processing

GPUs

- The way to reference those devices in TensorFlow is the following one:
  - "/cpu:0": To reference the server's CPU.
  - "/gpu:0": The server's GPU, if only one is available.
  - "/gpu:1": The second server's GPU, and so on

- To know (in the output) in which devices our operations and tensors are assigned we need to create a session with the option log_device_placement as True.

```
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
```

- If we want a specific operation to be executed in a specific device:

```
tf.device('/gpu:2')
```

Example of GPU use:

```
import tensorflow as tf

with tf.device('/gpu:2'):
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
c = tf.matmul(a, b)

sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
printsess.run(c)
```

Example with two GPUs :

```
import tensorflow as tf

c = []
for d in ['/gpu:2', '/gpu:3']:
with tf.device(d):
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
sum = tf.add_n(c)


sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
print sess.run(sum)
```

Output using `log_device_placement=True`

```
. . .
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: Tesla K40c
/job:localhost/replica:0/task:0/gpu:2 -> device: 2, name: Tesla K40c
/job:localhost/replica:0/task:0/gpu:3 -> device: 3, name: Tesla K40c
. . .

Const_3: /job:localhost/replica:0/task:0/gpu:3
I tensorflow/core/common_runtime/simple_placer.cc:289] Const_3: /job:localhost/replica:0/task:0/gpu:3
Const_2: /job:localhost/replica:0/task:0/gpu:3
I tensorflow/core/common_runtime/simple_placer.cc:289] Const_2: /job:localhost/replica:0/task:0/gpu:3
MatMul_1: /job:localhost/replica:0/task:0/gpu:3
I tensorflow/core/common_runtime/simple_placer.cc:289] MatMul_1: /job:localhost/replica:0/task:0/gpu:3
Const_1: /job:localhost/replica:0/task:0/gpu:2
I tensorflow/core/common_runtime/simple_placer.cc:289] Const_1: /job:localhost/replica:0/task:0/gpu:2
Const: /job:localhost/replica:0/task:0/gpu:2
I tensorflow/core/common_runtime/simple_placer.cc:289] Const: /job:localhost/replica:0/task:0/gpu:2
MatMul: /job:localhost/replica:0/task:0/gpu:2
I tensorflow/core/common_runtime/simple_placer.cc:289] MatMul: /job:localhost/replica:0/task:0/gpu:2
AddN: /job:localhost/replica:0/task:0/cpu:0
I tensorflow/core/common_runtime/simple_placer.cc:289] AddN: /job:localhost/replica:0/task:0/cpu:0
[[44.56.]
 [98.128.]]
. . .
```

Distributed TensorFlow: gRPC

More information: https://www.tensorflow.org/versions/r0.8/how_tos/distributed/index.html

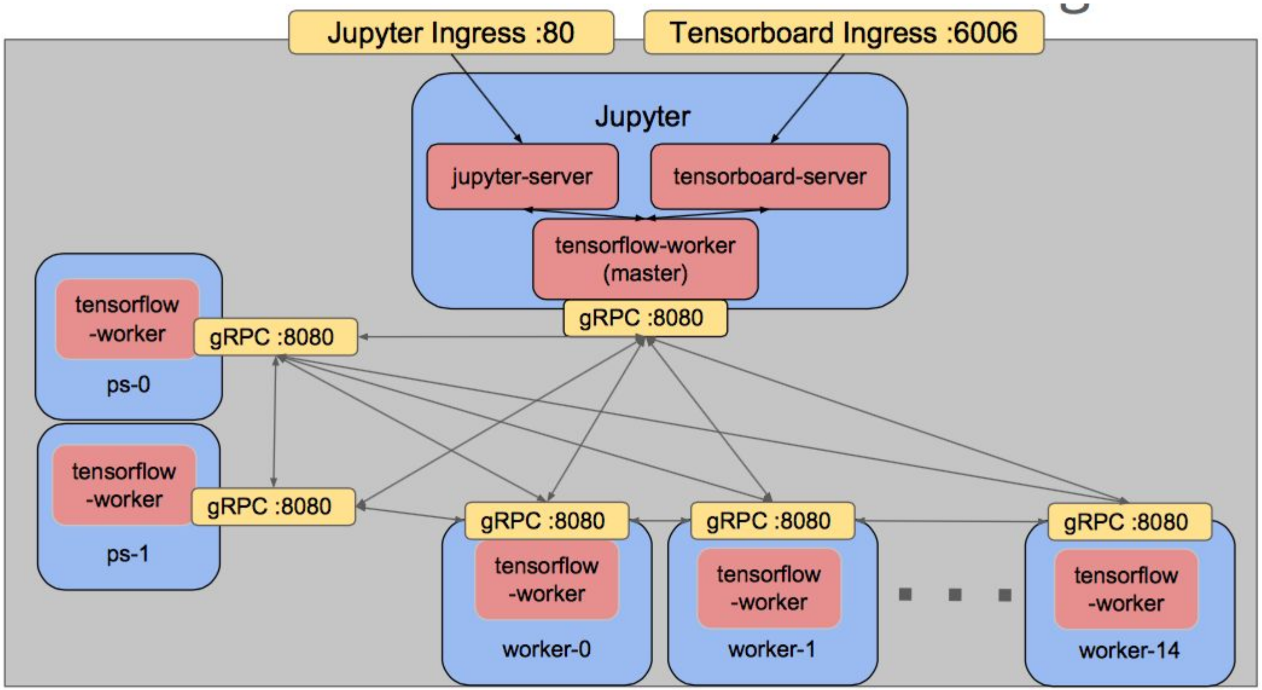


Image source: https://storage.googleapis.com/amy-jo/talks/tf-workshop.pdf

Class hands-on (or Homework)

- Reproduce the results (you can download the code from github) presented in this course in your laptop and show them to the instructors.

Next session:

- We will build a single layer neural network, step by step, with TensorFlow and use TensorBoard, a graph visualization tool