Lab 9 of SP19-BL-ENGR-E599-30563

Lab 9 will introduce the configuration of Tensorflow KMeans example.

Goal

- Install and configure Tensorflow (CPU version)
- Run KMeans example
- Basic Usage of Tensorboard

Deliverables

- Submit the computation time and centroid values with following parameters:
  - number of points: 10000
  - Dimension: 2
  - Centroids: 10
- Submit screenshots of the graph and loss function

Evaluation

Lab participation: credit for 1 point based upon a successful completion of the lab tasks

TensorFlow

Tensorflow is a machine learning library developed by Google. It is mostly used for Deep Learning. TensorFlow can run on multiple CPUs and GPUs.

TensorFlow Introduction

Tensor: an array of any number of dimensions

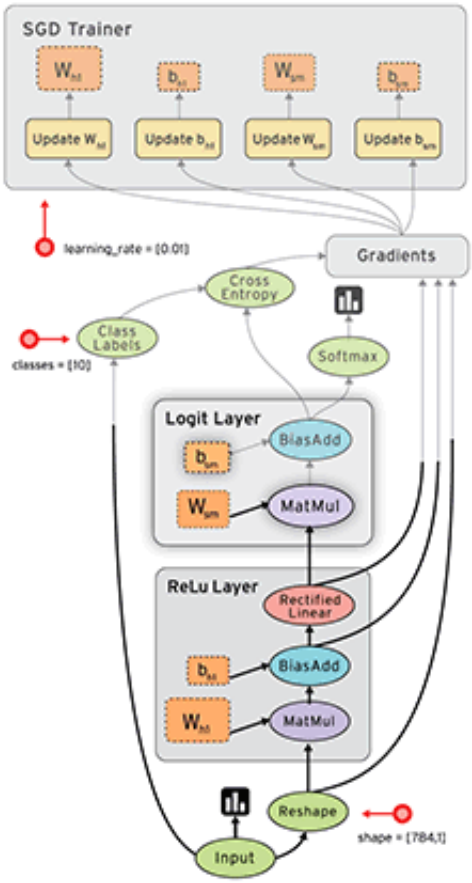
rank: number of dimension

shape: array lengths for each dimension

```
[1., 2., 3.] # rank 1, shape [3]
[[1., 2., 3.], [4., 5., 6.]] #rank 2, shape [2, 3]
```

Computational Graph

Tensorflow uses dataflow graph approach. User defines graph first and executes graph later.



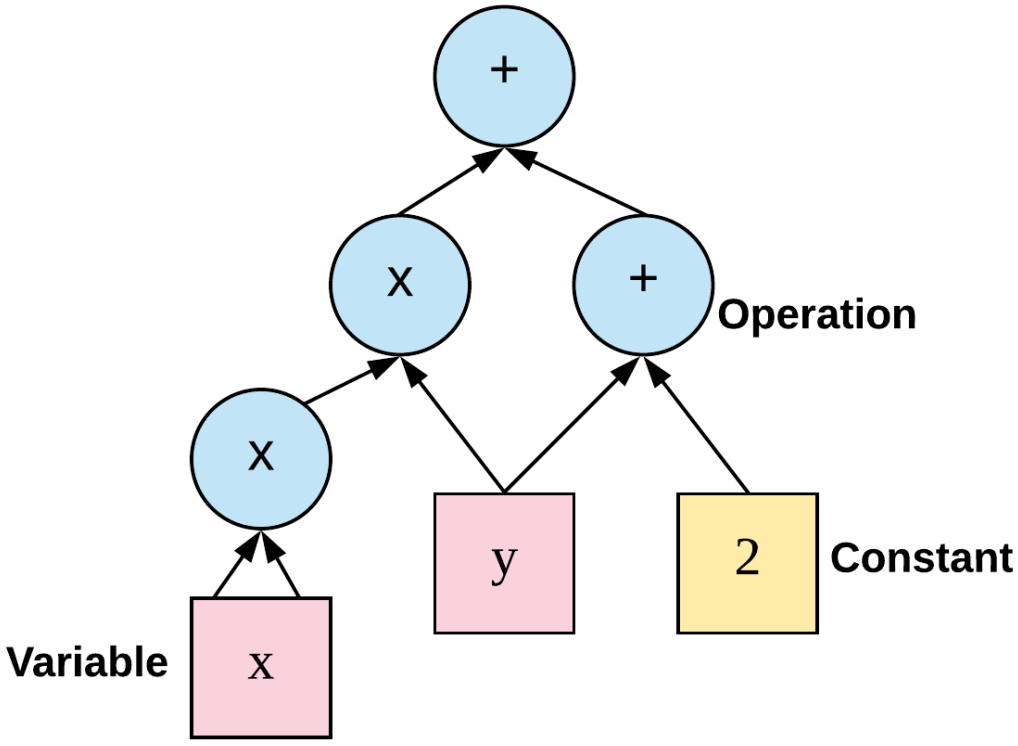
TensorFlow Computational Graph

Graph consists vertices and edges. Vertices represents tensorflow operations (such as matmul) and edges represents tensors.

Operations takes tensor(s) as input and produces a tensor as output.

Why Data Flow?

- **Parallelism:** It is easy to identify which operations can be run simultaneously. (Dependencies can be easliy seen)



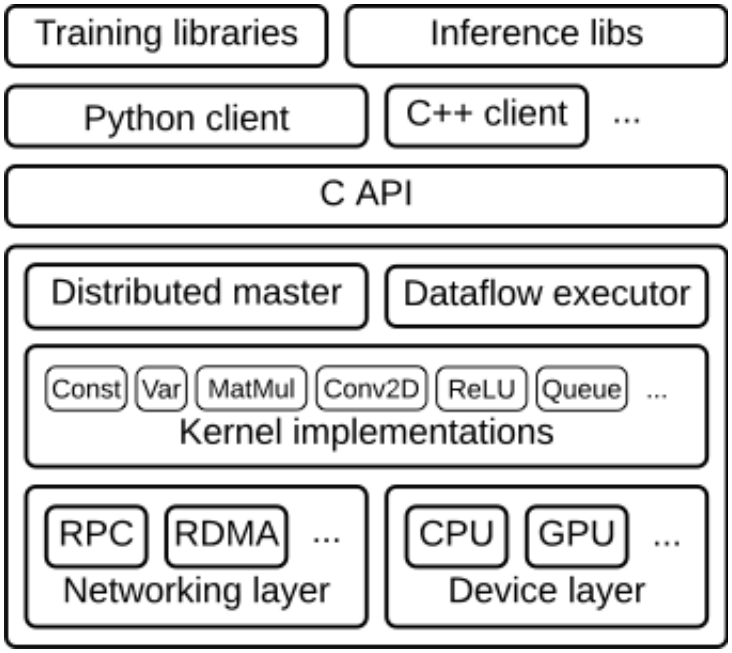
Simple Data Flow Graph (https://www.easy-tensorflow.com/tf-tutorials/basics/graph-and-session)

x*x and y+2 can be run simultaneously since there is no dependency.

- **Distributed execution:** Operations can be run on multiple devices (CPUs, GPUs, and TPUs) on different machines. TensorFlow uses MPI send and receive functions to send data to another machine.
- **Portability:** The dataflow graph is a language-independent. Graph can be created in python and used in C++.

TensorFlow Architecture

The figure below shows the basic tensorflow architecture.



TensorFlow Architecture

- **Client:** defines the computation graph and starts the execution of the graph.
- **Distributed master:** Partitions the subragphs into different workers and devices. Initiates execution of the graph piece.
- **Kernel implementations:** does the calculation for individual graph operations.

Constant

- Creates a tensor. Its value doesn't change.

```
const1 = tf.constant(5.0, dtype=tf.float32)
print(const)
#outputTensor("Const:0", shape=(), dtype=float32)
```

Session

Session executes the graph. (Build graph first and execute later)

```
sess = tf.Session()
print(sess.run(const1))
# output5.0
```

Variables

Variables are values (such as weight and biases) that can be changed in iterations.

```
my_int_variable = tf.get_variable("my_int_variable", [1, 2, 3], dtype=tf.int32,
  initializer=tf.zeros_initializer)
```

- **Variables needs to be initialized before use.**

```
session.run(tf.global_variables_initializer())
```

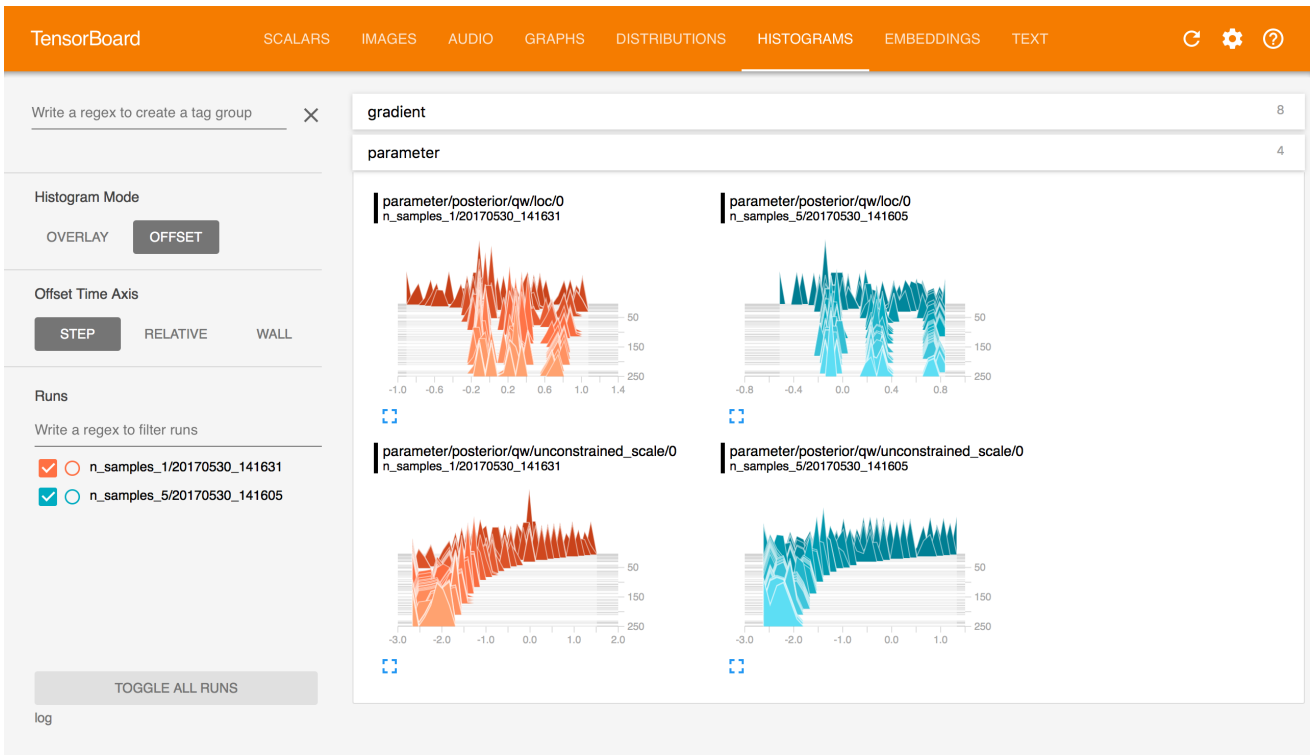
TensorFlow installation on Juliet Nodes

Run the code below. It will install TensorFlow.

```
pip install --upgrade --user tensorflow
```

Tensorboard

Allows us to visualize the graph, performance metrics, etc.



Tensorboard (http://edwardlib.org/images/tensorboard-histograms.png)

KMeans Walkthrough

- Source code is located at `/share/jproject/sakkas/tensorflow-tutorial`. Run the codes below to copy the codes to your home directory.

```
cd
cp -r /share/jproject/sakkas/tensorflow-tutorial .
cd tensorflow-tutorial
```

- The first part handles the arguments:
  - if 3 arguments are given, first one is number of points, second one is dimension and third one is number of cluster (centroids).
  - argv[0] is script name : kmeans.py

```
arg_len = len(sys.argv)
if arg_len < 3:
    #default values
    num_points = 1000
    dim = 2
    num_cluster = 10
else:
    num_points = int(sys.argv[1])
    dim = int(sys.argv[2])
    num_cluster = int(sys.argv[3])


points = np.random.uniform(0, 10, (num_points, dim))
```

- Points are generated by numpy.

```
points = np.random.uniform(0, 10, (num_points, dim))
# output:
# [[1.14229857, 1.24850134],
# [1.10465908, 3.526921  ],
# [2.26102466, 4.75565096],
# [0.49666799, 5.28391838]]
# two dimensional four points
```

- Placeholders holds place in the graph. Data is provided to graph in the session.

```
# we could use constant for points. But constants stored in the graph description
# points = tf.constant(np_points, dtype=tf.float32)

# None shape can take in any value while computation takes place
# [None, 2]
points_ph = tf.placeholder(dtype=tf.float32, name='points', shape=[None,dim])
```

- **Centroid variable**

```
#[num_cluster, dim]
centroids = tf.get_variable(name='centroids', shape=[num_cluster, dim],
        initializer = tf.initializers.random_uniform(minval=0, maxval=10.0,
seed=123))

# example initial centroids:
# [[0.4080963 , 2.0842123 ],
# [0.91802955, 7.0220065 ]]
```

- **Distance calculation**

  In KMeans, we need to calculate distances between points and centroids and select the closest one.

  Normally, we use loops to calculate distances. But in TensorFlow, we need different approach.

  - Dimension extension: We add extra dimension to points and centroids.

```
#[1, None, dim]
tf_expanded_points = tf.expand_dims(points_ph, 0, name='ex_points')

# shape is (1,4,2)
# [[[1.1422986  1.2485013 ]
#  [1.1046591  3.526921  ]
#  [2.2610247  4.755651  ]
#  [0.49666798 5.2839184 ]]]


#[num_cluster, 1, dim]
tf_expanded_centroids = tf.expand_dims(centroids, 1, name='ex_centroids')

# shape is (2, 1, 2)
# [[[0.4080963 , 2.0842123 ]],
# [[0.91802955, 7.0220065 ]]]
```

  - After dimension extension, when we use tf.substract, it calculates all distances.

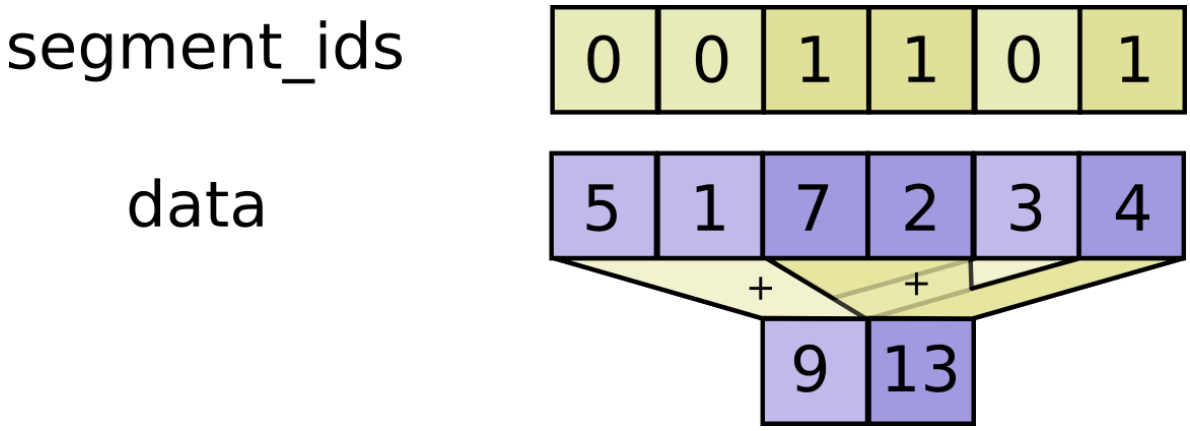
```
    with tf.name_scope("distance_calc"):
        #[num_cluster, None, dim]
        tf_substract =
tf.subtract(tf_expanded_points,tf_expanded_centroids,name='subtract')

# point_0: (1.1422986, 1.2485013), centr_0:(0.4080963 , 2.0842123), result:
(0.73420226, -0.83571096)

# shape is (2, 4, 2)
# [[[ 0.73420227 -0.835711  ]
# [ 0.69656277  1.4427087 ]
# [ 1.8529284   2.6714387 ]
# [ 0.08857167  3.199706  ]]
#
# [[ 0.22426903 -5.773505  ]
# [ 0.18662953 -3.4950855 ]
# [ 1.3429952  -2.2663555 ]
# [-0.42136157 -1.7380881 ]]]

        #[num_cluster, None, dim]
        tf_square=tf.square(tf_substract,name='square')
#
# [[[5.3905296e-01, 6.9841290e-01],
# [4.8519969e-01, 2.0814085e+00],
# [3.4333436e+00, 7.1365848e+00],
# [7.8449408e-03, 1.0238119e+01]],
# [[5.0296597e-02, 3.3333363e+01],
# [3.4830581e-02, 1.2215623e+01],
# [1.8036361e+00, 5.1363673e+00],
# [1.7754556e-01, 3.0209503e+00]]]

        #[num_cluster, None]
        tf_distances = tf.reduce_sum(tf_square, 2, name='red_sum')
#  (0.53905295 + 0.69841281) = 1.2374659 # distance between point0 and centroid0
# we could calculate square root but it wont affect the result
# [ 1.2374659,  2.5666082, 10.569928 , 10.245964 ],
# [33.38366  , 12.250454 ,  6.9400034,  3.1984959]]
```

- Assign point to the closest centroid:

```
#[None]
tf_assignments = tf.argmin(tf_distances, 0, name='asignments')
# [0, 0, 1, 1]
```

- Calculate new centroids
  - Unsorted segment sum

    We want to calculate the sum of the points assigned to the centroid.



Unsorted Segment Sum
(https://www.tensorflow.org/api_docs/python/tf/math/unsorted_segment_sum)

```
tf_sum = tf.unsorted_segment_sum(points, tf_assignments, num_cluster,
name='seg_sum')
# [ 2.24695765,  4.77542234],
# [ 2.75769265, 10.03956934]


tf_count = tf.unsorted_segment_sum(tf.ones_like(points), tf_assignments,
num_cluster, name='asgn_counts')

# [[2., 2.],
# [2., 2.]]


# calculate new centroids: sum/count
tf_new_centr = tf.cast(tf.div(tf_sum,tf_count, name='new_centroids'),
tf.float32)

# update operation
centroid_update = tf.assign(centroids, tf_new_centr)
```

- Tensorboard Operations

```
# total distance (we want to minimize distance between points and centroids)
tf_loss_op = tf.reduce_sum(tf.reduce_min(tf_distances, 0))

# scalar value
tf.summary.scalar('loss_summary', tf_loss_op)
tf_merged_summary = tf.summary.merge_all()
```

- Session

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

    # add graph to tensorboard
    writer = tf.summary.FileWriter('tensorboard_dir/', sess.graph)
    sess.run(tf.global_variables_initializer())

    for i in range(100):
        centr, merged_summary= sess.run([centroid_update, tf_merged_summary],
                                        feed_dict={points_ph: points})
        writer.add_summary(merged_summary, i)
```

Running KMeans example

```
# number of points 10000, points are 2 dimensional and 10 centroids
python kmeans.py 10000 2 10
```

Running Tensorboard:

- Run the code below to launch tensorboard

```
tensorboard --logdir=tensorboard_dir/
```

- **We need to access port 6006 from our laptop.**

  On Mac and Linux, open another terminal and login to your node with the code below.

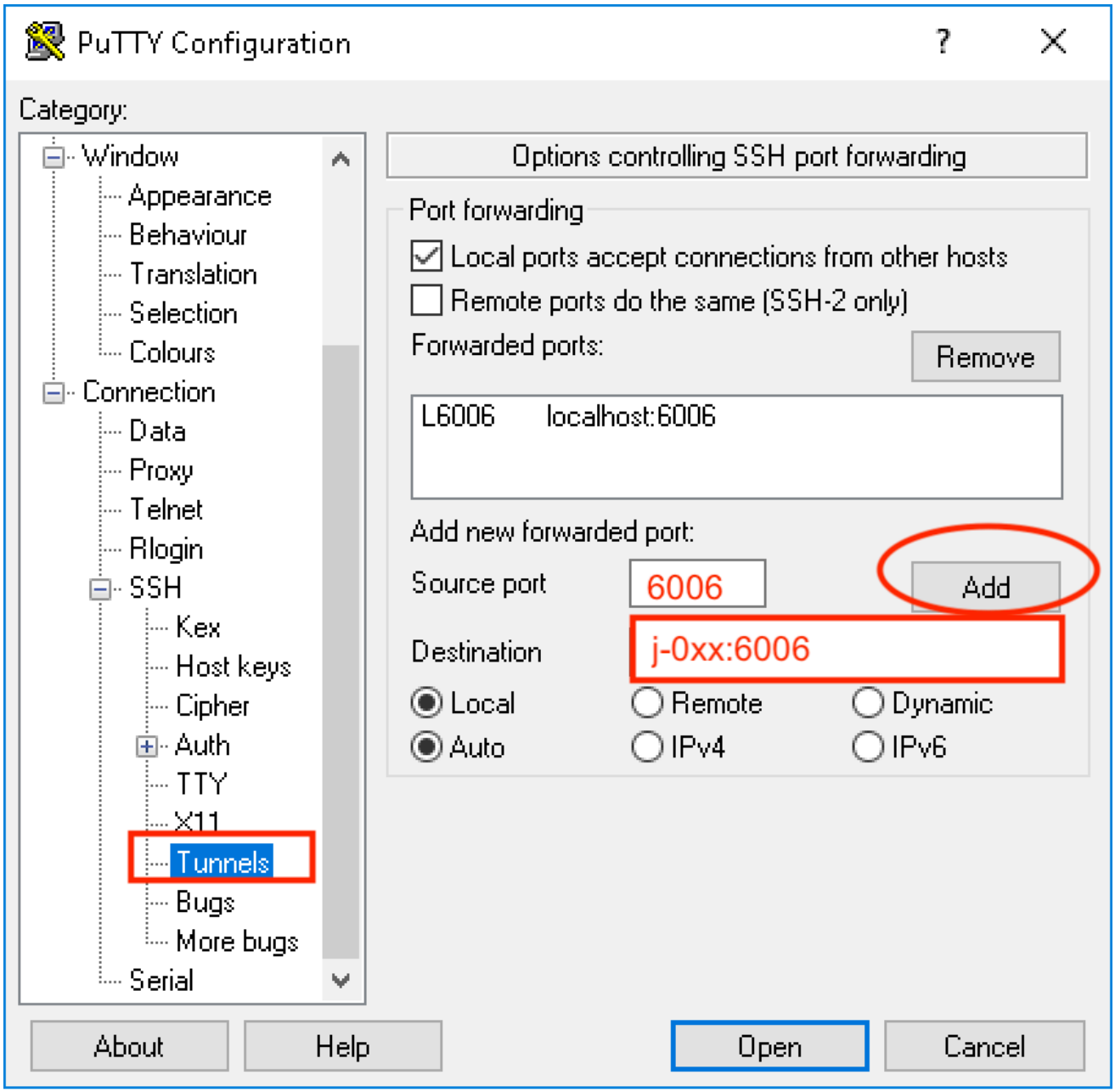
```
ssh -L 6006:j-<your-node>:6006 <user-name>@juliet.futuresystems.org
# example
# ssh -L 6006:j-029:6006 sakkas@juliet.futuresystems.org
```

On Windows (PuTTY), open another console and before login set the tunneling.

Source port: `6006`

Destination: `localhost:6006`

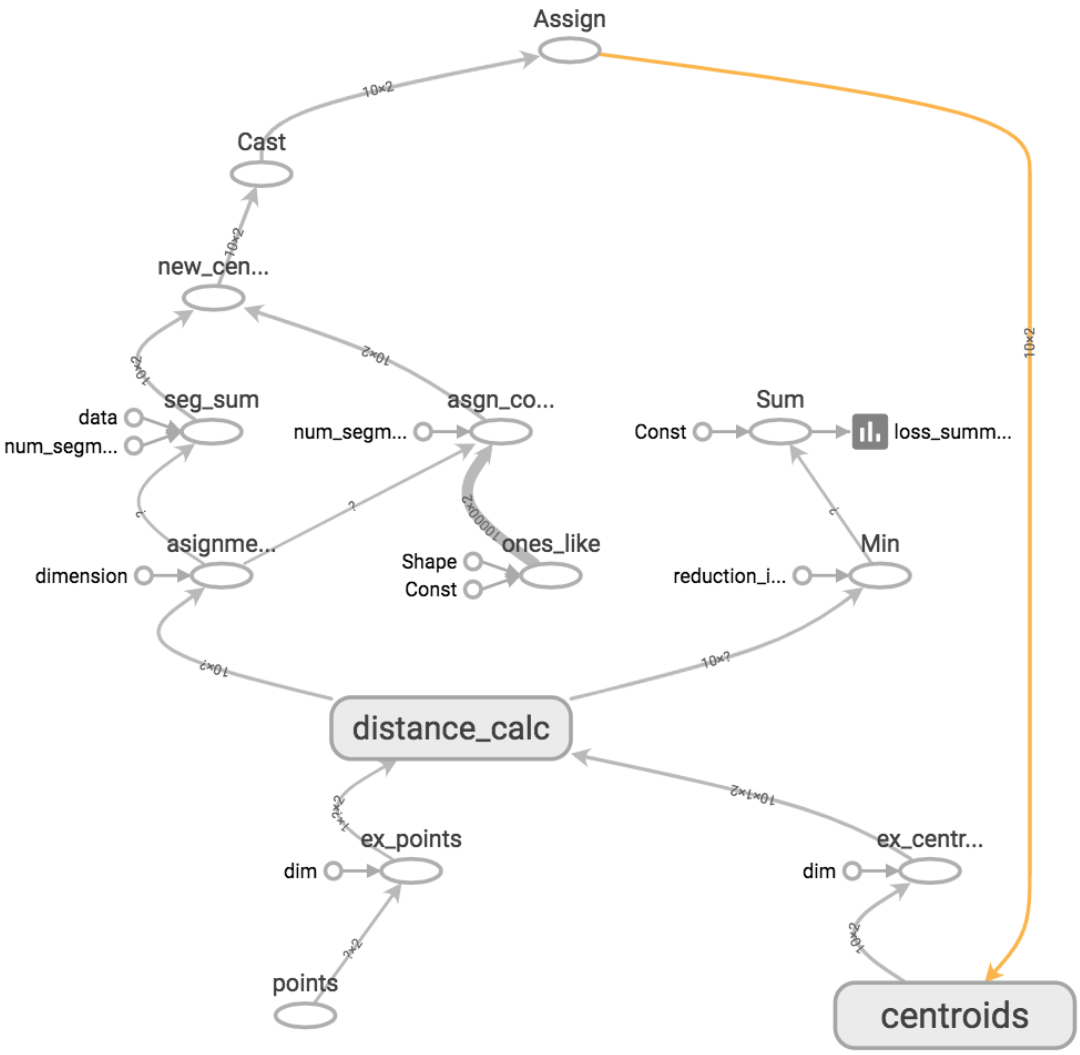
Don't forget to click `Add`



PuTTY Tunnel

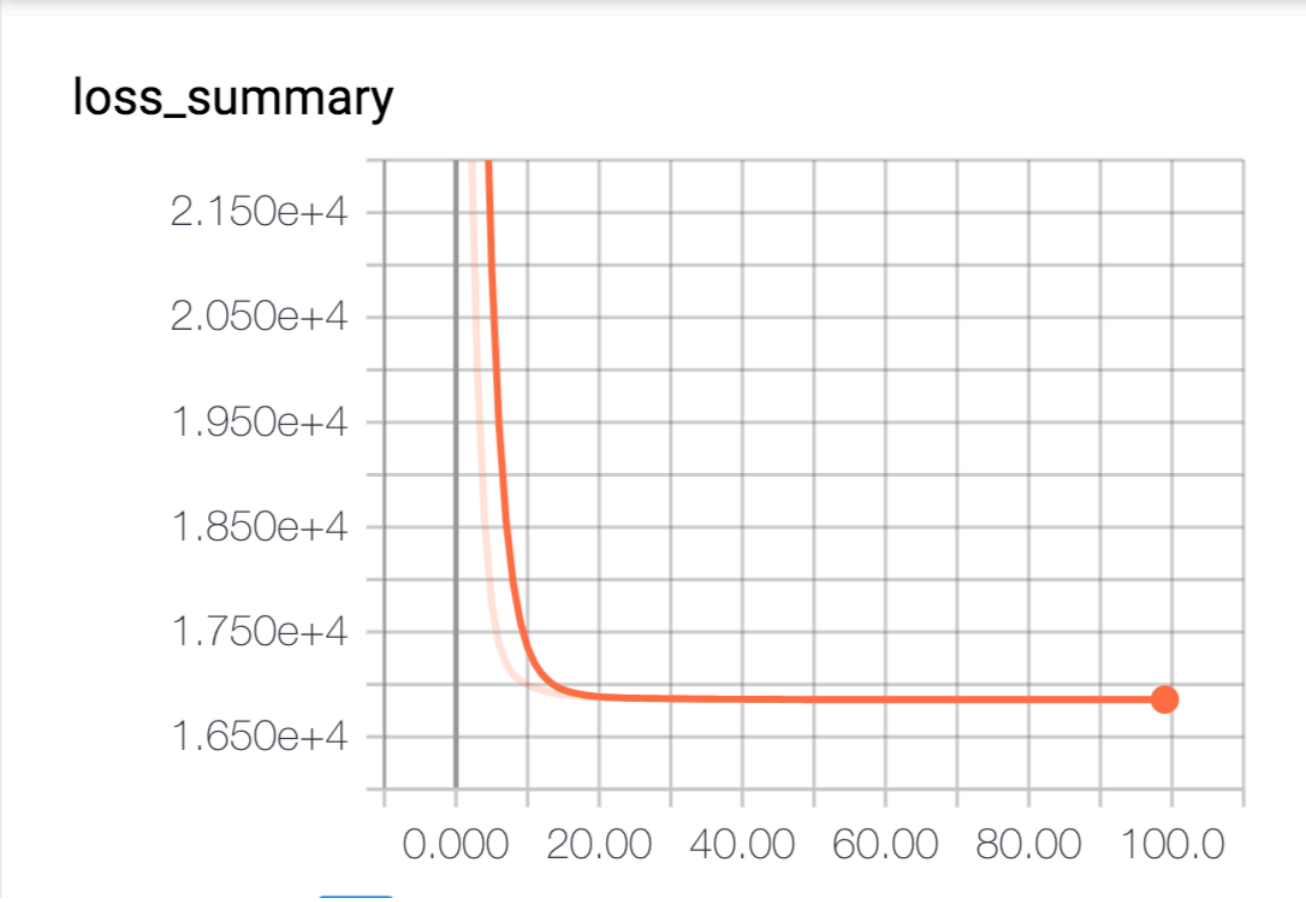
- Open your browser and open this page: `localhost:6006`

Tensorboard Graph



KMeans Graph

Loss Summary



Loss Graph

- As can be seen after 30 iterations there is no improvement. If you want you can add a early stop criteria. But this is optional. There is no submission for this.