Lecture 11: Introduction to TensorFlow I

In [2]:

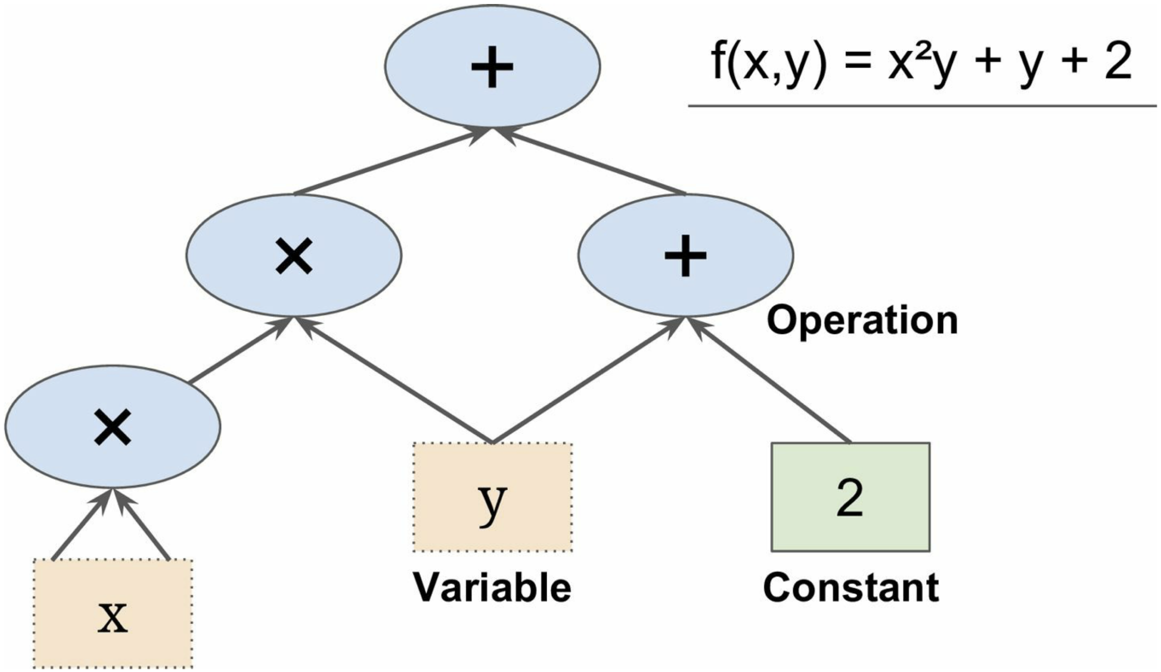
```python
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals

# Common imports
import numpy as np
import os

# To make this notebook's output stable across runs
def reset_graph(seed=42):
    tf.reset_default_graph()
    tf.set_random_seed(seed)
    np.random.seed(seed)

# To plot pretty figures
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
```

Overview of TensorFlow

[TensorFlow (https://www.tensorflow.org/)](https://www.tensorflow.org/) is an open source library developed by Google for numerical computation. It is particularly well suited for large-scale machine learning.

Based on the construction of *computational graphs*.

Computational graphs



[Credit: Geron]

User constructs the computational graph (can be constructed in Python).

TensorFlow takes computational graph and runs it efficiently via optimized C++ code.

Parallel and distributed computation

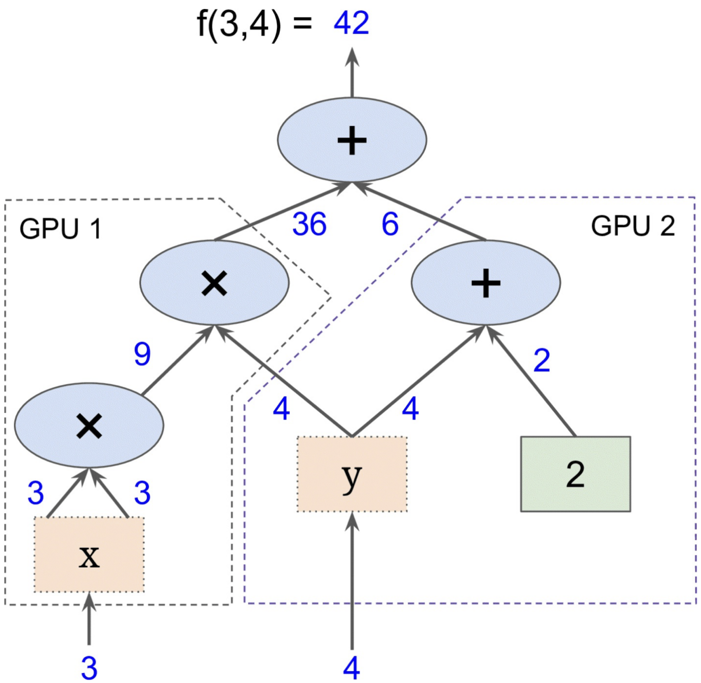


Figure 9-2. Parallel computation on multiple CPUs/GPUs/servers

[Credit: Geron]

Computational graphs can be broken up into different chunks, which are then run in parallel across many CPUs and/or GPUs (or highly distributed systems).

This approach allows TensorFlow to scale to big-data.

Scaling to big-data

For example, TensorFlow can be used to train neural networks with millions of parameters and training sets with billions of training instances.

Provides the infrastructure behind many of Google's large-scale machine learning products, e.g. Google Search, Google Photos, ...

Basics of computational graphs

Variables

In [3]:
```
import tensorflow as tf
reset_graph()
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

```
/Users/mcewen/anaconda3/envs/tensorflow_py35/lib/python3.5/site-packages/h5py/
__init__.py:36: FutureWarning: Conversion of the second argument of issubdtype
from `float` to `np.floating` is deprecated. In future, it will be treated as
`np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

In [4]:
```
f
```

Out[4]:
```
<tf.Tensor 'add_1:0' shape=() dtype=int32>
```

Variables

In [3]:
```
import tensorflow as tf
reset_graph()
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

```
/Users/mcewen/anaconda3/envs/tensorflow_py35/lib/python3.5/site-packages/h5py/
__init__.py:36: FutureWarning: Conversion of the second argument of issubdtype
from `float` to `np.floating` is deprecated. In future, it will be treated as
`np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

In [4]:
```
f
```

Out[4]:
```
<tf.Tensor 'add_1:0' shape=() dtype=int32>
```

This does not perform any computation.

We have just set up the computational graph.

Execution

Need to open a TensorFlow session, initialise variables, and run to evaluate the computational graph:

In [5]:

```
sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)
print(result)
sess.close()
```

```
42
```

Execution

Need to open a TensorFlow session, initialise variables, and run to evaluate the computational graph:

In [5]:
```
sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)
print(result)
sess.close()
```

```
42
```

Can avoid repeated `sess.run` calls as follows, where the session is set as the default session:

In [6]:
```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()

print(result)
```

```
42
```

Can also initialise all variables at once:

In [7]:
```
init = tf.global_variables_initializer()
```

Note that this does not perform initialisation immediately but rather sets up a node to perform initialisation when it is run.

In [8]:
```
with tf.Session() as sess:
    init.run()
    result = f.eval()
print(result)
```

```
42
```

Interactive sessions automatically sets themselves as the default session.

In [9]:

```
sess = tf.InteractiveSession()
init.run()
result = f.eval()
print(result)
sess.close()
```

```
42
```

Managing computational graphs

Any created node is added to default graph.

In [10]:
```
reset_graph()
x1 = tf.Variable(1)
x1.graph is tf.get_default_graph()
```

Out[10]:
```
True
```

Can manage multiple graphs by setting different graphs as the default graph inside a `with` block:

In [11]:
```
graph = tf.Graph()
with graph.as_default():
    x2 = tf.Variable(2)

x2.graph is graph, x2.graph is tf.get_default_graph()
```

Out[11]:
```
(True, False)
```

Lifecycle of a Node value

When create a node TensorFlow automatically determines dependencies and evaluates those first.

In [12]:
```
w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3

with tf.Session() as sess:
    print(y.eval())
    print(z.eval())
```

```
10
15
```

Lifecycle of a Node value

When create a node TensorFlow automatically determines dependencies and evaluates those first.

In [12]:

```
w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3

with tf.Session() as sess:
    print(y.eval())
    print(z.eval())
```

```
10
15
```

In the above `x` is *not* reused when evaluating `y` and `z`, i.e. `x` is evaluated twice.

Can evaluate multiple nodes in single graph run:

In [13]:
```
with tf.Session() as sess:
    y_val, z_val = sess.run([y,z])
    print(y_val)
    print(z_val)
```

```
10
15
```

Variable lifetime

The values of `Variables` are retained across graph runs.

All other node values are dropped between graph runs.

Linear regression with TensorFlow

In [14]:

```
import numpy as np
from sklearn.datasets import fetch_california_housing
reset_graph()

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
housing_data_target = housing.target.reshape(-1, 1)
```

In [15]:

```
housing.feature_names
```

Out[15]:

```
['MedInc',
 'HouseAge',
 'AveRooms',
 'AveBedrms',
 'Population',
 'AveOccup',
 'Latitude',
 'Longitude']
```

In [16]:

```
m, n, housing_data_plus_bias.shape, housing_data_target.shape
```

Out[16]:

```
(20640, 8, (20640, 9), (20640, 1))
```

In [17]:
```
X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

In [18]:
```
theta_value
```

Out[18]:
```
array([[-3.7465141e+01],
       [ 4.3573415e-01],
       [ 9.3382923e-03],
       [-1.0662201e-01],
       [ 6.4410698e-01],
       [-4.2513184e-06],
       [-3.7732250e-03],
       [-4.2664889e-01],
       [-4.4051403e-01]], dtype=float32)
```

Exercise: compute with numpy.

Exercise: compute with numpy.

In [19]:

```
X = housing_data_plus_bias
y = housing_data_target
theta_numpy = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
print(theta_numpy)
```

```
[[-3.69419202e+01]
 [ 4.36693293e-01]
 [ 9.43577803e-03]
 [-1.07322041e-01]
 [ 6.45065694e-01]
 [-3.97638942e-06]
 [-3.78654265e-03]
 [-4.21314378e-01]
 [-4.34513755e-01]]
```

Exercise: compute with SciKitLearn.

Exercise: compute with SciKitLearn.

```
In [20]: from sklearn.linear_model import LinearRegression
         lin_reg = LinearRegression()
         lin_reg.fit(housing.data, housing_data_target)
         print(np.r_[lin_reg.intercept_.reshape(-1, 1), lin_reg.coef_.T])
```

```
[[-3.69419202e+01]
 [ 4.36693293e-01]
 [ 9.43577803e-03]
 [-1.07322041e-01]
 [ 6.45065694e-01]
 [-3.97638942e-06]
 [-3.78654265e-03]
 [-4.21314378e-01]
 [-4.34513755e-01]]
```

Advantage of computing by TensorFlow is that computations can be deployed on various computational infrastructure (e.g. GPUs) and scaled to big-data.

Gradients

Training using gradient descent.

In [21]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]
```

Manually computing the gradients

Set up computational graph

In [22]:
```
reset_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="thet
a")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

Evaluate

In [23]:

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()
```

```
Epoch 0 MSE = 9.161543
Epoch 100 MSE = 0.7145007
Epoch 200 MSE = 0.5667047
Epoch 300 MSE = 0.5555716
Epoch 400 MSE = 0.5488116
Epoch 500 MSE = 0.54363626
Epoch 600 MSE = 0.53962916
Epoch 700 MSE = 0.53650916
Epoch 800 MSE = 0.5340678
Epoch 900 MSE = 0.53214705
```

In [24]:
```
best_theta
```

Out[24]:
```
array([[ 2.0685525 ],
       [ 0.8874027 ],
       [ 0.14401658],
       [-0.34770882],
       [ 0.36178368],
       [ 0.00393812],
       [-0.04269557],
       [-0.6614528 ],
       [-0.63752776]], dtype=float32)
```

Using autodiff

Autodiff builds derivatives of each stage of the computational graph so that gradients can be computed automatically and efficiently.

Set up computational graph

In [25]:

```
reset_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="thet
a")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
```

Set up computational graph

In [25]:

```
reset_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="thet
a")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
```

In [26]:

```
gradients = tf.gradients(mse, [theta])[0]  # gradient of MSE w.r.t. theta
```

Set up computational graph

In [25]:
```
reset_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="thet
a")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
```

In [26]:
```
gradients = tf.gradients(mse, [theta])[0]  # gradient of MSE w.r.t. theta
```

In [27]:
```
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

Evaluate

In [28]:

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()
```

```
Epoch 0 MSE = 9.161543
Epoch 100 MSE = 0.7145006
Epoch 200 MSE = 0.56670463
Epoch 300 MSE = 0.5555716
Epoch 400 MSE = 0.5488117
Epoch 500 MSE = 0.5436362
Epoch 600 MSE = 0.53962916
Epoch 700 MSE = 0.53650916
Epoch 800 MSE = 0.5340678
Epoch 900 MSE = 0.53214717
```

In [29]:

```
print("Best theta:")
print(best_theta)
```

```
Best theta:
[[ 2.0685525 ]
 [ 0.8874027 ]
 [ 0.14401658]
 [-0.34770882]
 [ 0.36178368]
 [ 0.00393811]
 [-0.04269556]
 [-0.6614528 ]
 [-0.6375277 ]]
```

Exercise: compute gradients.

Compute the partial derivatives of the following function (`my_func`) at `(a,b) = (0.2,0.3)` by:

1. numerical integration
2. autodiff in TensorFlow

In [30]:
```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

In [31]:
```
my_func(0.2, 0.3)
```

Out[31]:
```
-0.21253923284754916
```

In [32]:

```
delta = 0.01
f = my_func
df_da = (f(0.2+delta,0.3) - f(0.2-delta,0.3))/(2*delta)
df_db = (f(0.2,0.3+delta) - f(0.2,0.3-delta))/(2*delta)
df_da, df_db
```

Out[32]:

```
(-1.1383901861704486, 0.19675140591134677)
```

In [33]:

```
reset_graph()

a = tf.Variable(0.2, name="a")
b = tf.Variable(0.3, name="b")
z = tf.constant(0.0, name="z0")
for i in range(100):
    z = a * tf.cos(z + i) + z * tf.sin(b - i)
```

In [33]:

```
reset_graph()

a = tf.Variable(0.2, name="a")
b = tf.Variable(0.3, name="b")
z = tf.constant(0.0, name="z0")
for i in range(100):
    z = a * tf.cos(z + i) + z * tf.sin(b - i)
```

In [34]:

```
grads = tf.gradients(z, [a, b])
init = tf.global_variables_initializer()
```

In [33]:
```
reset_graph()

a = tf.Variable(0.2, name="a")
b = tf.Variable(0.3, name="b")
z = tf.constant(0.0, name="z0")
for i in range(100):
    z = a * tf.cos(z + i) + z * tf.sin(b - i)
```

In [34]:
```
grads = tf.gradients(z, [a, b])
init = tf.global_variables_initializer()
```

In [35]:
```
with tf.Session() as sess:
    init.run()
    print(z.eval())
    print(sess.run(grads))
```

```
-0.21253741
[-1.1388494, 0.19671395]
```

Using an optimizer

So far we have implemented optimizers by hand but TensorFlow provides many built-in optimizers.

Set up computational graph

In [36]:

```
reset_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="thet
a")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
```

Set up computational graph

In [36]:
```
reset_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="thet
a")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
```

Set up optimizer

In [37]:
```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

Evaluate

In [38]:

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()
```

```
Epoch 0 MSE = 9.161543
Epoch 100 MSE = 0.7145006
Epoch 200 MSE = 0.56670463
Epoch 300 MSE = 0.5555716
Epoch 400 MSE = 0.5488117
Epoch 500 MSE = 0.5436362
Epoch 600 MSE = 0.53962916
Epoch 700 MSE = 0.53650916
Epoch 800 MSE = 0.5340678
Epoch 900 MSE = 0.53214717
```

In [39]:

```
print("Best theta:")
print(best_theta)
```

```
Best theta:
[[ 2.0685525 ]
 [ 0.8874027 ]
 [ 0.14401658]
 [-0.34770882]
 [ 0.36178368]
 [ 0.00393811]
 [-0.04269556]
 [-0.6614528 ]
 [-0.6375277 ]]
```

Feeding data

Previously data used in each graph execution was static.

For stochastic algorithms we need to consider a different subset of the data for each execution.

This can be achieved with *placeholder* nodes (rather than *constant* nodes), where data is fed by a dictionary.

In [40]:
```
reset_graph()

A = tf.placeholder(tf.float32, shape=(None, 3))
B = A + 5
with tf.Session() as sess:
    B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
    B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
```

In [41]:
```
print(B_val_1)
```

```
[[6. 7. 8.]]
```

In [42]:
```
print(B_val_2)
```

```
[[ 9. 10. 11.]
 [12. 13. 14.]]
```

In [43]:
```
feed_dict={A: [[1, 2, 3]]}
type(feed_dict)
```

Out[43]:
```
dict
```

Mini-batch gradient descent

Extend previous gradient descent example to mini-batch gradient descent.

In [44]:
```
reset_graph()

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

In [45]:
```
learning_rate = 0.01
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="thet
a")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()
```

In [46]:
```
n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

In [47]:
```
def fetch_batch(epoch, batch_index, batch_size):
    np.random.seed(epoch * n_batches + batch_index)
    indices = np.random.randint(m, size=batch_size)
    X_batch = scaled_housing_data_plus_bias[indices]
    y_batch = housing_data_target[indices]
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()
```

In [48]:
```
best_theta
```

Out[48]:
```
array([[ 2.0703337 ],
       [ 0.8637145 ],
       [ 0.12255151],
       [-0.31211874],
       [ 0.38510373],
       [ 0.00434168],
       [-0.01232954],
       [-0.83376896],
       [-0.8030471 ]], dtype=float32)
```