Crash Course on TensorFlow™

Vincent Lepetit

TensorFlow

Created by Google for easily implementing Deep Networks;

Library for Python, but can also be used with C and Java;

Exists for Linux, Mac OSX, Windows;

Official documentation: `https://www.tensorflow.org`

Why 'Tensor', Why 'Flow'?

A tensor can be

a scalar: `3`, rank 0, shape `[]`

a vector: `[1., 2., 3.]`, rank 1, shape `[3]`

a matrix, `[[1, 2, 3], [4, 5, 6]]`, rank 2, shape `[2, 3]`

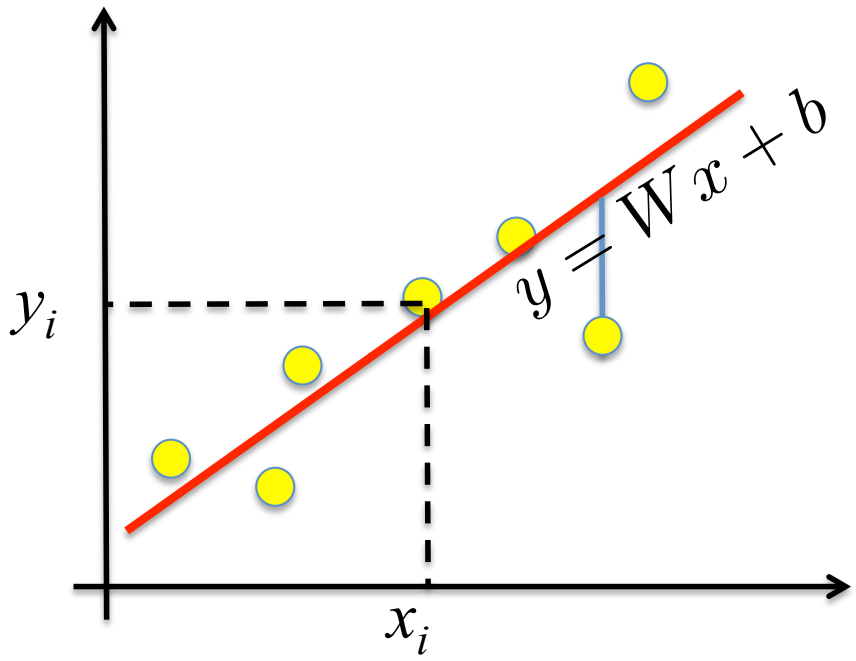
their extension to more dimensions:

`[[[1, 2, 3], [[7, 8, 9]]]`, rank 3, shape `[2, 1, 3]`

Computations in TensorFlow are defined using a graph of operations applied to tensors.

First Full Example:
Linear Regression
(from the official documentation)

The Math Problem We Will Solve Linear regression:



We want to fit a linear model to some data.

Formally, this means that we want to estimate the parameters W and b of the model:

$$y = Wx + b$$

W and b are scalar. We will estimate them by minimizing:

$$\text{loss} = \sum_i (Wx_i + b - y_i)^2$$

where the (x_i, y_i) are training data.

Gradient Descent

$$\text{loss}(W, b) = \sum_i (W x_i + b - y_i)^2$$

$$(\hat{W}, \hat{b}) = \arg\min \ \text{loss}(W, b)$$

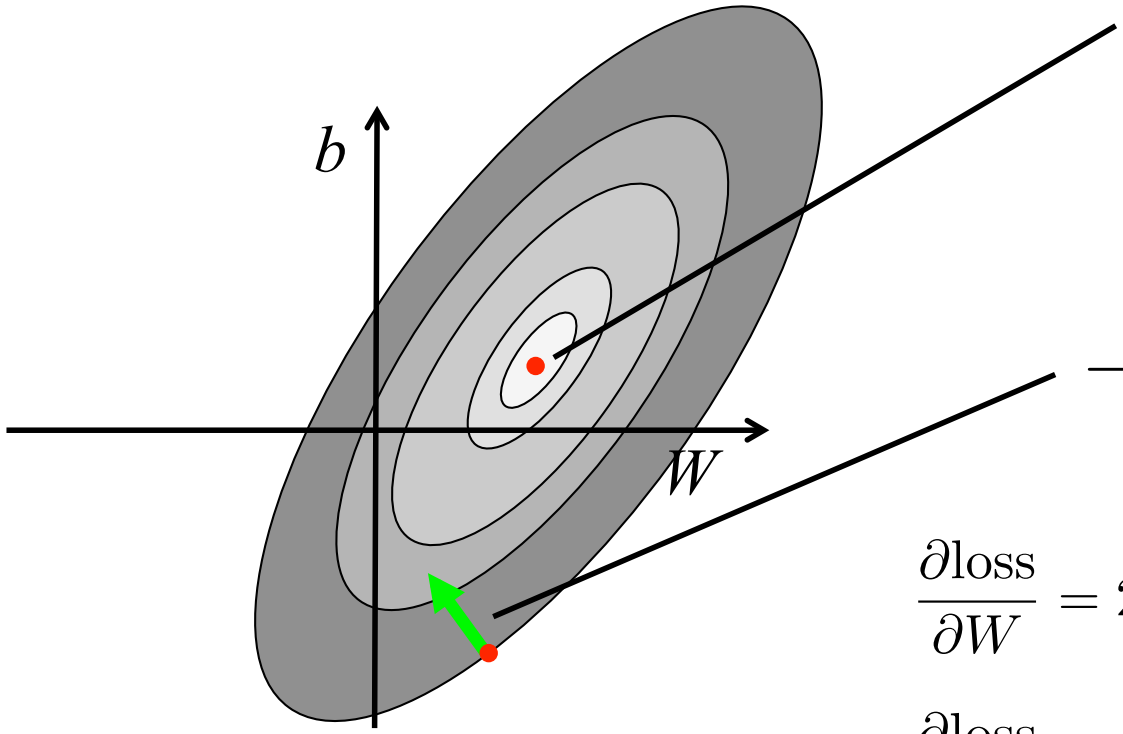
Linear regression can be solved using linear algebra (at least when the problem is small).

Here we will use gradient descent as this will be a simple example to start with TensorFlow.

Gradient Descent

$$\text{loss}(W, b) = \sum_i (W x_i + b - y_i)^2$$

$$(\hat{W}, \hat{b}) = \arg\min \ \text{loss}(W, b)$$



$(\hat{W}, \hat{b})$

$$- \begin{bmatrix} \frac{\partial \text{loss}}{\partial W} \\ \frac{\partial \text{loss}}{\partial b} \end{bmatrix}$$

$$\frac{\partial \text{loss}}{\partial W} = 2 \sum_i x_i (W x_i + b - y_i)$$

$$\frac{\partial \text{loss}}{\partial b} = 2 \sum_i (W x_i + b - y_i)$$

```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

```
import numpy as np
import tensorflow as tf
```

`tf` will stand for TensorFlow

```
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
x = tf.placeholder(tf.float32)
linear_model =
y = tf.placeho
# loss
loss = tf.redu
# optimizer
optimizer = tf
train = optimi
# training dat
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

Our unknowns.

They are `tf.Variable`

We need to provide their initial values and types.

TensorFlow Graph Element

Can be:

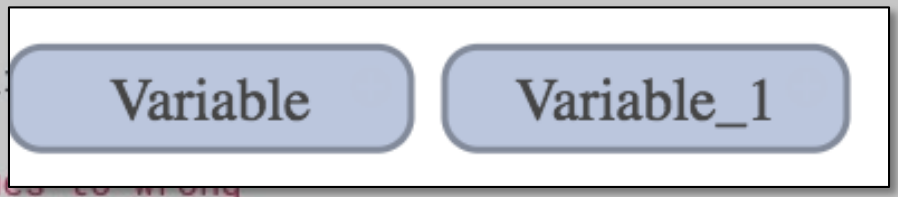
- A tensor (`tf.Tensor`);
- An operation: add, mul, etc. (`tf.Operation`);
- A variable (`tf.Variable`, which is in fact made of a `tf.Operation` (`assign`) and a `tf.Tensor`);
- and other things.

```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
x = tf.placeholder(tf.float32)
linear_model =
y = tf.placeho
# loss
loss = tf.redu
# optimizer
optimizer = tf
train = optimi
# training dat
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_
sess = tf.Session()
sess.run(init) # reset value
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

Our unknowns.

They are `tf.Variable`

We need to provide their initial values and types



```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_s
# optimizer
optimizer = tf.tra
train = optimizer.
# training data
x_train = [1,2,3,4
y_train = [0,-1,-2
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # r
for i in range(100
  sess.run(train,

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

The input.

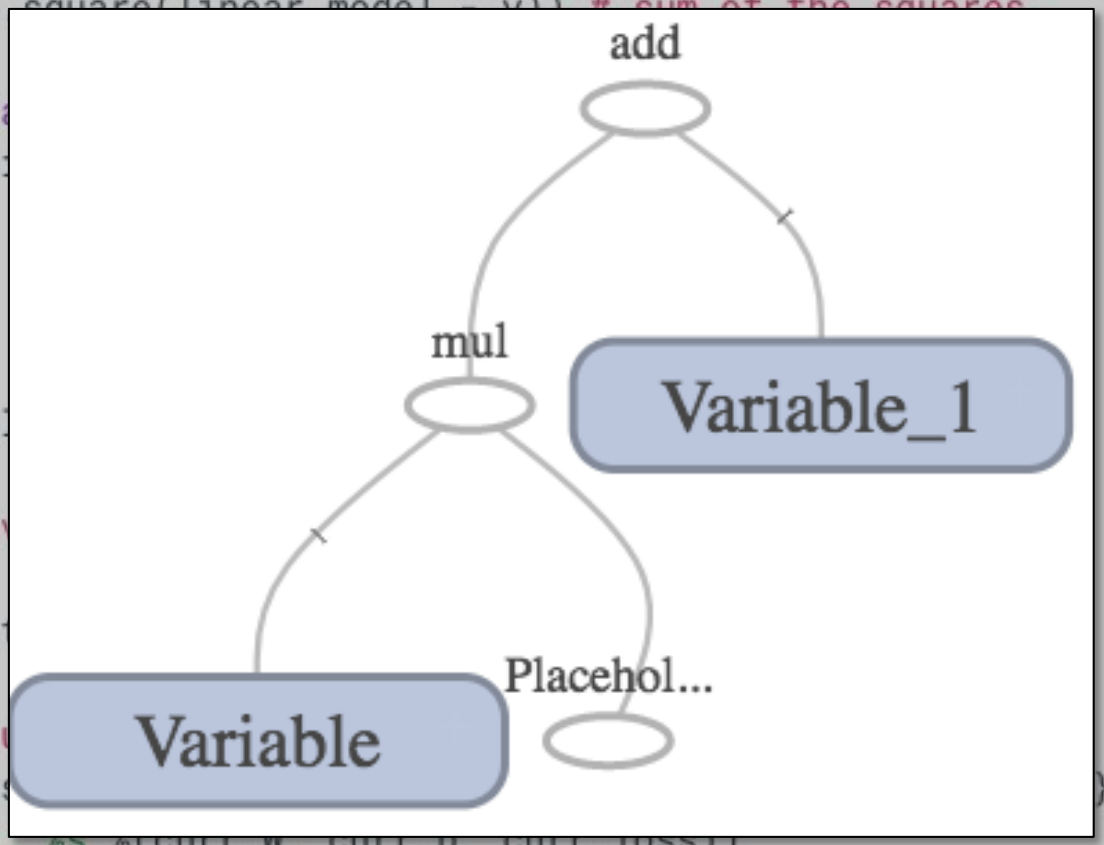
It is a `tf.placeholder`

This will be useful to tell TensorFlow that the input is 1 float when we define the loss function.

```
import numpy as np
import tensorflo

# Model paramete
W = tf.Variable(
b = tf.Variable(
# Model input ar
x = tf.placeholder(tf.float32)

linear_model = W * x + b

# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.Gra
train = optimizer.minim
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variabl
sess = tf.Session()
sess.run(init) # reset
for i in range(1000):
  sess.run(train, {x:x_

# evaluate training acc
curr_W, curr_b, curr_lo
print("W: %s b: %s loss:
```

`linear_model` is a `tf.Operation`

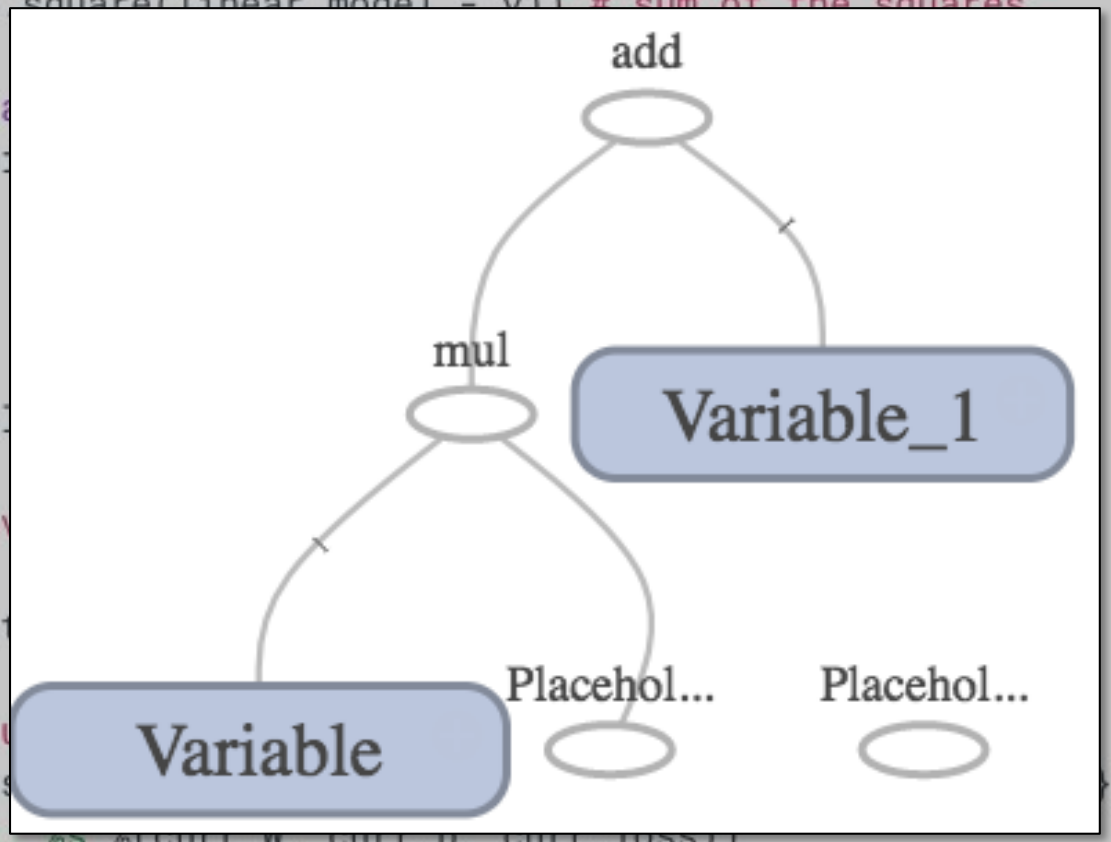
It is the *predicted* output, and will be useful to define the loss function.



```
import numpy as np
import tensorflow as tf

# Mod
W = t
b = t
# Mod
x = t
linear_model = W * x + b
y = tf.placeholder(tf.float32)
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.Gra
train = optimizer.minim
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variab
sess = tf.Session()
sess.run(init) # reset
for i in range(1000):
  sess.run(train, {x:x_
# evaluate training acc
curr_W, curr_b, curr_los
print("W: %s b: %s loss:
```

This is a `tf.placeholder` for the *expected* output.

It will be useful to define the loss function.



The loss function: $\text{loss} = \sum_i (Wx_i + b - y_i)^2$

Note that we cannot write for example:

```
(linear_model - y) ** 2
```

we have to write:

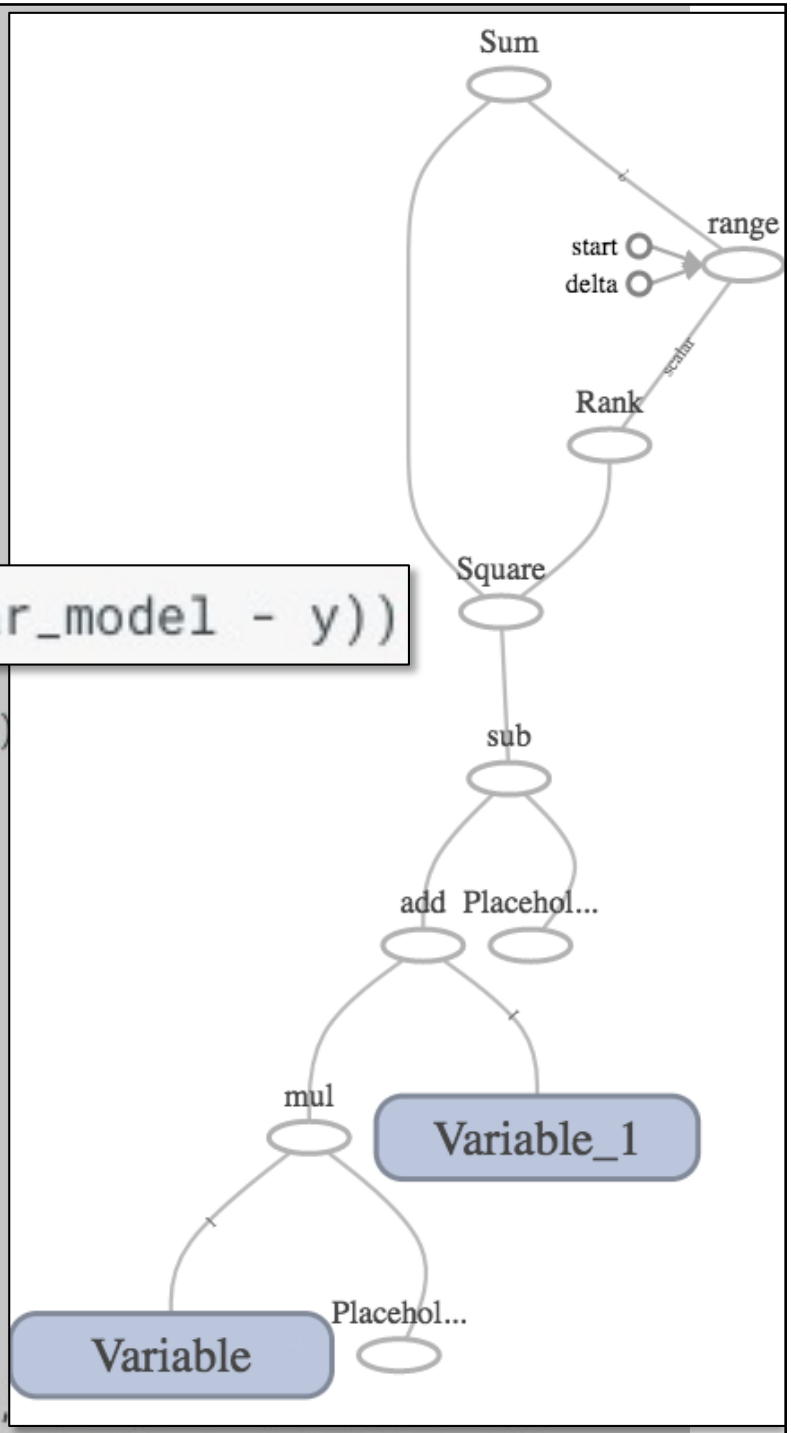
```
tf.square(linear_model - y)
```

```
import numpy
import tensc

# Model para
W = tf.Varia
b = tf.Varia
# Model inpu
x = tf.place
linear_model
y = tf.place

loss = tf.reduce_sum(tf.square(linear_model - y))
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
```

```python
loss = tf.reduce_sum(tf.square(linear_model - y))
```

```python
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss],
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```



```
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
#
x
y
#
i
s
s
f
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

Creates an `optimizer` object.

It implements gradient descent.

0.01 is the step size.

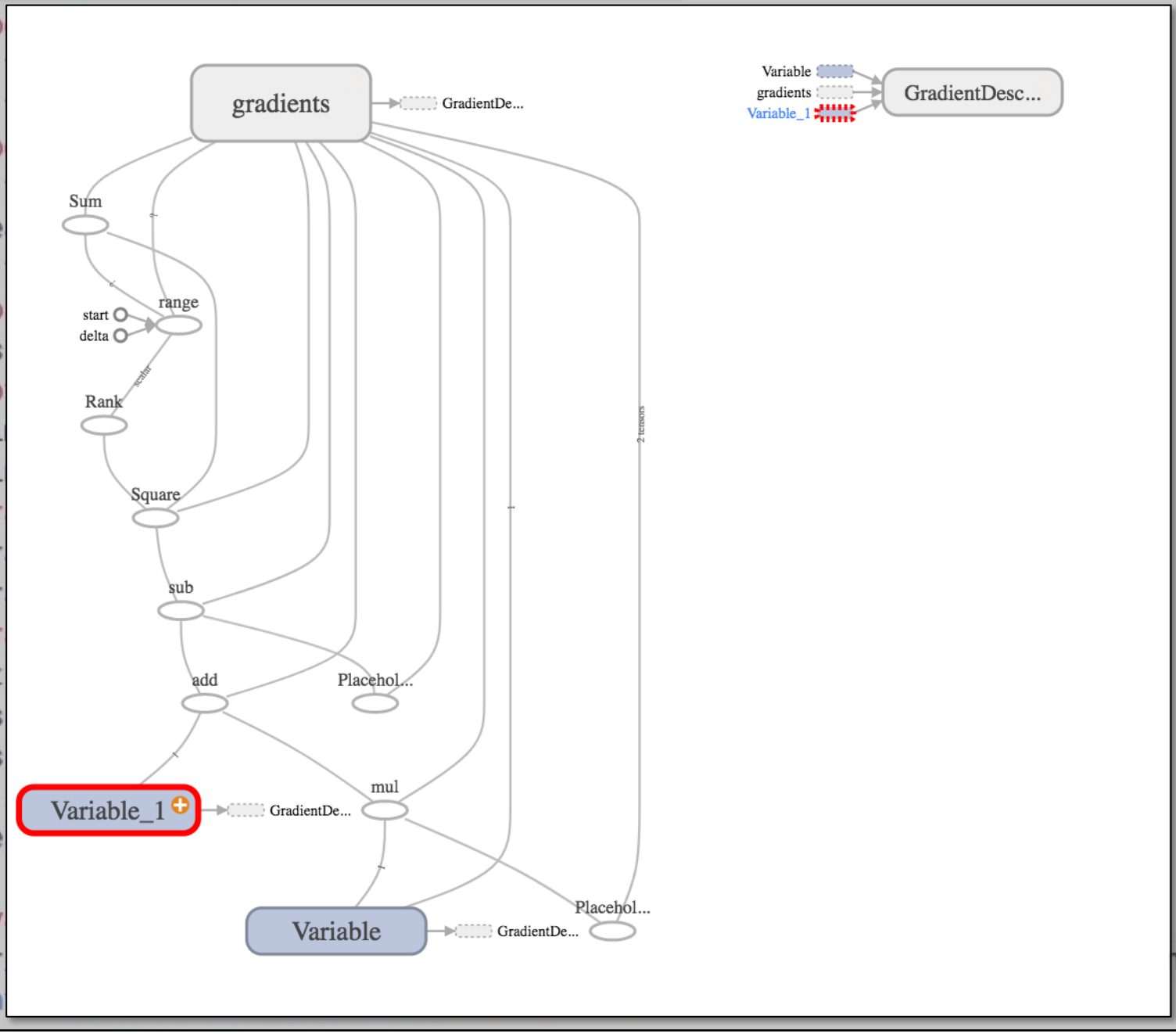
```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

```
train = optimizer.minimize(loss)
```

Create an object that will be used to perform the minimization.

Still no optimization is actually ran.

```
import numpy as np
train = optimizer.minimize(loss)
# Mo
W =
b =
# Mo
x =
line
y =
# lo
loss
# op
opti
trai
# tr
x_tr
y_tr
# tr
init
sess
sess
for
  se

# ev
curr
prin
                                                                    n})
```



```
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
```

```
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
```

```
0.01)
# training data
x_train = [1,2,3,4]
y_t
# t
ini
ses
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

These are our training data: (1,0), (2, -1), (3, -2), (4, -3)

They are regular Python arrays.

```
import numpy as np
import tensorflow as tf

# Model parameters
```

`init` is a handle to the TensorFlow sub-graph that initializes all the global variables.

`sess` is an object on which we will call the `run()` function.

Still nothing, until

`sess.run(init)`

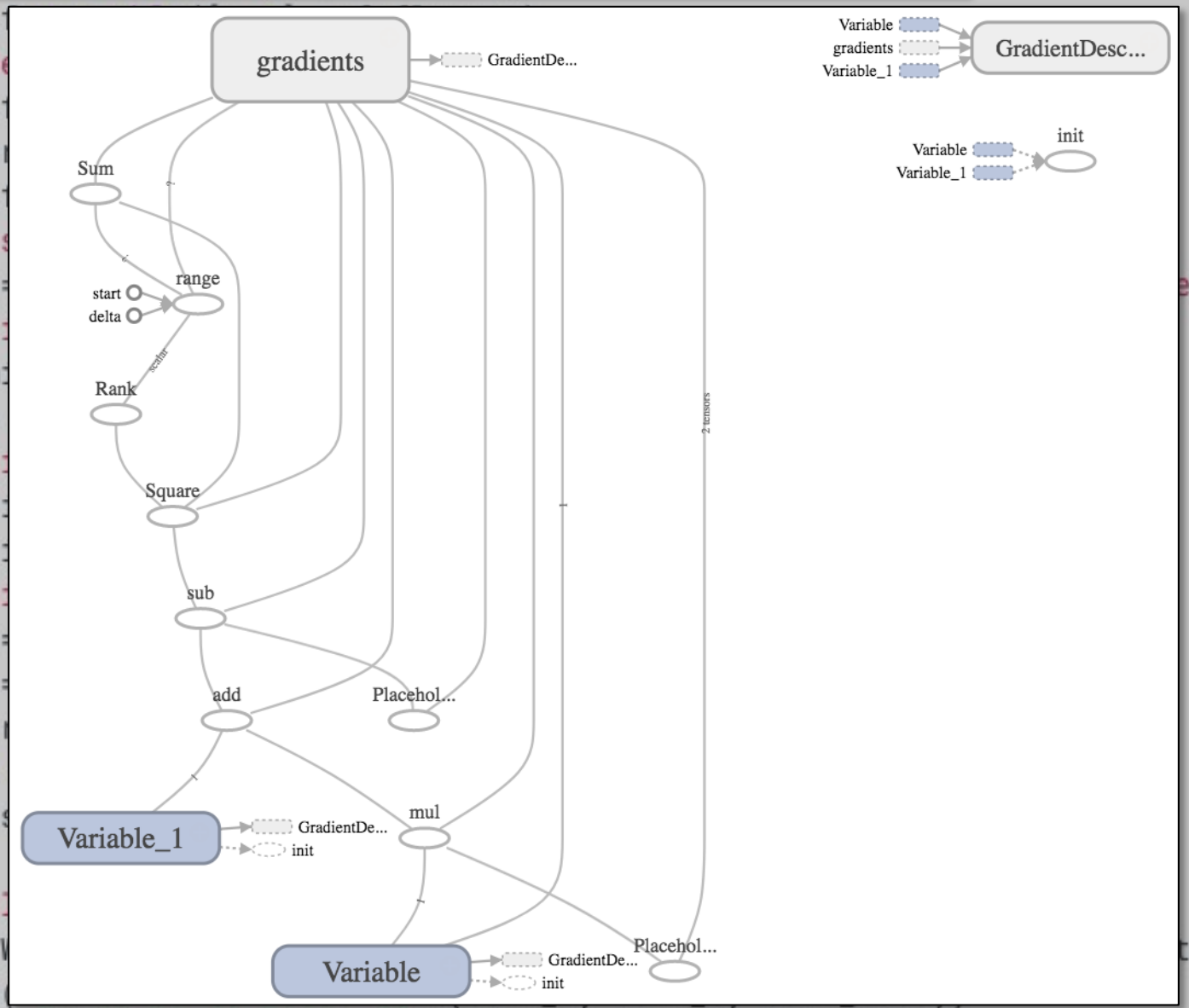
is called, which initializes $\mathbf{W}$ and $\mathbf{b}$.

```
y_train = [0,-1,-2,-3]
```

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

```
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```



```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
```

Does 1000 steps of gradient descent.

`{x:x_train, y:y_train}` is a regular Python dictionary of tensors, created from the `x_train` and `y_train` Python arrays.

It associates a value for x_i to a value for y_i.

`sess.run(train, {x:x_train, y:y_train})` applies the `train` handle to this data.

```python
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

the `tensorflow.Session.run` function and the TensorFlow graph

```
v = session.run(
                 fetches,
                 feed_dict=None,
                 options=None,
                 run_metadata=None
               )
```

`fetches` is a TensorFlow graph element (or a tuple, list, etc. of graph elements);

`feed_dict` contains the input and expected data used to compute the values of the elements in `fetches`;

The return values are the values of the elements in `fetches`, given the data in `feed_dict`

See example next slide:

```
curr_W, curr_b, curr_loss = sess.run([W, b, loss],
                                     {x:x_train, y:y_train})
```

```
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
```

Evaluate `W`, `b`, and `loss`, given `x` in `x_train` and `y` in `y_train`.

```
curr_W, curr_b, curr_loss = sess.run([W, b, loss],
                                     {x:x_train, y:y_train})
```

```
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

```python
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W*x+b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

Remark:

Because linear regression is very common, there is already an object for it. See:

`tf.contrib.learn.LinearRegressor`

```
import numpy as np
import tensorflow as tf
```

We can give better names to the graph's nodes:

```
with tf.variable_scope("W"):
  W = tf.Variable([.3], tf.float32)
with tf.variable_scope("b"):
  h = tf.Variable([-.3], tf.float32)
with tf.variable_scope("input"):
  x = tf.placeholder(tf.float)
```

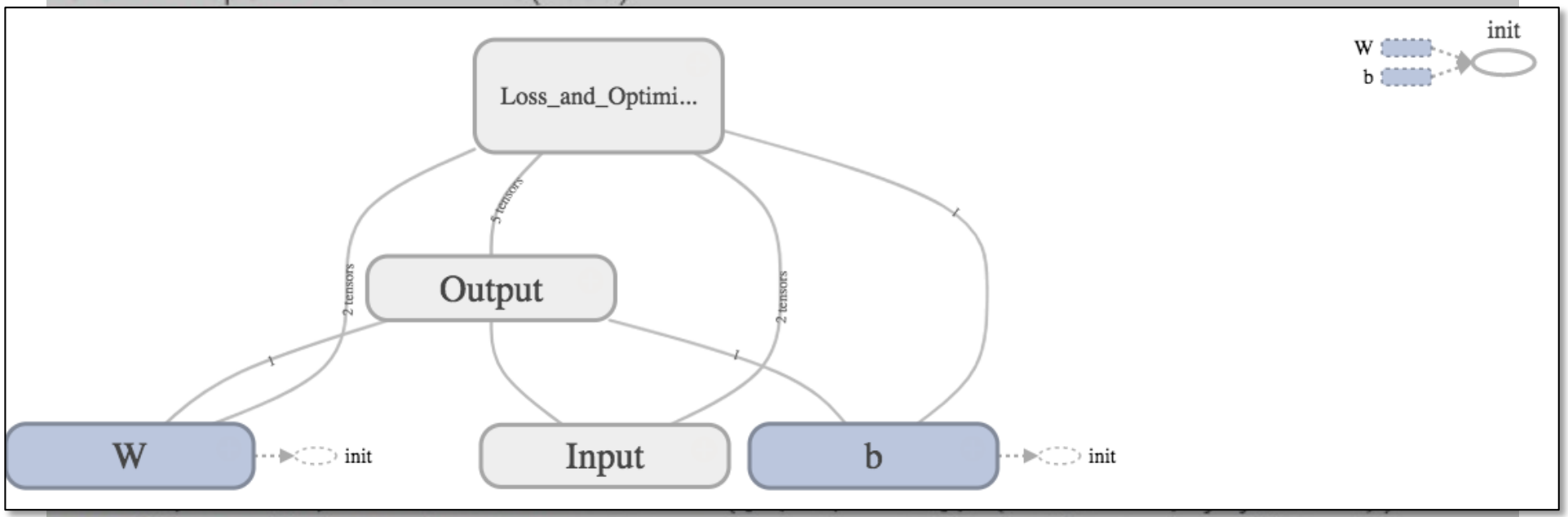
```
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
```

We can group several nodes under the same name:

```
with tf.variable_scope("output"):
  linear_model = W * x + b
  y = tf.placeholder(tf.float)
```

```
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

```python
with tf.variable_scope("W"):
  W = tf.Variable([.3], tf.float32)
with tf.variable_scope("b"):
  h = tf.Variable([-.3], tf.float32)
with tf.variable_scope("input"):
  x = tf.placeholder(tf.float)
with tf.variable_scope("output"):
  linear_model = W * x + b
  y = tf.placeholder(tf.float)
…
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
```

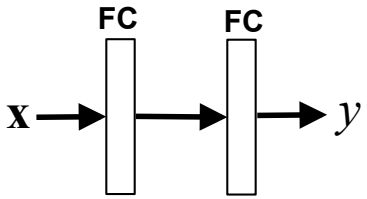


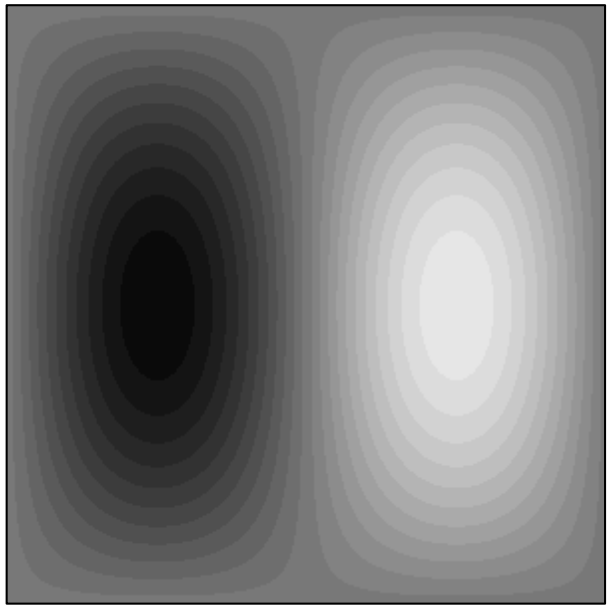

```python
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

Second Example: Two-Layer Network

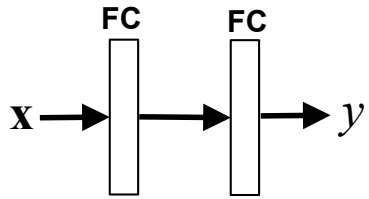
A Two-Layer Network

We will train a two-layer network to approximate a 2D function $\mathbf{F(x)}$:



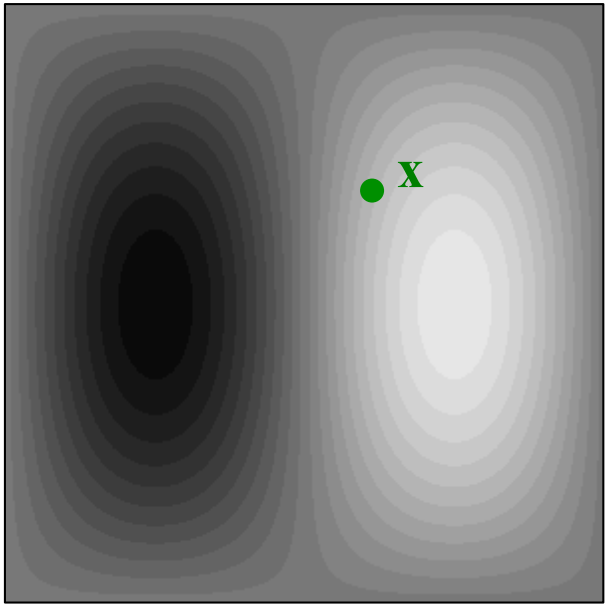


Our Two-Layer Network



The input is a 2D point $\mathbf{x}$;

The output is a scalar value y



In TensorFlow, it is simpler to consider left vector-matrix multiplications, and $\mathbf{x}$ will be a 2d row vector.

Hidden layer:

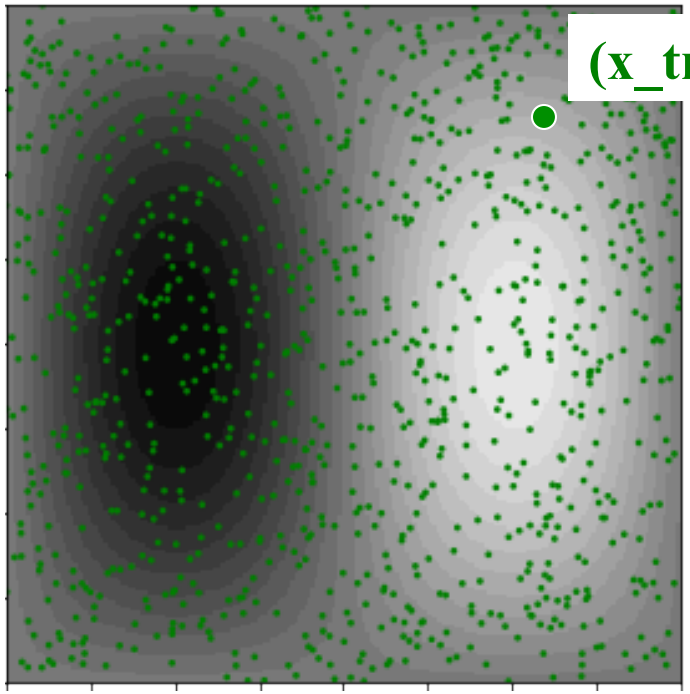
$$\mathbf{h}_1 = \mathrm{ReLU}(\mathbf{x}\,\mathbf{W}_1 + \mathbf{b}_1)$$

Output layer:

$$h_2 = \mathbf{h}_1\,\mathbf{W}_2 + b_2$$

Loss function

Training set:



Hidden layer:

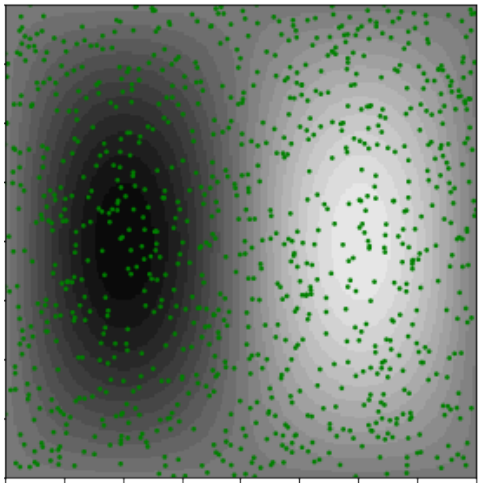
$\mathbf{h}_1 = \mathrm{ReLU}(\mathbf{x}\,\mathbf{W}_1 + \mathbf{b}_1)$

Output layer:

$h_2 = \mathbf{h}_1\,\mathbf{W}_2 + b_2$

$$\text{Loss} = \frac{1}{N_s}\sum_{i=1}^{N_s}\left(h_2(\mathbf{x\_train}) - y\_train_i\right)^2$$

Generating Training Data

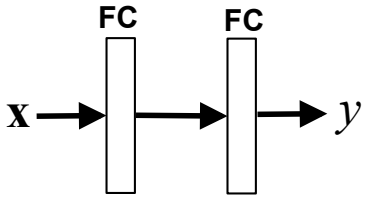


```
def F_mv(x1, x2):
    return np.sin(3.14 * x1 / 2.0) * np.cos(3.14 * x2 / 4.0)

A = 2
nb_samples = 1000
F = F_mv
X_train = np.random.uniform(-A, +A, shape=(nb_samples, 2))
# Each Y_train[i] is an array to be compatible with the output of the network,
# which will be a Tensor:
Y_train = np.zeros(shape=(nb_samples,1)
for i in range(nb_samples):
    Y_train[i] = [F(X_train[i][0],X_train[i][1])]
```

Defining the Network

Hidden layer: $\mathbf{h}_1 = \mathrm{ReLU}(\mathbf{x}\,\mathbf{W}_1 + \mathbf{b}_1)$

Output layer: $h_2 = \mathbf{h}_1\,\mathbf{W}_2 + b_2$




```
#Number of neurons in the hidden layer
N1 = 20

tf.reset_default_graph()
session = tf.InteractiveSession()

with tf.variable_scope("Input"):
    x = tf.placeholder(tf.float32, shape=[None,2])

with tf.variable_scope("HiddenLayer"):
    W1 = tf.Variable( tf.truncated_normal([2,N1], stddev=np.sqrt(6 / (2+N1)) ))
    b1 = tf.Variable(tf.zeros([N1]))
    h1 = tf.nn.relu( tf.add(tf.matmul(x, W1), b1) )

with tf.variable_scope("OutputLayer"):
    W2 = tf.Variable(tf.truncated_normal([N1,1], stddev=np.sqrt(6 / (1+N1))))
    b2 = tf.Variable(tf.constant(0.))
    h2 = tf.add(tf.matmul(h1, W2), b2)

with tf.variable_scope("Output"):
    y = tf.placeholder(tf.float32,[None,1])

with tf.variable_scope("Loss"):
    Loss = tf.reduce_mean(tf.square(h2 - y))
```

$$\text{Loss} = \frac{1}{N_s}\sum_{i=1}^{N_s}\left(h_2(\mathbf{x\_train}) - y\_train_i\right)^2$$

Running the Optimization

```
eta = 1e-3
optimizer_one_step = tf.train.GradientDescentOptimizer(eta).minimize(Loss)
session.run(tf.global_variables_initializer())

epochs = int(1e4)
batch_size = 10
epochs_between_two_evaluations = 1e3

# Network optimization:
for i in range(epochs):
    idx = np.random.permutation(X_train.shape[0])[:batch_size]
    feed_dict = {x:X_train[idx], y:Y_train[idx]}
    optimizer_one_step.run(feed_dict=feed_dict)
    if i % epochs_between_two_evaluations == 0:
        [curr_loss] = session.run([Loss], {x:X_train, y:Y_train})
        print(curr_loss)
```

Note the generation of the random batch: This is done by keeping the `batch_size` first elements of the `np.random.permutation` function

Visualizing the Predicted Function without using the `run()` function

```
[curr_W1, curr_b1, curr_W2, curr_b2, curr_loss] = \
  session.run([W1, b1, W2, b2, Loss], {x:X_train, y:Y_train})

curr_W1 = np.asmatrix(curr_W1)
curr_W2 = np.asmatrix(curr_W2)

I = visualize_2layers(100, 100, A, A, curr_W1, curr_b1, curr_W2, curr_b2)
plt.imshow(I)
```

visualize_2layers()

```
def ReLU(x):
    if x > 0.0:
        return x
    else:
        return 0.0

def visualize_2layers(Width, Height, U, V, W1, b1, W2, b2):
    H2 = np.empty((Height, Width))
    for i in range(Width):
        u = U * ((i -  Width / 2.0) / ( Width / 2.0))
        for j in range(Height):
            v = V * ((j - Height / 2.0) / (Height / 2.0))
            x = [u,v]
            h1 = np.vectorize(ReLU)(x * W1 + b1)
            h2 = h1 * W2 + b2
            H2[j,i] = h2[0,0]

    return H2
```

$$\mathbf{h}_1 = \mathrm{ReLU}(\mathbf{x}\,\mathbf{W}_1 + \mathbf{b}_1)$$
$$h_2 = \mathbf{h}_1\,\mathbf{W}_2 + b_2$$

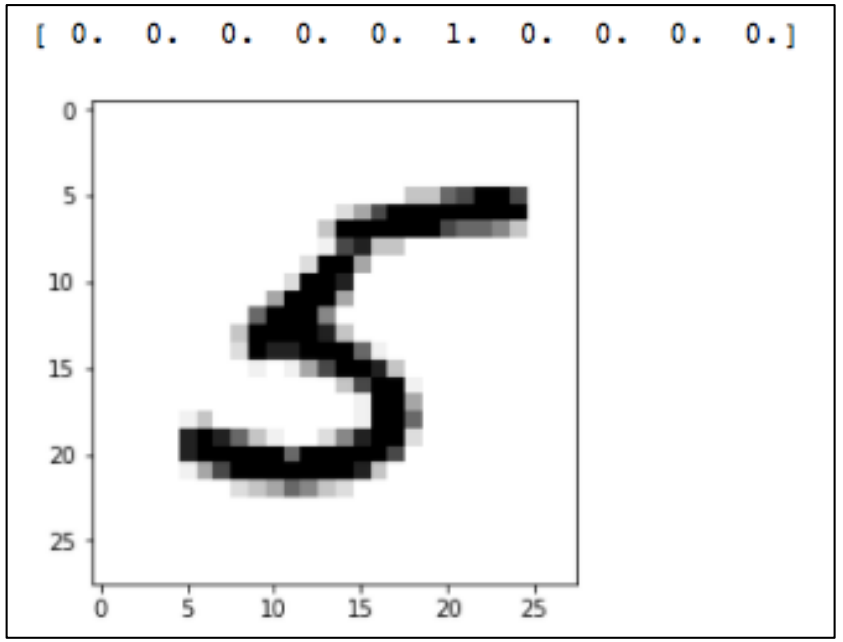
Third Example:
Linear Classification on MNIST

Downloading the MNIST Dataset

```
import numpy as np
import tensorflow as tf

#Read training, test, and validation data:
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

The images are stored as Python vectors.
We can visualize them with for example:

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
%matplotlib inline

im = mnist.train.images[0]
im = tmp.reshape(28,28)

plt.imshow(im, cmap = cm.Greys)
print(mnist.train.labels[0])
```



Model

$$\mathbf{y} = \text{softmax}(\mathbf{x}\mathbf{W} + \mathbf{b})$$

with

$$\text{softmax}(\mathbf{h})_i = \frac{\exp \mathbf{h}_i}{\sum_j \exp \mathbf{h}_j}$$

```
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# Placeholder for the input:
x = tf.placeholder(tf.float32, [None, n_input])

W = tf.Variable(tf.zeros([n_input, n_classes]))
b = tf.Variable(tf.zeros([n_classes]))

# Predicted output:
y_pred = tf.nn.softmax(tf.add(tf.matmul(x, W), b))
```

$$\mathbf{y} = \mathrm{softmax}(\mathbf{xW} + \mathbf{b})$$

Loss Function

Loss function, cross-entropy:

$$L(\mathbf{y}, \mathbf{y}_{\text{expected}}) = -\sum_i \mathbf{y}_{\text{expected}_i} \log(\mathbf{y}_i)$$

$$L(\mathbf{y}, \mathbf{y}_{\text{expected}}) = -\sum_i \mathbf{y}_{\text{expected}_i} \log(\mathbf{y}_i)$$

```
# Loss function:
cross_entropy =
  tf.reduce_mean(
    -tf.reduce_sum(
      y_exp * tf.log(y_pred),
      reduction_indices=[1]
    )
  )
```

Training

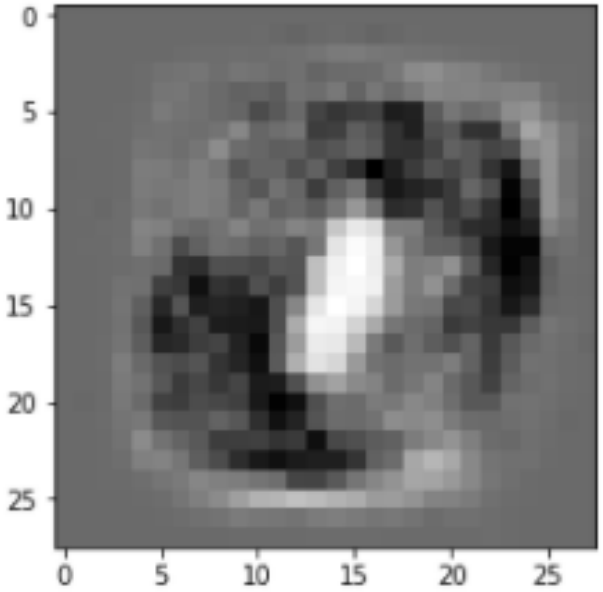
```
train_step = \
tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

sess = tf.InteractiveSession()
tf.global_variables_initializer().run()

for step in range(1000):
    print(step)
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_exp: batch_ys})
```

Testing

```
correct_prediction = tf.equal(tf.argmax(y_pred,1), tf.argmax(y_exp,1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

print(sess.run(accuracy, feed_dict={x: mnist.test.images, \
                                    y_exp: mnist.test.labels}))
```

Visualizing the Model (after optimization)

```
W_array = W.eval(sess)

#First column of W:
I = W_array.flatten()[0::10].reshape((28,28))

plt.imshow(I, cmap = cm.Greys)
```



Visualizing the Model During Optimization

TensorFlow comes with TensorBoard, a program that can display data saved using TensorFlow functions on a browser.

To visualize the columns of W during optimization with TensorBoard, we need to:

1. create a Tensor that contains these columns as an image. This will be done by our function `display_W`;

2. tell TensorFlow that this Tensor is an image and part of a 'summary': A 'summary' is made of data useful for monitoring the optimization, and made to be read by TensorBoard.

3. create a FileWriter object.

4. during optimization, we can save the image of the columns using the FileWriter object, and visualize the images using TensorBoard.

```
display_W( )
```

Third Example:
A Convolutional Neural Network

Loading the Data

As before:

```
import numpy as np
import tensorflow as tf

#Read training, test, and validation data:
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

Model

$$\mathbf{h}_1 = [\mathrm{ReLU}(f_{1,1} * \mathbf{x}), \ldots, \mathrm{ReLU}(f_{1,n} * \mathbf{x})]$$
$$\mathbf{h}_2 = [\mathrm{pool}(\mathbf{h}_{1,1}), \ldots, \mathrm{pool}(\mathbf{h}_{1,n})]$$
$$\mathbf{h}_3 = [\mathrm{ReLU}(f_{3,1} * \mathbf{h}_{2,1}), \ldots, \mathrm{ReLU}(f_{3,n} * \mathbf{h}_{2,n})]$$
$$\mathbf{h}_4 = [\mathrm{pool}(\mathbf{h}_{3,1}), \ldots, \mathrm{pool}(\mathbf{h}_{3,n})]$$
$$\mathbf{h}_5 = \mathrm{ReLU}(\mathbf{W}_5\mathbf{h}_4 + \mathbf{b}_5)$$
$$\mathbf{y} = \mathbf{W}_6\mathbf{h}_5 + \mathbf{b}_6$$

We Need to Convert the Input Vectors into Images

```
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

#Placeholder for the input:
x = tf.placeholder(tf.float32, [None, n_input])

#images are stored as vectors, we reshape them as images:
im = tf.reshape(x, shape=[-1, 28, 28, 1])  # 28x28 = 784
```

First Convolutional Layer

$$\mathbf{h}_1 = [\mathrm{ReLU}(f_{1,1} * \mathbf{x}), \ldots, \mathrm{ReLU}(f_{1,n} * \mathbf{x})]$$

```
#32 convolutional 5x5 filters and biases on the first layer:
# Filters and biases are initialized
# using values drawn from a normal distribution:
F1 = tf.Variable(tf.random_normal([5, 5, 1, 32]))
b1 = tf.Variable(tf.random_normal([32]))
F1_im = tf.nn.conv2d(im, F1, strides=[1, 1, 1, 1],\
                     padding='SAME')
h1 = tf.nn.relu( tf.nn.bias_add(F1_im, b1) )
```

First Pooling Layer

$$\mathbf{h}_2 = [\text{pool}(\mathbf{h}_{1,1}), \ldots, \text{pool}(\mathbf{h}_{1,n})]$$

```
#Pooling on 2x2 regions:
h2 = tf.nn.max_pool(h1,
                    ksize=[1, 2, 2, 1],
                    strides=[1, 2, 2, 1],
                    padding='SAME')
```

Second Convolutional and Pooling Layers

$$\mathbf{h}_3 = [\mathrm{ReLU}(f_{3,1} * \mathbf{h}_{2,1}), \ldots, \mathrm{ReLU}(f_{3,n} * \mathbf{h}_{2,n})]$$
$$\mathbf{h}_4 = [\mathrm{pool}(\mathbf{h}_{3,1}), \ldots, \mathrm{pool}(\mathbf{h}_{3,n})]$$

```
#Second convolutional layer: 64 5x5x32 filters:
F3 = tf.Variable(tf.random_normal([5, 5, 32, 64]))
b3 = tf.Variable(tf.random_normal([64]))
F3_im = tf.nn.conv2d(h2, F3, strides=[1, 1, 1, 1],
                     padding='SAME')
h3 = tf.nn.relu( tf.nn.bias_add(F3_im, b3) )

#Second pooling layer:
h4 = tf.nn.max_pool(h3, ksize=[1, 2, 2, 1],
                    strides=[1, 2, 2, 1], padding='SAME')
```

Two Fully Connected Layers

$$\mathbf{h}_5 = \mathrm{ReLU}(\mathbf{W}_5\mathbf{h}_4 + \mathbf{b}_5)$$
$$\mathbf{y} = \mathbf{W}_6\mathbf{h}_5 + \mathbf{b}_6$$

```
#First fully connected layer, 1024 output:
h4_vect = tf.reshape(h4, [-1, 7*7*64])
W5 = tf.Variable(tf.random_normal([7*7*64, 1024]))
b5 = tf.Variable(tf.random_normal([1024]))
h5 = tf.nn.relu( tf.add(tf.matmul(h4_vect, W5), b5 ))

#Second fully connected layer, 1024 input, 10 output:
W6 = tf.Variable(tf.random_normal([1024, n_classes]))
b6 = tf.Variable(tf.random_normal([n_classes]))

#Final predicted output:
y_pred = tf.add(tf.matmul(h5, W6), b6)


#Placeholder for the expected output:
y_exp = tf.placeholder(tf.float32, [None, n_classes])
```

Two Fully Connected Layers

```
loss = tf.reduce_mean(
  tf.nn.softmax_cross_entropy_with_logits(
    logits=y_pred, labels=y_exp
  )
)

optimizer = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)

sess = tf.InteractiveSession()

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

Optimization

```
step = 1
training_iters = 20000000
batch_size = 128

# Keep training until reach max iterations
while step * batch_size < training_iters:
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    sess.run(optimizer,
             feed_dict={x: batch_x, y_exp: batch_y})
    step += 1
```

Adding Evaluation (1)

Before optimization, let's define:

```
# Evaluate model
is_prediction_correct = tf.equal(tf.argmax(y_pred, 1),
                                 tf.argmax(y_exp,  1))
accuracy = tf.reduce_mean(tf.cast(is_prediction_correct,
                                  tf.float32))
```

Adding Evaluation (2)

Printing performance on test set during optimization:

```
while step * batch_size < training_iters:
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    sess.run(optimizer, feed_dict={x: batch_x,
                                   y_exp: batch_y})
    if step % display_step == 0:
        # Calculate batch loss and accuracy
        acc  = sess.run([accuracy],
                        feed_dict={x: mnist.test.images,
                                   y_exp: mnist.test.labels})
        print(acc)
    step += 1
```

Adding Dropout (1)

Dropout is not really useful here, but we will see how to add it to this simple example:

```
#First fully connected layer, 1024 output:
h4_vect = tf.reshape(h4, [-1, 7*7*64])
W5 = tf.Variable(tf.random_normal([7*7*64, 1024]))
b5 = tf.Variable(tf.random_normal([1024]))
h5 = tf.nn.relu( tf.add(tf.matmul(h4_vect, W5), b5 ))

keep_prob = tf.placeholder(tf.float32)
h5 = tf.nn.dropout(h5, keep_prob)
```

`keep_prob` will be set to 0.5 for training, and 1.0 for actual evaluation.

Adding Dropout (2)

```
# Keep training until reach max iterations
while step * batch_size < training_iters:
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    # Optimization:
    sess.run(optimizer,
             feed_dict={x: batch_x,
                        y_exp: batch_y,
                        keep_prob: 0.5})
    # Evaluation:
    if step % display_step == 0:
        # Calculate batch loss and accuracy
        acc  = sess.run([accuracy],
                        feed_dict={x: mnist.test.images,
                                   y_exp: mnist.test.labels,
                                   keep_prob: 1.0}
                        )
        print(acc)
    step += 1
```