Lecture note 3: Linear and Logistic Regression in TensorFlow

"CS 20SI: TensorFlow for Deep Learning Research" (cs20si.stanford.edu)
Prepared by Chip Huyen (huyenn@stanford.edu)
Reviewed by Danijar Hafner

We've learned a lot in the previous two lectures. These are the key concepts from the previous two lectures. If you're unsure about any of these, you should go back to the previous two lectures to make sure.

Graphs and sessions

TF Ops: constants, variables, functions

TensorBoard!

Lazy loading

We've already covered the fundamentals of TensorFlow. Yes, we're that fast! Let's put them all together to see what we could do.

1. Linear Regression in TensorFlow

Let's start with a simple linear regression example. I hope you all are already familiar with linear regression. If not, you can read about it on Wikipedia.

Problem: We often hear insurance companies using factors such as number of fire and theft in a neighborhood to calculate how dangerous the neighborhood is. My question is: is it redundant? Is there a relationship between the number of fire and theft in a neighborhood, and if there is, can we find it?

In other words, can we find a function f so that if X is the number of fires and Y is the number of thefts, then: Y = f(X)?

Given the relationship, if we have the number of fires in a particular area, can we predict the number of thefts in that area.

We have the dataset collected by the U.S. Commission on Civil Rights, courtesy of Cengage Learning.

Dataset Description:
Name: Fire and Theft in Chicago
X = fires per 1000 housing units

Y = thefts per 1000 population
within the same Zip code in the Chicago metro area
Total number of Zip code areas: 42

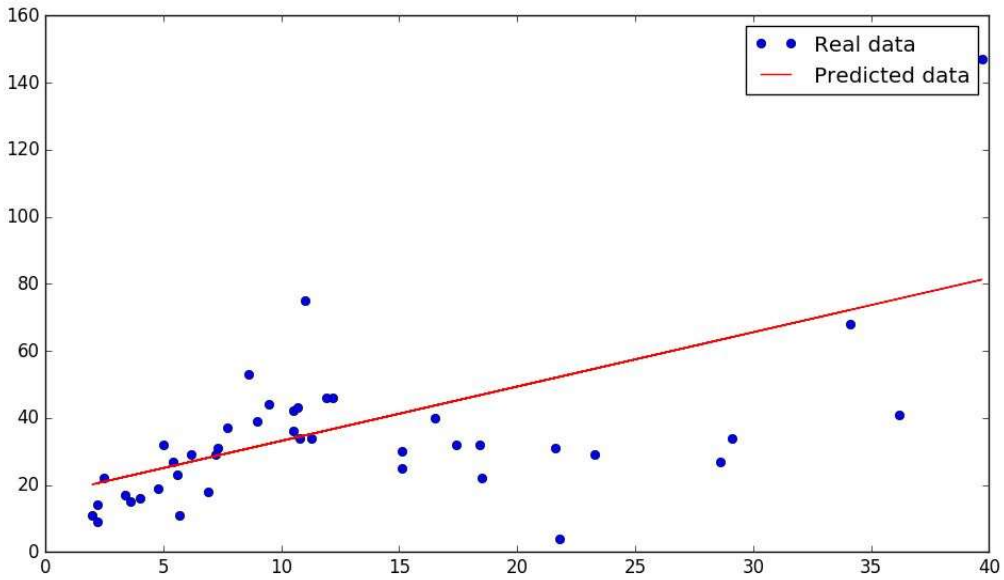
Solution:
First, assume that the relationship between the number of fires and thefts are linear:
Y = wX + b

We need to find the scalar parameters w and b, using mean squared error as the loss function.
Let's write the program.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import xlrd

DATA_FILE = "data/fire_theft.xls"

# Step 1: read in data from the .xls file
book = xlrd.open_workbook(DATA_FILE, encoding_override="utf-8")
sheet = book.sheet_by_index(0)
data = np.asarray([sheet.row_values(i) for i in range(1, sheet.nrows)])
n_samples = sheet.nrows - 1

# Step 2: create placeholders for input X (number of fire) and label Y (number of
theft)
X = tf.placeholder(tf.float32, name="X")
Y = tf.placeholder(tf.float32, name="Y")

# Step 3: create weight and bias, initialized to 0
w = tf.Variable(0.0, name="weights")
b = tf.Variable(0.0, name="bias")

# Step 4: construct model to predict Y (number of theft) from the number of fire
Y_predicted = X * w + b

# Step 5: use the square error as the loss function
loss = tf.square(Y - Y_predicted, name="loss")

# Step 6: using gradient descent with learning rate of 0.01 to minimize loss
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)

with tf.Session() as sess:
	# Step 7: initialize the necessary variables, in this case, w and b
	sess.run(tf.global_variables_initializer())

	# Step 8: train the model
	for i in range(100): # run 100 epochs
		for x, y in data:
			# Session runs train_op to minimize loss
```

```
                    sess.run(optimizer, feed_dict={X: x, Y:y})

        # Step 9: output the values of w and b
        w_value, b_value = sess.run([w, b])
```

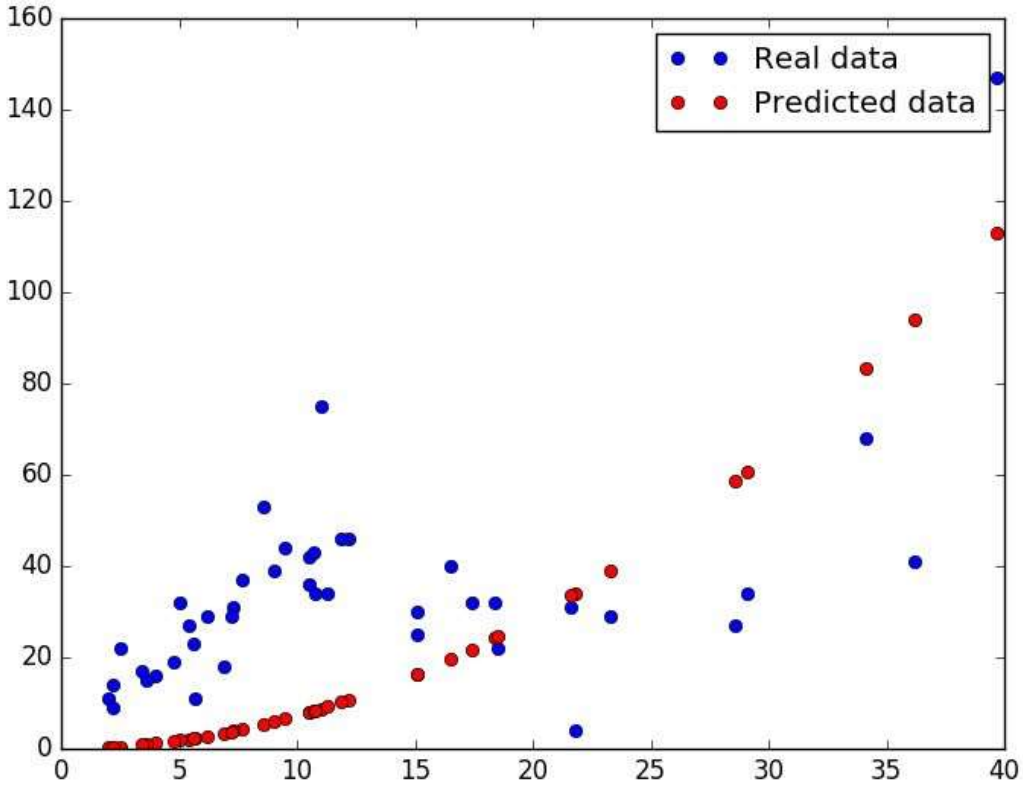
After training for 100 epochs, we got the average square loss to be 1372.77701716 with w = 1.62071, b = 16.9162. The loss is quite large.



It doesn't quite fit. Can we do better with quadratic function Y = wXX + uX + b?
Let's try. We only have to add another variable b and change the formula for Y_predicted.

```
# Step 3: create variables: weights_1, weights_2, bias. All are initialized to 0
w = tf.Variable(0.0, name="weights_1")
u = tf.Variable(0.0, name="weights_2")
b = tf.Variable(0.0, name="bias")

# Step 4: predict Y (number of theft) from the number of fire
Y_predicted = X * X * w + X * u + b

# Step 5: Profit!
```

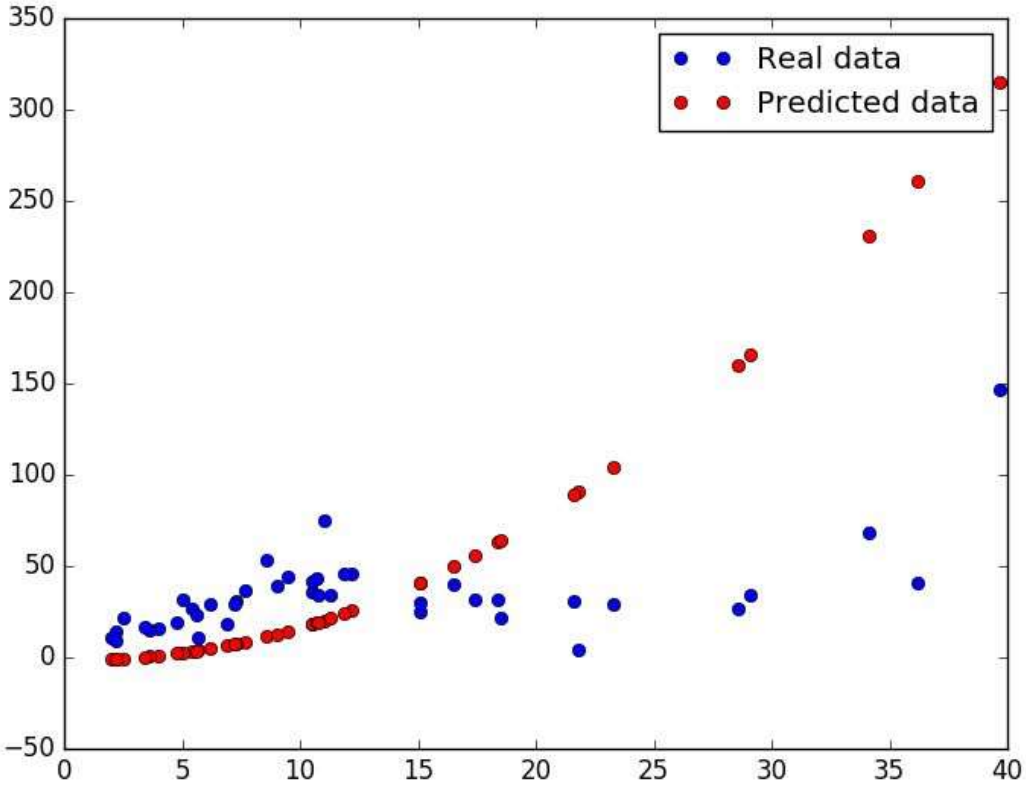
After 10 epochs, we got the average square loss to be 797.335975976 with w, u, b = [0.071343 0.010234 0.00143057]



This takes less time to converge than the linear function, but still completely off due to the several outliers on the right. It probably does better with Huber loss[1] instead of MSE or a 3rd degree polynomial as the function f. You can try at home.

Using Huber loss for quadratic model, I got something that's slightly better at ignoring the outliers:

[1] The Huber loss is basically a compromise between absolute loss and squared loss. Huber loss function is quadratic for residuals smaller than a certain value, and linear for residuals larger than that certain value.



How do we know that our model is correct?

Use correlation coefficient R-squared

In case you don't know what R-squared is, Minitab has a great blog post explaining it here. Below is the gist of it:

"R-squared is a statistical measure of how close the data are to the fitted regression line.

It is also known as the coefficient of determination, or the coefficient of multiple determination for multiple regression.

The definition of R-squared is fairly straight-forward; it is the percentage of the response variable variation that is explained by a linear model.

R-squared = Explained variation / Total variation"

Run on a test set

We've learned in machine learning class that it all comes down to validation and testing. So the first method is obviously to test our model on a test set.

Having separate datasets for training, validating, and testing are great, but this means that we will have less data for training. There is a lot of literature that helps us get around the cases where we don't have a lot of data to spare, such as k-fold cross validation.

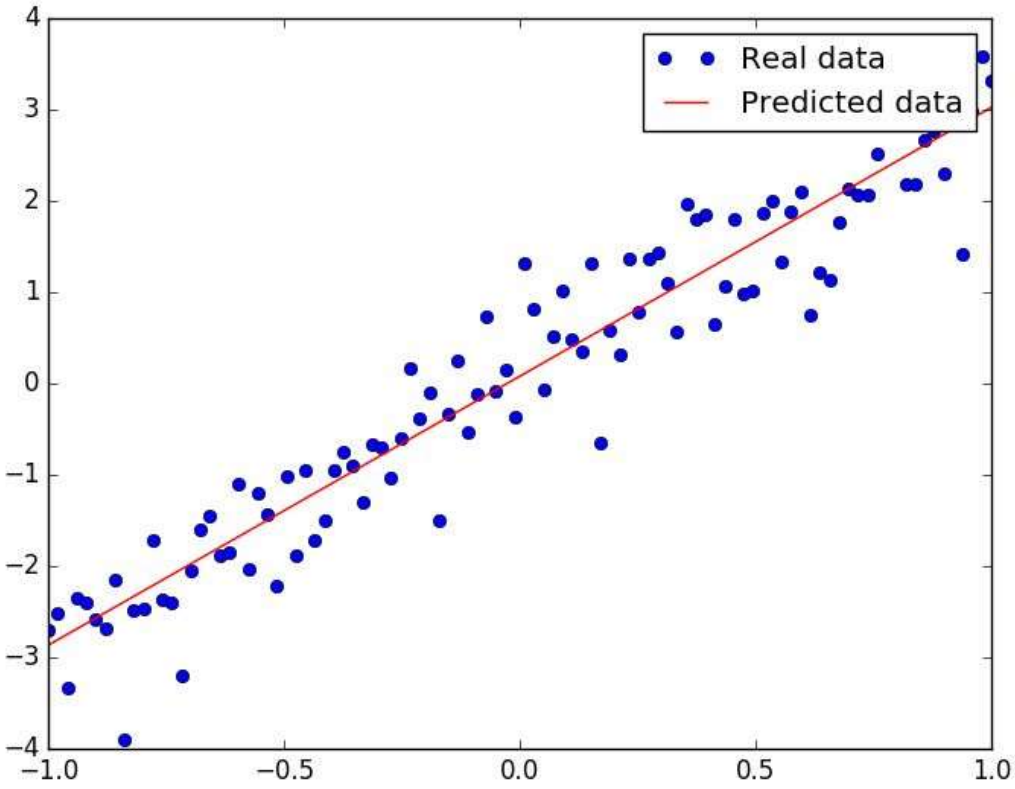
Test our model with dummy data

Another way we can test our model is test it on dummy data. For example, in this case, we can create some dummy data whose linear relation we already know to test our model. In this case, let's create 100 data points (X, Y) such that Y ~ 3 * X, and see if our model output w = 3, b = 0.

Generating dummy data:

```
# each value y is approximately linear but with some random noise

X_input = np.linspace(-1, 1, 100)
Y_input = X_input * 3 + np.random.randn(X_input.shape[0]) * 0.5
```

We use numpy array for X_input and Y_input to support iteration later (when we feed in inputs to placeholders X and Y).



It fits beautifully!

Moral of the story: dummy data is a lot easier to handle than real world data, because dummy data was generated to match the assumptions of our model. Real world is tough!

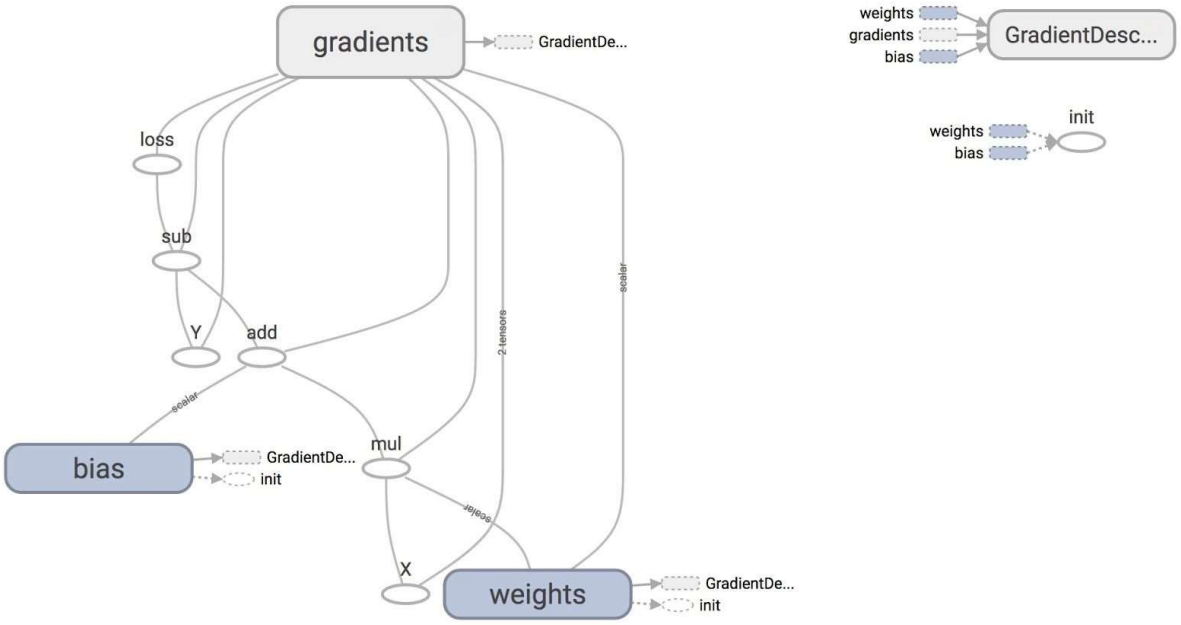
Analyze the code

The code in our model is pretty straightforward, except for two lines:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(loss)
sess.run(optimizer, feed_dict={X: x, Y:y})
```

I remember the first time I ran into code similar to these, I was very confused. Two questions:
1. Why is train_op in the fetches list of tf.Session.run()?
2. How does TensorFlow know what variables to update?

We can actually pass any TensorFlow ops as fetches in tf.Session.run(). TensorFlow will execute the part of the graph that those ops depend on. In this case, we see that train_op has the purpose of minimize loss, and loss depends on variables w and b.



From the graph, you can see that the giant node GrandientDescentOptimizer depends on 3 nodes: weights, bias, and gradients (which is automatically taken care of for us).

Optimizers

GradientDescentOptimizer means that our update rule is gradient descent. TensorFlow does auto differentiation for us, then update the values of w and b to minimize the loss. Autodiff is amazing!

By default, the optimizer trains all the trainable variables whose objective function depend on. If there are variables that you do not want to train, you can set the keyword trainable to False when you declare a variable. One example of a variable you don't want to train is the variable global_step, a common variable you will see in many TensorFlow model to keep track of how many times you've run your model.

```
global_step = tf.Variable(0, trainable=False, dtype=tf.int32)
learning_rate = 0.01 * 0.99 ** tf.cast(global_step, tf.float32)

increment_step = global_step.assign_add(1)
```

```
optimizer = tf.GradientDescentOptimizer(learning_rate) # learning rate can be a
tensor
```

While we are at it, let's look at the full definition of the class tf.Variable:

```
tf.Variable(initial_value=None, trainable=True, collections=None,
validate_shape=True, caching_device=None, name=None, variable_def=None, dtype=None,
expected_shape=None, import_scope=None)
```

You can also ask your optimizer to take gradients of specific variables. You can also modify the gradients calculated by your optimizer.

```
# create an optimizer.
optimizer = GradientDescentOptimizer(learning_rate=0.1)

# compute the gradients for a list of variables.
grads_and_vars = opt.compute_gradients(loss, <list of variables>)

# grads_and_vars is a list of tuples (gradient, variable).  Do whatever you
# need to the 'gradient' part, for example, subtract each of them by 1.
subtracted_grads_and_vars = [(gv[0] - 1.0, gv[1]) for gv in grads_and_vars]

# ask the optimizer to apply the subtracted gradients.
optimizer.apply_gradients(subtracted_grads_and_vars)
```

More on computing gradients

The optimizer classes automatically compute derivatives on your graph, but creators of new Optimizers or expert users can call the lower-level functions below.

```
tf.gradients(ys, xs, grad_ys=None, name='gradients',
colocate_gradients_with_ops=False, gate_gradients=False, aggregation_method=None)
```

This method constructs symbolic partial derivatives of sum of ys w.r.t. x in xs. ys and xs are each a Tensor or a list of tensors. grad_ys is a list of Tensor, holding the gradients received by the ys. The list must be the same length as ys.

Technical detail: This is especially useful when training only parts of a model. For example, we can use tf.gradients() for to take the derivative G of the loss w.r.t. to the middle layer. Then we use an optimizer to minimize the difference between the middle layer output M and M + G. This only updates the lower half of the network.

List of optimizers

GradientDescentOptimizer is not the only update rule that TensorFlow supports. Here is the list of optimizers that TensorFlow supports, as of 1/8/2017. The names are self-explanatory. You can visit the official documentation for more details:

tf.train.GradientDescentOptimizer
tf.train.AdadeltaOptimizer
tf.train.AdagradOptimizer
tf.train.AdagradDAOptimizer
tf.train.MomentumOptimizer
tf.train.AdamOptimizer
tf.train.FtrlOptimizer
tf.train.ProximalGradientDescentOptimizer
tf.train.ProximalAdagradOptimizer
tf.train.RMSPropOptimizer

Sebastian Ruder, a PhD candidate at the Insight Research Centre for Data Analytics did a pretty great comparison of these optimizers in his blog post. If you're too lazy to read, here is the conclusion:

"RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numerator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Kingma et al. [15] show that its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice."

TL;DR: Use AdamOptimizer.

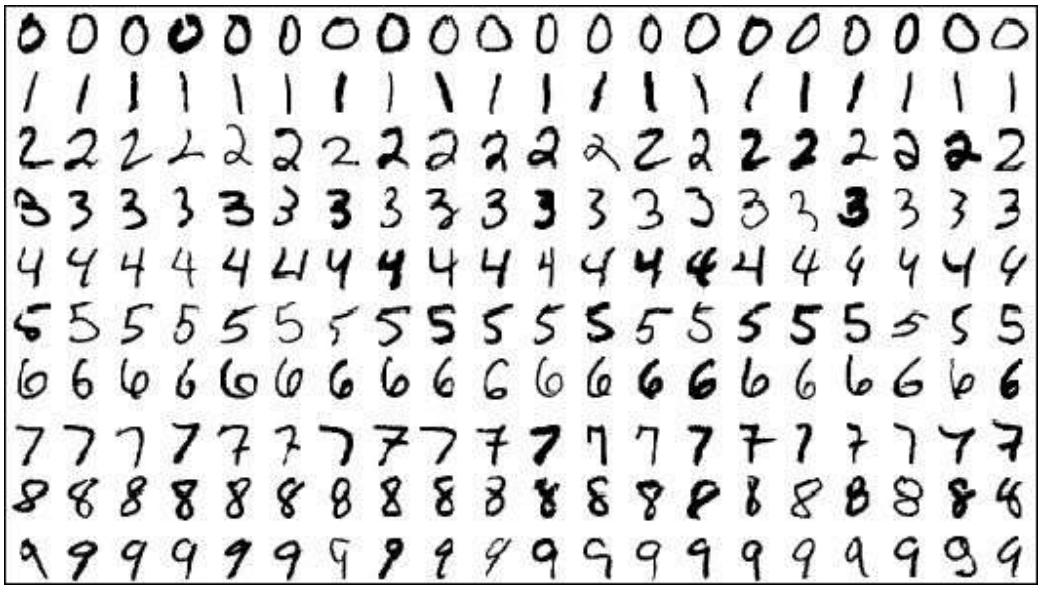
Discussion questions

What are some of the real world problems that we can solve using linear regression? Can you write a quick program to do so?

Logistic Regression in TensorFlow

We can't talk about linear regression without logistic regression. Let's illustrate logistic regression in TensorFlow solving the good old classifier on the MNIST database.

The MNIST database (Mixed National Institute of Standards and Technology database) is probably one of the most popular databases used for training various image processing systems It is a database of handwritten digits. The images look like this:



Each image is 28 x 28 pixels, flatten to be a 1-d tensor of size 784. Each comes with a label. For example, images on the first row is labelled as 0, the second as 1, and so on. The dataset is hosted on Yann Lecun's website (http://yann.lecun.com/exdb/mnist/).

TF Learn (the simplified interface of TensorFlow) has a script that lets you load the MNIST dataset from Yann Lecun's website and divide it into train set, validation set, and test set.

```
from tensorflow.examples.tutorials.mnist import input_data
MNIST = input_data.read_data_sets("/data/mnist", one_hot=True)
```

One-hot encoding
In digital circuits, one-hot refers to a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0).

In this case, one-hot encoding means that if the output of the image is the digit 7, then the output will be encoded as a vector of 10 elements with all elements being 0, except for the element at index 7 which is 1.

MNIST is a TensorFlow's Datasets object. It has 55,000 data points of training data (MNIST.train), 10,000 points of test data (MNIST.test), and 5,000 points of validation data (MNIST.validation).

The construction of the logistic regression model is pretty similar to the linear regression model. However, now we have A LOT more data. We learned in CS229 that if we calculate gradient after every single data point it'd be painfully slow. One way to go around this is to batch 'em up. Fortunately, TensorFlow has a wonderful support for batching data.

To do batched logistic regression, we just need to change the dimension of X_placeholder and Y_placeholder to be able to accommodate batch_size data points.

```
X = tf.placeholder(tf.float32, [batch_size, 784], name="image")
Y = tf.placeholder(tf.float32, [batch_size, 10], name="label")
```

And when you feed in data to the placeholder, instead of feeding each data point, we can feed in the batch_size number of data points.

```
X_batch, Y_batch = mnist.test.next_batch(batch_size)
sess.run(train_op, feed_dict={X: X_batch, Y:Y_batch})
```

Here is the full implementation.

```
import time
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

# Step 1: Read in data
# using TF Learn's built in function to load MNIST data to the folder data/mnist
MNIST = input_data.read_data_sets("/data/mnist", one_hot=True)

# Step 2: Define parameters for the model
learning_rate = 0.01
batch_size = 128
n_epochs = 25

# Step 3: create placeholders for features and labels
# each image in the MNIST data is of shape 28*28 = 784
# therefore, each image is represented with a 1x784 tensor
# there are 10 classes for each image, corresponding to digits 0 - 9.
# each label is one hot vector.
X = tf.placeholder(tf.float32, [batch_size, 784])
Y = tf.placeholder(tf.float32, [batch_size, 10])

# Step 4: create weights and bias
# w is initialized to random variables with mean of 0, stddev of 0.01
# b is initialized to 0
# shape of w depends on the dimension of X and Y so that Y = tf.matmul(X, w)
# shape of b depends on Y
w = tf.Variable(tf.random_normal(shape=[784, 10], stddev=0.01), name="weights")
```

```
b = tf.Variable(tf.zeros([1, 10]), name="bias")

# Step 5: predict Y from X and w, b
# the model that returns probability distribution of possible label of the image
# through the softmax layer
# a batch_size x 10 tensor that represents the possibility of the digits
logits = tf.matmul(X, w) + b

# Step 6: define loss function
# use softmax cross entropy with logits as the loss function
# compute mean cross entropy, softmax is applied internally
entropy = tf.nn.softmax_cross_entropy_with_logits(logits, Y)
loss = tf.reduce_mean(entropy) # computes the mean over examples in the batch

# Step 7: define training op
# using gradient descent with learning rate of 0.01 to minimize cost
optimizer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(loss)

init = tf.global_variables_initializer()

with tf.Session() as sess:
       sess.run(init)
       n_batches = int(MNIST.train.num_examples/batch_size)
       for i in range(n_epochs): # train the model n_epochs times
              for _ in range(n_batches):
                     X_batch, Y_batch = MNIST.train.next_batch(batch_size)
                     sess.run([optimizer, loss], feed_dict={X: X_batch, Y:Y_batch})

# average loss should be around 0.35 after 25 epochs
```

Running on my Mac, the batch version of the model with batch size 128 runs in 0.5 second, while the non-batch model runs in 24 seconds! However, note that higher batch size typically requires more epochs since it does fewer update steps. See “mini-batch size” in Bengio's practical tips.

We can actually test the model because we have a test set. Let’s see how we can do it in TensorFlow.

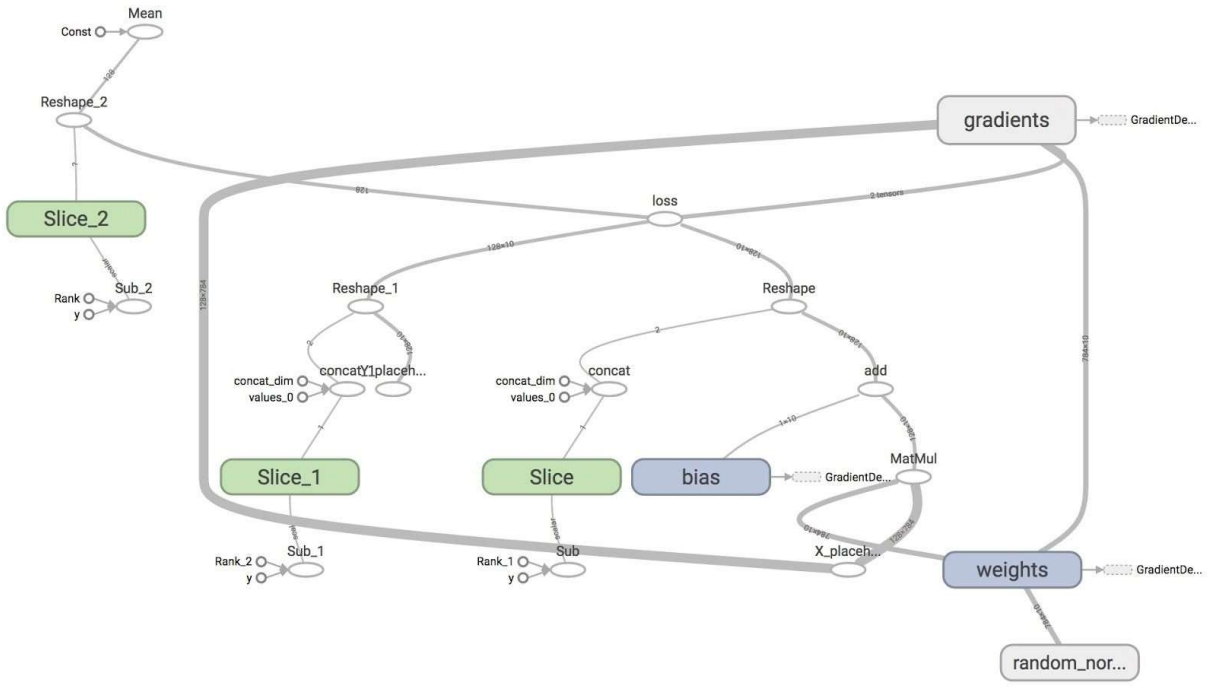
```
# test the model
       n_batches = int(MNIST.test.num_examples/batch_size)
       total_correct_preds = 0
       for i in range(n_batches):
              X_batch, Y_batch = MNIST.test.next_batch(batch_size)
              _, loss_batch, logits_batch = sess.run([optimizer, loss, logits],
feed_dict={X: X_batch, Y:Y_batch})
              preds = tf.nn.softmax(logits_batch)
              correct_preds = tf.equal(tf.argmax(preds, 1), tf.argmax(Y_batch, 1))
              accuracy = tf.reduce_sum(tf.cast(correct_preds, tf.float32)) # similar
```

```
to numpy.count_nonzero(boolarray) :(
              total_correct_preds += sess.run(accuracy)

       print "Accuracy {0}".format(total_correct_preds/MNIST.test.num_examples)
```

We achieved the accuracy of 90% after 10 epochs. This is about what we can get from a linear classifier.

Note: TensorFlow has a feeder (dataset parser) for MNIST but don't count on it having a feeder for any dataset. You should learn how to write your own data parser.

Here is how our graph looks in on TensorBoard:



WOW
I know. That's why we'll learn how to structure our model in the next lecture!