Introduction

Let's get you up and running with TensorFlow!

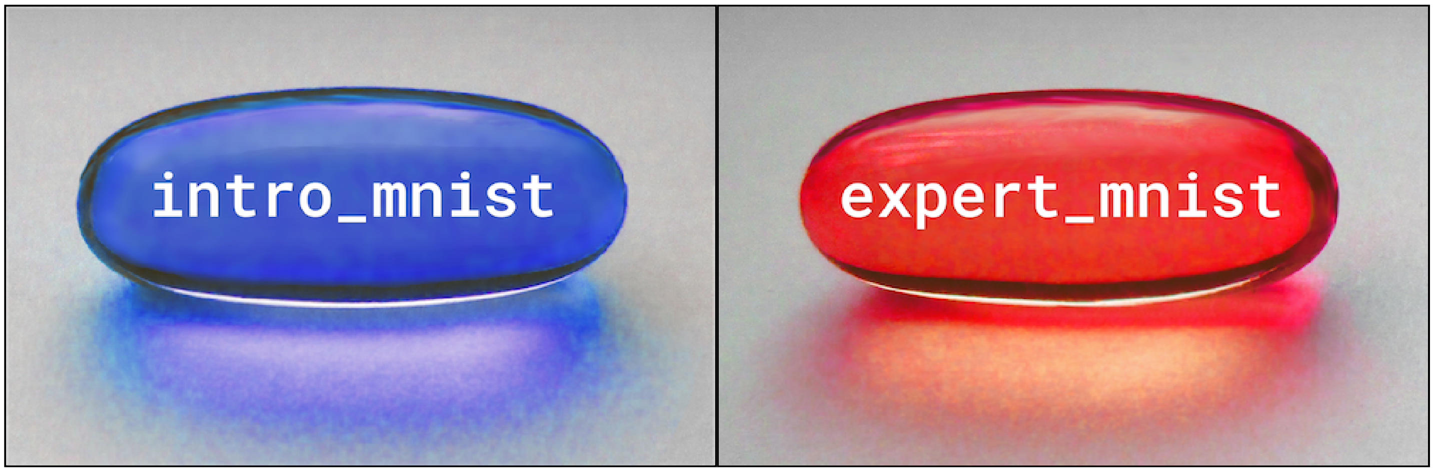
But before we even get started, let's peek at what TensorFlow code looks like in the Python API, so you have a sense of where we're headed.

Here's a little Python program that makes up some data in two dimensions, and then fits a line to it.

```
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# Try to find values for W and b that compute y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but TensorFlow will
# figure that out for us.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimize the mean squared errors.
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Before starting, initialize the variables.  We will 'run' this first.
init = tf.initialize_all_variables()

# Launch the graph.
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))
    # Learns best fit is W: [0.1], b: [0.3]
```

The first part of this code builds the data flow graph. TensorFlow does not actually run any computation until the session is created and the `run` function is called.

To whet your appetite further, we suggest you check out what a classical machine learning problem looks like in TensorFlow. In the land of neural networks the most "classic" classical problem is the MNIST handwritten digit classification. We offer two introductions here, one for machine learning newbies, and one for pros. If you've already trained dozens of MNIST models in other software packages, please take the red pill. If you've never even heard of MNIST, definitely take the blue pill. If you're somewhere in between, we suggest skimming blue, then red.



Images licensed CC BY-SA 4.0; original by W. Carter

If you're already sure you want to learn and install TensorFlow you can skip these and charge ahead. Don't worry, you'll still get to see MNIST -- we'll also use MNIST as an example in our technical tutorial where we elaborate on TensorFlow features.

Recommended Next Steps

- Download and Setup
- Basic Usage
- TensorFlow Mechanics 101
- Tinker with a neural network in your browser

MNIST For ML Beginners

This tutorial is intended for readers who are new to both machine learning and TensorFlow. If you already know what MNIST is, and what softmax (multinomial logistic) regression is, you might prefer this faster paced tutorial. Be sure to install TensorFlow before starting either tutorial.

When one learns how to program, there's a tradition that the first thing you do is print "Hello World." Just like programming has Hello World, machine learning has MNIST.

MNIST is a simple computer vision dataset. It consists of images of handwritten digits like these:

It also includes labels for each image, telling us which digit it is. For example, the labels for the above images are 5, 0, 4, and 1.

In this tutorial, we're going to train a model to look at images and predict what digits they are. Our goal isn't to train a really elaborate model that achieves state-of-the-art performance -- although we'll give you code to do that later! -- but rather to dip a toe into using TensorFlow. As such, we're going to start with a very simple model, called a Softmax Regression.

The actual code for this tutorial is very short, and all the interesting stuff happens in just three lines. However, it is very important to understand the ideas behind it: both how TensorFlow works and the core machine learning concepts. Because of this, we are going to very carefully work through the code.

About this tutorial

This tutorial is an explanation, line by line, of what is happening in the mnist_softmax.py code.

You can use this tutorial in a few different ways, including:

- Copy and paste each code snippet, line by line, into a Python environment as you read through the explanations of each line.

- Run the entire `mnist_softmax.py` Python file either before or after reading through the explanations, and use this tutorial to understand the lines of code that aren't clear to you.

What we will accomplish in this tutorial:

- Learn about the MNIST data and softmax regressions

- Create a function that is a model for recognizing digits, based on looking at every pixel in the image

- Use Tensorflow to train the model to recognize digits by having it "look" at thousands of examples (and run our first Tensorflow session to do so)

- Check the model's accuracy with our test data

The MNIST Data

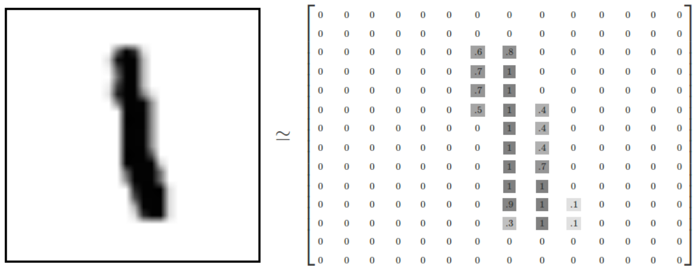
The MNIST data is hosted on Yann LeCun's website. If you are copying and pasting in the code from this tutorial, start here with these two lines of code which will download and read in the data automatically:

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

The MNIST data is split into three parts: 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`). This split is very important: it's essential in machine learning that we have separate data which we don't learn from so that we can make sure that what we've learned actually generalizes!

As mentioned earlier, every MNIST data point has two parts: an image of a handwritten digit and a corresponding label. We'll call the images "x" and the labels "y". Both the training set and test set contain images and their corresponding labels; for example the training images are `mnist.train.images` and the training labels are `mnist.train.labels`.

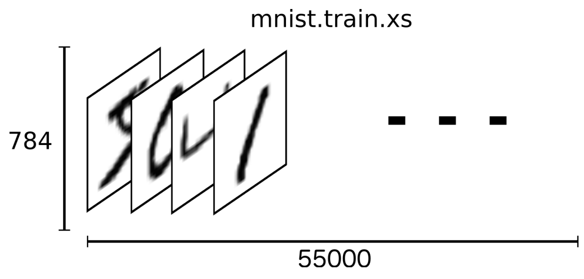
Each image is 28 pixels by 28 pixels. We can interpret this as a big array of numbers:



We can flatten this array into a vector of 28x28 = 784 numbers. It doesn't matter how we flatten the array, as long as we're consistent between images. From this perspective, the MNIST images are just a bunch of points in a 784-dimensional vector space, with a very rich structure (warning: computationally intensive visualizations).

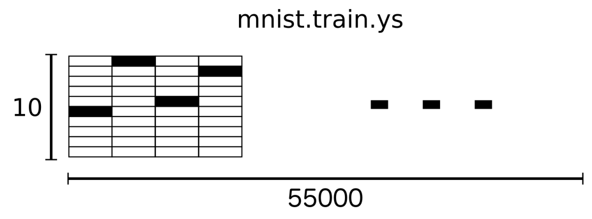
Flattening the data throws away information about the 2D structure of the image. Isn't that bad? Well, the best computer vision methods do exploit this structure, and we will in later tutorials. But the simple method we will be using here, a softmax regression (defined below), won't.

The result is that `mnist.train.images` is a tensor (an n-dimensional array) with a shape of `[55000, 784]`. The first dimension is an index into the list of images and the second dimension is the index for each pixel in each image. Each entry in the tensor is a pixel intensity between 0 and 1, for a particular pixel in a particular image.



Each image in MNIST has a corresponding label, a number between 0 and 9 representing the digit drawn in the image.

For the purposes of this tutorial, we're going to want our labels as "one-hot vectors". A one-hot vector is a vector which is 0 in most dimensions, and 1 in a single dimension. In this case, the nth digit will be represented as a vector which is 1 in the nth dimension. For example, 3 would be $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$. Consequently, `mnist.train.labels` is a `[55000, 10]` array of floats.



We're now ready to actually make our model!

Softmax Regressions

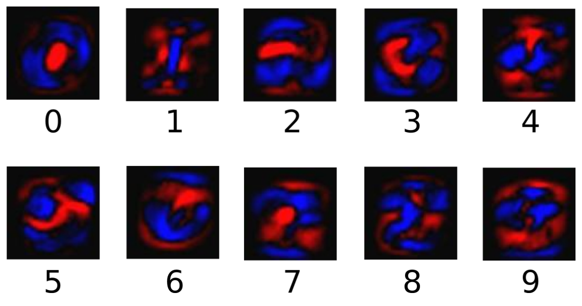
We know that every image in MNIST is of a handwritten digit between zero and nine. So there are only ten possible things that a given image can be. We want to be able to look at an image and give the probabilities for it being each digit. For example, our model might look at a picture of a nine and be 80% sure it's a nine, but give a 5% chance to it being an eight (because of the top loop) and a bit of probability to all the others because it isn't 100% sure.

This is a classic case where a softmax regression is a natural, simple model. If you want to assign probabilities to an object being one of several different things, softmax is the thing to do, because softmax gives us a list of values between 0 and 1 that add up to 1. Even later on, when we train more sophisticated models, the final step will be a layer of softmax.

A softmax regression has two steps: first we add up the evidence of our input being in certain classes, and then we convert that evidence into probabilities.

To tally up the evidence that a given image is in a particular class, we do a weighted sum of the pixel intensities. The weight is negative if that pixel having a high intensity is evidence against the image being in that class, and positive if it is evidence in favor.

The following diagram shows the weights one model learned for each of these classes. Red represents negative weights, while blue represents positive weights.



We also add some extra evidence called a bias. Basically, we want to be able to say that some things are more likely independent of the input. The result is that the evidence for a class i given an input x

is:

$$\text{evidence}_i = \sum_j W_{i,\,j} x_j + b_i$$

where W_i is the weights and b_i is the bias for class i, and j is an index for summing over the pixels in our input image x. We then convert the evidence tallies into our predicted probabilities y using the "softmax" function:

$$y = \text{softmax}(\text{evidence})$$

Here softmax is serving as an "activation" or "link" function, shaping the output of our linear function into the form we want -- in this case, a probability distribution over 10 cases. You can think of it as converting tallies of evidence into probabilities of our input being in each class. It's defined as:

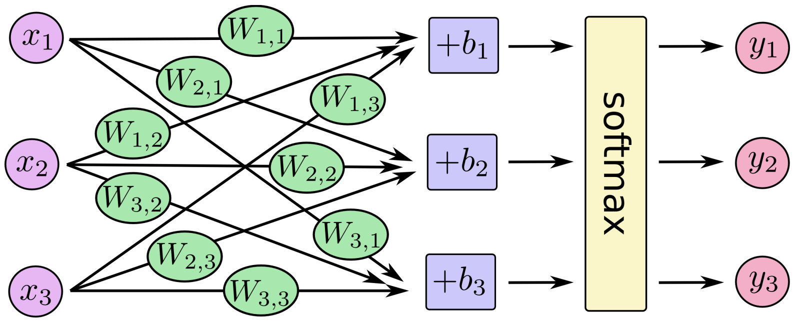
$$\text{softmax}(x) = \text{normalize}(\exp(x))$$

If you expand that equation out, you get:

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

But it's often more helpful to think of softmax the first way: exponentiating its inputs and then normalizing them. The exponentiation means that one more unit of evidence increases the weight given to any hypothesis multiplicatively. And conversely, having one less unit of evidence means that a hypothesis gets a fraction of its earlier weight. No hypothesis ever has zero or negative weight. Softmax then normalizes these weights, so that they add up to one, forming a valid probability distribution. (To get more intuition about the softmax function, check out the section on it in Michael Nielsen's book, complete with an interactive visualization.)

You can picture our softmax regression as looking something like the following, although with a lot more xs. For each output, we compute a weighted sum of the xs, add a bias, and then apply softmax.



If we write that out as equations, we get:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

We can "vectorize" this procedure, turning it into a matrix multiplication and vector addition. This is helpful for computational efficiency. (It's also a useful way to think.)

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right)$$

More compactly, we can just write:

$$y = \text{softmax}(Wx + b)$$

Now let's turn that into something that Tensorflow can use.

Implementing the Regression

To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations on GPUs or in a distributed manner, where there can be a high cost to transferring data.

TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, TensorFlow lets us describe a graph of interacting operations that run entirely outside Python. (Approaches like this can be seen in a few machine learning libraries.)

To use TensorFlow, first we need to import it.

```
import tensorflow as tf
```

We describe these interacting operations by manipulating symbolic variables. Let's create one:

```
x = tf.placeholder(tf.float32, [None, 784])
```

`x` isn't a specific value. It's a `placeholder`, a value that we'll input when we ask TensorFlow to run a computation. We want to be able to input any number of MNIST images, each flattened into a 784-dimensional vector. We represent this as a 2-D tensor of floating-point numbers, with a shape `[None, 784]`. (Here `None` means that a dimension can be of any length.)

We also need the weights and biases for our model. We could imagine treating these like additional inputs, but TensorFlow has an even better way to handle it: `Variable`. A `Variable` is a modifiable tensor that lives in TensorFlow's graph of interacting operations. It can be used and even modified by the computation. For machine learning applications, one generally has the model parameters be `Variable`s.

```
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

We create these `Variable`s by giving `tf.Variable` the initial value of the `Variable`: in this case, we initialize both `W` and `b` as tensors full of zeros. Since we are going to learn `W` and `b`, it doesn't matter very much what they initially are.

Notice that `W` has a shape of [784, 10] because we want to multiply the 784-dimensional image vectors by it to produce 10-dimensional vectors of evidence for the difference classes. `b` has a shape of [10] so we can add it to the output.

We can now implement our model. It only takes one line to define it!

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

First, we multiply `x` by `W` with the expression `tf.matmul(x, W)`. This is flipped from when we multiplied them in our equation, where we had Wx, as a small trick to deal with `x` being a 2D tensor with multiple inputs. We then add `b`, and finally apply `tf.nn.softmax`.

That's it. It only took us one line to define our model, after a couple short lines of setup. That isn't because TensorFlow is designed to make a softmax regression particularly easy: it's just a very flexible way to describe many kinds of numerical computations, from machine learning models to physics simulations. And once defined, our model can be run on different devices: your computer's CPU, GPUs, and even phones!

Training

In order to train our model, we need to define what it means for the model to be good. Well, actually, in machine learning we typically define what it means for a model to be bad. We call this the cost, or the loss, and it represents how far off our model is from our desired outcome. We try to minimize that error, and the smaller the error margin, the better our model is.

One very common, very nice function to determine the loss of a model is called "cross-entropy." Cross-entropy arises from thinking about information compressing codes in information theory but it winds up being an important idea in lots of areas, from gambling to machine learning. It's defined as:

$$H_{y'}(y) = -\sum_i y'_i \log(y_i)$$

Where y is our predicted probability distribution, and y' is the true distribution (the one-hot vector with the digit labels). In some rough sense, the cross-entropy is measuring how inefficient our predictions are for describing the truth. Going into more detail about cross-entropy is beyond the scope of this tutorial, but it's well worth understanding.

To implement cross-entropy we need to first add a new placeholder to input the correct answers:

```
y_ = tf.placeholder(tf.float32, [None, 10])
```

Then we can implement the cross-entropy function, $-\sum y' \log(y)$:

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[
```

First, `tf.log` computes the logarithm of each element of `y`. Next, we multiply each element of `y_` with the corresponding element of `tf.log(y)`. Then `tf.reduce_sum` adds the elements in the second dimension of y, due to the `reduction_indices=[1]` parameter. Finally, `tf.reduce_mean` computes the mean over all the examples in the batch.

(Note that in the source code, we don't use this formulation, because it is numerically unstable. Instead, we apply `tf.nn.softmax_cross_entropy_with_logits` on the unnormalized logits (e.g., we call `softmax_cross_entropy_with_logits` on `tf.matmul(x, W) + b`), because this more numerically stable function internally computes the softmax activation. In your code, consider using tf.nn.(sparse_)softmax_cross_entropy_with_logits instead).

Now that we know what we want our model to do, it's very easy to have TensorFlow train it to do so. Because TensorFlow knows the entire graph of your computations, it can automatically use the backpropagation algorithm to efficiently determine how your variables affect the loss you ask it to minimize. Then it can apply your choice of optimization algorithm to modify the variables and reduce the loss.

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

In this case, we ask TensorFlow to minimize `cross_entropy` using the gradient descent algorithm with a learning rate of 0.5. Gradient descent is a simple procedure, where TensorFlow simply shifts each variable a little bit in the direction that reduces the cost. But TensorFlow also provides many other optimization algorithms: using one is as simple as tweaking one line.

What TensorFlow actually does here, behind the scenes, is to add new operations to your graph which implement backpropagation and gradient descent. Then it gives you back a single operation which, when run, does a step of gradient descent training, slightly tweaking your variables to reduce the loss.

Now we have our model set up to train. One last thing before we launch it, we have to create an operation to initialize the variables we created. Note that this defines the operation but does not run it yet:

```
init = tf.initialize_all_variables()
```

We can now launch the model in a `Session`, and now we run the operation that initializes the variables:

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

Let's train -- we'll run the training step 1000 times!

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

Each step of the loop, we get a "batch" of one hundred random data points from our training set. We run `train_step` feeding in the batches data to replace the `placeholder`s.

Using small batches of random data is called stochastic training -- in this case, stochastic gradient descent. Ideally, we'd like to use all our data for every step of training because that would give us a better sense of what we should be doing, but that's expensive. So, instead, we use a different subset every time. Doing this is cheap and has much of the same benefit.

Evaluating Our Model

How well does our model do?

Well, first let's figure out where we predicted the correct label. `tf.argmax` is an extremely useful function which gives you the index of the highest entry in a tensor along some axis. For example, `tf.argmax(y,1)` is the label our model thinks is most likely for each input, while `tf.argmax(y_,1)` is the correct label. We can use `tf.equal` to check if our prediction matches the truth.

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

That gives us a list of booleans. To determine what fraction are correct, we cast to floating point numbers and then take the mean. For example, `[True, False, True, True]` would become `[1,0,1,1]` which would become `0.75`.

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

Finally, we ask for our accuracy on our test data.

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

This should be about 92%.

Is that good? Well, not really. In fact, it's pretty bad. This is because we're using a very simple model. With some small changes, we can get to 97%. The best models can get to over 99.7% accuracy! (For more information, have a look at this list of results.)

What matters is that we learned from this model. Still, if you're feeling a bit down about these results, check out the next tutorial where we do a lot better, and learn how to build more sophisticated models using TensorFlow!

Deep MNIST for Experts

TensorFlow is a powerful library for doing large-scale numerical computation. One of the tasks at which it excels is implementing and training deep neural networks. In this tutorial we will learn the basic building blocks of a TensorFlow model while constructing a deep convolutional MNIST classifier.

This introduction assumes familiarity with neural networks and the MNIST dataset. If you don't have a background with them, check out the introduction for beginners. Be sure to install TensorFlow before starting.

About this tutorial

The first part of this tutorial explains what is happening in the mnist_softmax.py code, which is a basic implementation of a Tensorflow model. The second part shows some ways to improve the accuracy.

You can copy and paste each code snippet from this tutorial into a Python environment, or you can choose to just read through the code.

What we will accomplish in this tutorial:

- Create a softmax regression function that is a model for recognizing MNIST digits, based on looking at every pixel in the image
- Use Tensorflow to train the model to recognize digits by having it "look" at thousands of examples (and run our first Tensorflow session to do so)
- Check the model's accuracy with our test data
- Build, train, and test a multilayer convolutional neural network to improve the results

Setup

Before we create our model, we will first load the MNIST dataset, and start a TensorFlow session.

Load MNIST Data

If you are copying and pasting in the code from this tutorial, start here with these two lines of code which will download and read in the data automatically:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

Here `mnist` is a lightweight class which stores the training, validation, and testing sets as NumPy arrays. It also provides a function for iterating through data minibatches, which we will use below.

Start TensorFlow InteractiveSession

TensorFlow relies on a highly efficient C++ backend to do its computation. The connection to this backend is called a session. The common usage for TensorFlow programs is to first create a graph and then launch it in a session.

Here we instead use the convenient `InteractiveSession` class, which makes TensorFlow more flexible about how you structure your code. It allows you to interleave operations which build a computation graph with ones that run the graph. This is particularly convenient when working in interactive contexts like IPython. If you are not using an `InteractiveSession`, then you should build the entire computation graph before starting a session and launching the graph.

```
import tensorflow as tf
sess = tf.InteractiveSession()
```

Computation Graph

To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations on GPUs or in a distributed manner, where there can be a high cost to transferring data.

TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, TensorFlow lets us describe a graph of interacting operations that run entirely outside Python. This approach is similar to that used in Theano or Torch.

The role of the Python code is therefore to build this external computation graph, and to dictate which parts of the computation graph should be run. See the Computation Graph section of Basic Usage for more detail.

Build a Softmax Regression Model

In this section we will build a softmax regression model with a single linear layer. In the next section, we will extend this to the case of softmax regression with a multilayer convolutional network.

Placeholders

We start building the computation graph by creating nodes for the input images and target output classes.

```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

Here `x` and `y_` aren't specific values. Rather, they are each a `placeholder` -- a value that we'll input when we ask TensorFlow to run a computation.

The input images `x` will consist of a 2d tensor of floating point numbers. Here we assign it a `shape` of `[None, 784]`, where `784` is the dimensionality of a single flattened 28 by 28 pixel MNIST image, and `None` indicates that the first dimension, corresponding to the batch size, can be of any size. The target output classes `y_` will also consist of a 2d tensor, where each row is a one-hot 10-dimensional vector indicating which digit class (zero through nine) the corresponding MNIST image belongs to.

The `shape` argument to `placeholder` is optional, but it allows TensorFlow to automatically catch bugs stemming from inconsistent tensor shapes.

Variables

We now define the weights `W` and biases `b` for our model. We could imagine treating these like additional inputs, but TensorFlow has an even better way to handle them: `Variable`. A `Variable` is a value that lives in TensorFlow's computation graph. It can be used and even modified by the computation. In machine learning applications, one generally has the model parameters be `Variable`s.

```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

We pass the initial value for each parameter in the call to `tf.Variable`. In this case, we initialize both `W` and `b` as tensors full of zeros. `W` is a 784x10 matrix (because we have 784 input features and 10 outputs) and `b` is a 10-dimensional vector (because we have 10 classes).

Before `Variable`s can be used within a session, they must be initialized using that session. This step takes the initial values (in this case tensors full of zeros) that have already been specified, and assigns them to each `Variable`. This can be done for all `Variables` at once:

```
sess.run(tf.initialize_all_variables())
```

Predicted Class and Loss Function

We can now implement our regression model. It only takes one line! We multiply the vectorized input images `x` by the weight matrix `W`, add the bias `b`.

```
y = tf.matmul(x,W) + b
```

We can specify a loss function just as easily. Loss indicates how bad the model's prediction was on a single example; we try to minimize that while training across all the examples. Here, our loss function is the cross-entropy between the target and the softmax activation function applied to the model's prediction. As in the beginners tutorial, we use the stable formulation:

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, y_))
```

Note that `tf.nn.softmax_cross_entropy_with_logits` internally applies the softmax on the model's unnormalized model prediction and sums across all classes, and `tf.reduce_mean` takes the average over these sums.

Train the Model

Now that we have defined our model and training loss function, it is straightforward to train using TensorFlow. Because TensorFlow knows the entire computation graph, it can use automatic differentiation to find the gradients of the loss with respect to each of the variables. TensorFlow has a variety of built-in optimization algorithms. For this example, we will use steepest gradient descent, with a step length of 0.5, to descend the cross entropy.

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

What TensorFlow actually did in that single line was to add new operations to the computation graph. These operations included ones to compute gradients, compute parameter update steps, and apply update steps to the parameters.

The returned operation `train_step`, when run, will apply the gradient descent updates to the parameters. Training the model can therefore be accomplished by repeatedly running `train_step`.

```
for i in range(1000):
  batch = mnist.train.next_batch(100)
  train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

We load 100 training examples in each training iteration. We then run the `train_step` operation, using `feed_dict` to replace the `placeholder` tensors `x` and `y_` with the training examples. Note that you can replace any tensor in your computation graph using `feed_dict` -- it's not restricted to just `placeholder`s.

Evaluate the Model

How well did our model do?

First we'll figure out where we predicted the correct label. `tf.argmax` is an extremely useful function which gives you the index of the highest entry in a tensor along some axis. For example, `tf.argmax(y,1)` is the label our model thinks is most likely for each input, while `tf.argmax(y_,1)` is the true label. We can use `tf.equal` to check if our prediction matches the truth.

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

That gives us a list of booleans. To determine what fraction are correct, we cast to floating point numbers and then take the mean. For example, `[True, False, True, True]` would become `[1,0,1,1]` which would become `0.75`.

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

Finally, we can evaluate our accuracy on the test data. This should be about 92% correct.

```
print(accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

Build a Multilayer Convolutional Network

Getting 92% accuracy on MNIST is bad. It's almost embarrassingly bad. In this section, we'll fix that, jumping from a very simple model to something moderately sophisticated: a small convolutional neural network. This will get us to around 99.2% accuracy -- not state of the art, but respectable.

Weight Initialization

To create this model, we're going to need to create a lot of weights and biases. One should generally initialize weights with a small amount of noise for symmetry breaking, and to prevent 0 gradients. Since we're using ReLU neurons, it is also good practice to initialize them with a slightly positive initial bias to avoid "dead neurons". Instead of doing this repeatedly while we build the model, let's create two handy functions to do it for us.

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

Convolution and Pooling

TensorFlow also gives us a lot of flexibility in convolution and pooling operations. How do we handle the boundaries? What is our stride size? In this example, we're always going to choose the vanilla version. Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input. Our pooling is plain old max pooling over 2x2 blocks. To keep our code cleaner, let's also abstract those operations into functions.

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
```

First Convolutional Layer

We can now implement our first layer. It will consist of convolution, followed by max pooling. The convolutional will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of `[5, 5, 1, 32]`. The first two dimensions are the patch size, the next is the number of input channels, and the last is the number of output channels. We will also have a bias vector with a component for each output channel.

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

To apply the layer, we first reshape `x` to a 4d tensor, with the second and third dimensions corresponding to image width and height, and the final dimension corresponding to the number of color channels.

```
x_image = tf.reshape(x, [-1,28,28,1])
```

We then convolve `x_image` with the weight tensor, add the bias, apply the ReLU function, and finally max pool.

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

Second Convolutional Layer

In order to build a deep network, we stack several layers of this type. The second layer will have 64 features for each 5x5 patch.

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

Densely Connected Layer

Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons to allow processing on the entire image. We reshape the tensor from the pooling layer into a batch of vectors, multiply by a weight matrix, add a bias, and apply a ReLU.

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

Dropout

To reduce overfitting, we will apply dropout before the readout layer. We create a `placeholder` for the probability that a neuron's output is kept during dropout. This allows us to turn dropout on during training, and turn it off during testing. TensorFlow's `tf.nn.dropout` op automatically handles scaling neuron outputs in addition to masking them, so dropout just works without any additional scaling.[1]

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Readout Layer

Finally, we add a layer, just like for the one layer softmax regression above.

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

Train and Evaluate the Model

How well does this model do? To train and evaluate it we will use code that is nearly identical to that for the simple one layer SoftMax network above.

The differences are that:

- We will replace the steepest gradient descent optimizer with the more sophisticated ADAM optimizer.

- We will include the additional parameter `keep_prob` in `feed_dict` to control the dropout rate.

- We will add logging to every 100th iteration in the training process.

Feel free to go ahead and run this code, but it does 20,000 training iterations and may take a while (possibly up to half an hour), depending on your processor.

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_conv, y_
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

The final test set accuracy after running this code should be approximately 99.2%.

We have learned how to quickly and easily build, train, and evaluate a fairly sophisticated deep learning model using TensorFlow.

1: For this small convolutional network, performance is actually nearly identical with and without dropout. Dropout is often very effective at reducing overfitting, but it is most useful when training very large neural networks. ↩

Basic Usage

To use TensorFlow you need to understand how TensorFlow:

- Represents computations as graphs.
- Executes graphs in the context of `Sessions`.
- Represents data as tensors.
- Maintains state with `Variables`.
- Uses feeds and fetches to get data into and out of arbitrary operations.

Overview

TensorFlow is a programming system in which you represent computations as graphs. Nodes in the graph are called ops (short for operations). An op takes zero or more `Tensors`, performs some computation, and produces zero or more `Tensors`. A `Tensor` is a typed multi-dimensional array. For example, you can represent a mini-batch of images as a 4-D array of floating point numbers with dimensions `[batch, height, width, channels]`.

A TensorFlow graph is a description of computations. To compute anything, a graph must be launched in a `Session`. A `Session` places the graph ops onto `Devices`, such as CPUs or GPUs, and provides methods to execute them. These methods return tensors produced by ops as numpy `ndarray` objects in Python, and as `tensorflow::Tensor` instances in C and C++.

The computation graph

TensorFlow programs are usually structured into a construction phase, that assembles a graph, and an execution phase that uses a session to execute ops in the graph.

For example, it is common to create a graph to represent and train a neural network in the construction phase, and then repeatedly execute a set of training ops in the graph in the execution phase.

TensorFlow can be used from C, C++, and Python programs. It is presently much easier to use the Python library to assemble graphs, as it provides a large set of helper functions not available in the C and C++ libraries.

The session libraries have equivalent functionalities for the three languages.

Building the graph

To build a graph start with ops that do not need any input (source ops), such as `Constant`, and pass their output to other ops that do computation.

The ops constructors in the Python library return objects that stand for the output of the constructed ops. You can pass these to other ops constructors to use as inputs.

The TensorFlow Python library has a default graph to which ops constructors add nodes. The default graph is sufficient for many applications. See the Graph class documentation for how to explicitly manage multiple graphs.

```
import tensorflow as tf

# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)
```

The default graph now has three nodes: two `constant()` ops and one `matmul()` op. To actually multiply the matrices, and get the result of the multiplication, you must launch the graph in a session.

Launching the graph in a session

Launching follows construction. To launch a graph, create a `Session` object. Without arguments the session constructor launches the default graph.

See the Session class for the complete session API.

```
# Launch the default graph.
sess = tf.Session()

# To run the matmul op we call the session 'run()' method, passing 'product'
# which represents the output of the matmul op.  This indicates to the call
# that we want to get the output of the matmul op back.
#
# All inputs needed by the op are run automatically by the session.  They
# typically are run in parallel.
#
# The call 'run(product)' thus causes the execution of three ops in the
# graph: the two constants and matmul.
#
# The output of the op is returned in 'result' as a numpy `ndarray` object.
result = sess.run(product)
print(result)
# ==> [[ 12.]]

# Close the Session when we're done.
sess.close()
```

Sessions should be closed to release resources. You can also enter a `Session` with a "with" block. The `Session` closes automatically at the end of the `with` block.

```
with tf.Session() as sess:
  result = sess.run([product])
  print(result)
```

The TensorFlow implementation translates the graph definition into executable operations distributed across available compute resources, such as the CPU or one of your computer's GPU cards. In general you do not have to specify CPUs or GPUs explicitly. TensorFlow uses your first GPU, if you have one, for as many operations as possible.

If you have more than one GPU available on your machine, to use a GPU beyond the first you must assign ops to it explicitly. Use `with...Device` statements to specify which CPU or GPU to use for operations:

```
with tf.Session() as sess:
  with tf.device("/gpu:1"):
    matrix1 = tf.constant([[3., 3.]])
    matrix2 = tf.constant([[2.],[2.]])
    product = tf.matmul(matrix1, matrix2)
    ...
```

Devices are specified with strings. The currently supported devices are:

- `"/cpu:0"`: The CPU of your machine.
- `"/gpu:0"`: The GPU of your machine, if you have one.
- `"/gpu:1"`: The second GPU of your machine, etc.

See Using GPUs for more information about GPUs and TensorFlow.

Launching the graph in a distributed session

To create a TensorFlow cluster, launch a TensorFlow server on each of the machines in the cluster. When you instantiate a Session in your client, you pass it the network location of one of the machines in the cluster:

```
with tf.Session("grpc://example.org:2222") as sess:
  # Calls to sess.run(...) will be executed on the cluster.
  ...
```

This machine becomes the master for the session. The master distributes the graph across other machines in the cluster (workers), much as the local implementation distributes the graph across available compute resources within a machine.

You can use "with tf.device():" statements to directly specify workers for particular parts of the graph:

```
with tf.device("/job:ps/task:0"):
  weights = tf.Variable(...)
  biases = tf.Variable(...)
```

See the Distributed TensorFlow How To for more information about distributed sessions and clusters.

Interactive Usage

The Python examples in the documentation launch the graph with a `Session` and use the `Session.run()` method to execute operations.

For ease of use in interactive Python environments, such as IPython you can instead use the `InteractiveSession` class, and the `Tensor.eval()` and `Operation.run()` methods. This avoids having to keep a variable holding the session.

```
# Enter an interactive TensorFlow Session.
import tensorflow as tf
sess = tf.InteractiveSession()

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])

# Initialize 'x' using the run() method of its initializer op.
x.initializer.run()

# Add an op to subtract 'a' from 'x'.  Run it and print the result
sub = tf.sub(x, a)
print(sub.eval())
# ==> [-2. -1.]

# Close the Session when we're done.
sess.close()
```

Tensors

TensorFlow programs use a tensor data structure to represent all data -- only tensors are passed between operations in the computation graph. You can think of a TensorFlow tensor as an n-dimensional array or list. A tensor has a static type, a rank, and a shape. To learn more about how TensorFlow handles these concepts, see the Rank, Shape, and Type reference.

Variables

Variables maintain state across executions of the graph. The following example shows a variable serving as a simple counter. See Variables for more details.

```
# Create a Variable, that will be initialized to the scalar value 0.
state = tf.Variable(0, name="counter")

# Create an Op to add one to `state`.

one = tf.constant(1)
new_value = tf.add(state, one)
update = tf.assign(state, new_value)

# Variables must be initialized by running an `init` Op after having
# launched the graph.  We first have to add the `init` Op to the graph.
init_op = tf.initialize_all_variables()

# Launch the graph and run the ops.
with tf.Session() as sess:
  # Run the 'init' op
  sess.run(init_op)
  # Print the initial value of 'state'
  print(sess.run(state))
  # Run the op that updates 'state' and print 'state'.
  for _ in range(3):
    sess.run(update)
    print(sess.run(state))

# output:

# 0
# 1
# 2
# 3
```

The `assign()` operation in this code is a part of the expression graph just like the `add()` operation, so it does not actually perform the assignment until `run()` executes the expression.

You typically represent the parameters of a statistical model as a set of Variables. For example, you would store the weights for a neural network as a tensor in a Variable. During training you update this tensor by running a training graph repeatedly.

Fetches

To fetch the outputs of operations, execute the graph with a `run()` call on the `Session` object and pass in the tensors to retrieve. In the previous example we fetched the single node `state`, but you can also fetch multiple tensors:

```
input1 = tf.constant([3.0])
input2 = tf.constant([2.0])
input3 = tf.constant([5.0])
intermed = tf.add(input2, input3)
mul = tf.mul(input1, intermed)

with tf.Session() as sess:
  result = sess.run([mul, intermed])
  print(result)

# output:
# [array([ 21.], dtype=float32), array([ 7.], dtype=float32)]
```

All the ops needed to produce the values of the requested tensors are run once (not once per requested tensor).

Feeds

The examples above introduce tensors into the computation graph by storing them in `Constants` and `Variables`. TensorFlow also provides a feed mechanism for patching a tensor directly into any operation in the graph.

A feed temporarily replaces the output of an operation with a tensor value. You supply feed data as an argument to a `run()` call. The feed is only used for the run call to which it is passed. The most common use case involves designating specific operations to be "feed" operations by using tf.placeholder() to create them:

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
```

```
output = tf.mul(input1, input2)

with tf.Session() as sess:
  print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))

# output:
# [array([ 14.], dtype=float32)]
```

A `placeholder()` operation generates an error if you do not supply a feed for it. See the MNIST fully-connected feed tutorial (source code) for a larger-scale example of feeds.