# Getting Started

For a brief overview of TensorFlow programming fundamentals, see the following guide:

* [Getting Started with TensorFlow](https://www.tensorflow.org/get_started/get_started)

MNIST has become the canonical dataset for trying out a new machine learning toolkit. We offer three guides that each demonstrate a different approach to training an MNIST model on TensorFlow:

* [MNIST for ML Beginners](https://www.tensorflow.org/get_started/mnist/beginners), which introduces MNIST through the high-level API.
* [Deep MNIST for Experts](https://www.tensorflow.org/get_started/mnist/pros), which is more-in depth than "MNIST for ML Beginners," and assumes some familiarity with machine learning concepts.
* [TensorFlow Mechanics 101](https://www.tensorflow.org/get_started/mnist/mechanics), which introduces MNIST through the low-level API.

For developers new to TensorFlow, the high-level API is a good place to start. To learn about the high-level API, read the following guides:

* [tf.contrib.learn Quickstart](https://www.tensorflow.org/get_started/tflearn), which introduces this API.
* [Building Input Functions with tf.contrib.learn](https://www.tensorflow.org/get_started/input_fn), which takes you into a somewhat more sophisticated use of this API.
* [Logging and Monitoring Basics with tf.contrib.learn](https://www.tensorflow.org/get_started/monitors), which explains how to audit the progress of model training.

TensorBoard is a utility to visualize different aspects of machine learning. The following guides explain how to use TensorBoard:

* [TensorBoard: Visualizing Learning](https://www.tensorflow.org/get_started/summaries_and_tensorboard), which gets you started.
* [TensorBoard: Embedding Visualization](https://www.tensorflow.org/get_started/embedding_viz), which demonstrates how to view and interact with high-dimensional data, such as embeddings.
* [TensorBoard: Graph Visualization](https://www.tensorflow.org/get_started/graph_viz), which explains how to visualize the computational graph. Graph visualization is typically more useful for programmers using the low-level API.

# Getting Started With TensorFlow

This guide gets you started programming in TensorFlow. Before using this guide, [install TensorFlow](https://www.tensorflow.org/install/index). To get the most out of this guide, you should know the following:

* How to program in Python.
* At least a little bit about arrays.
* Ideally, something about machine learning. However, if you know little or nothing about machine learning, then this is still the first guide you should read.

TensorFlow provides multiple APIs. The lowest level API--TensorFlow Core-- provides you with complete programming control. We recommend TensorFlow Core for machine learning researchers and others who require fine levels of control over their models. The higher level APIs are built on top of TensorFlow Core. These higher level APIs are typically easier to learn and use than TensorFlow Core. In addition, the higher level APIs make repetitive tasks easier and more consistent between different users. A high-level API like tf.contrib.learn helps you manage data sets, estimators, training and inference. Note that a few of the high-level TensorFlow APIs--those whose method names contain contrib-- are still in development. It is possible that some contrib methods will change or become obsolete in subsequent TensorFlow releases.

This guide begins with a tutorial on TensorFlow Core. Later, we demonstrate how to implement the same model in tf.contrib.learn. Knowing TensorFlow Core principles will give you a great mental model of how things are working internally when you use the more compact higher level API.

# Tensors

The central unit of data in TensorFlow is the **tensor**. A tensor consists of a set of primitive values shaped into an array of any number of dimensions. A tensor's **rank** is its number of dimensions. Here are some examples of tensors:

3 # a rank 0 tensor; this is a scalar with shape []  
[1. ,2., 3.] # a rank 1 tensor; this is a vector with shape [3]  
[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]  
[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]

## TensorFlow Core tutorial

### Importing TensorFlow

The canonical import statement for TensorFlow programs is as follows:

import tensorflow as tf

This gives Python access to all of TensorFlow's classes, methods, and symbols. Most of the documentation assumes you have already done this.

### The Computational Graph

You might think of TensorFlow Core programs as consisting of two discrete sections:

1. Building the computational graph.
2. Running the computational graph.

A **computational graph** is a series of TensorFlow operations arranged into a graph of nodes. Let's build a simple computational graph. Each node takes zero or more tensors as inputs and produces a tensor as an output. One type of node is a constant. Like all TensorFlow constants, it takes no inputs, and it outputs a value it stores internally. We can create two floating point Tensors node1 and node2 as follows:

node1 = tf.constant(3.0, tf.float32)  
node2 = tf.constant(4.0) # also tf.float32 implicitly  
print(node1, node2)

The final print statement produces

Tensor("Const:0", shape=(), dtype=float32) Tensor("Const\_1:0", shape=(), dtype=float32)

Notice that printing the nodes does not output the values 3.0 and 4.0 as you might expect. Instead, they are nodes that, when evaluated, would produce 3.0 and 4.0, respectively. To actually evaluate the nodes, we must run the computational graph within a **session**. A session encapsulates the control and state of the TensorFlow runtime.

The following code creates a Session object and then invokes its run method to run enough of the computational graph to evaluate node1 and node2. By running the computational graph in a session as follows:

sess = tf.Session()  
print(sess.run([node1, node2]))

we see the expected values of 3.0 and 4.0:

[3.0, 4.0]

We can build more complicated computations by combining Tensor nodes with operations (Operations are also nodes.). For example, we can add our two constant nodes and produce a new graph as follows:

node3 = tf.add(node1, node2)  
print("node3: ", node3)  
print("sess.run(node3): ",sess.run(node3))

The last two print statements produce

node3:  Tensor("Add\_2:0", shape=(), dtype=float32)  
sess.run(node3):  7.0

TensorFlow provides a utility called TensorBoard that can display a picture of the computational graph. Here is a screenshot showing how TensorBoard visualizes the graph:

As it stands, this graph is not especially interesting because it always produces a constant result. A graph can be parameterized to accept external inputs, known as **placeholders**. A **placeholder** is a promise to provide a value later.

a = tf.placeholder(tf.float32)  
b = tf.placeholder(tf.float32)  
adder\_node = a + b  # + provides a shortcut for tf.add(a, b)

The preceding three lines are a bit like a function or a lambda in which we define two input parameters (a and b) and then an operation on them. We can evaluate this graph with multiple inputs by using the feed\_dict parameter to specify Tensors that provide concrete values to these placeholders:

print(sess.run(adder\_node, {a: 3, b:4.5}))  
print(sess.run(adder\_node, {a: [1,3], b: [2, 4]}))

resulting in the output

7.5  
[ 3.  7.]

In TensorBoard, the graph looks like this:

We can make the computational graph more complex by adding another operation. For example,

add\_and\_triple = adder\_node \* 3.  
print(sess.run(add\_and\_triple, {a: 3, b:4.5}))

produces the output

22.5

The preceding computational graph would look as follows in TensorBoard:

In machine learning we will typically want a model that can take arbitrary inputs, such as the one above. To make the model trainable, we need to be able to modify the graph to get new outputs with the same input. **Variables**allow  us to add trainable parameters to a graph. They are constructed with a type and initial value:

W = tf.Variable([.3], tf.float32)  
b = tf.Variable([-.3], tf.float32)  
x = tf.placeholder(tf.float32)  
linear\_model = W \* x + b

Constants are initialized when you call tf.constant, and their value can never change. By contrast, variables are not initialized when you call tf.Variable. To initialize all the variables in a TensorFlow program, you must explicitly call a special operation as follows:

init = tf.global\_variables\_initializer()  
sess.run(init)

It is important to realize init is a handle to the TensorFlow sub-graph that initializes all the global variables. Until we call sess.run, the variables are uninitialized.

Since x is a placeholder, we can evaluate linear\_model for several values of x simultaneously as follows:

print(sess.run(linear\_model, {x:[1,2,3,4]}))

to produce the output

[ 0.          0.30000001  0.60000002  0.90000004]

We've created a model, but we don't know how good it is yet. To evaluate the model on training data, we need a yplaceholder to provide the desired values, and we need to write a loss function.

A loss function measures how far apart the current model is from the provided data. We'll use a standard loss model for linear regression, which sums the squares of the deltas between the current model and the provided data. linear\_model - y creates a vector where each element is the corresponding example's error delta. We call tf.square to square that error. Then, we sum all the squared errors to create a single scalar that abstracts the error of all examples using tf.reduce\_sum:

y = tf.placeholder(tf.float32)  
squared\_deltas = tf.square(linear\_model - y)  
loss = tf.reduce\_sum(squared\_deltas)  
print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))

producing the loss value

23.66

We could improve this manually by reassigning the values of W and b to the perfect values of -1 and 1. A variable is initialized to the value provided to tf.Variable but can be changed using operations like tf.assign. For example, W=-1 and b=1 are the optimal parameters for our model. We can change W and baccordingly:

fixW = tf.assign(W, [-1.])  
fixb = tf.assign(b, [1.])  
sess.run([fixW, fixb])  
print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))

The final print shows the loss now is zero.

0.0

We guessed the "perfect" values of W and b, but the whole point of machine learning is to find the correct model parameters automatically. We will show how to accomplish this in the next section.

## tf.train API

A complete discussion of machine learning is out of the scope of this tutorial. However, TensorFlow provides **optimizers** that slowly change each variable in order to minimize the loss function. The simplest optimizer is **gradient descent**. It modifies each variable according to the magnitude of the derivative of loss with respect to that variable. In general, computing symbolic derivatives manually is tedious and error-prone. Consequently, TensorFlow can automatically produce derivatives given only a description of the model using the function tf.gradients. For simplicity, optimizers typically do this for you. For example,

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

sess.run(init) # reset values to incorrect defaults.  
for i in range(1000):  
  sess.run(train, {x:[1,2,3,4], y:[0,-1,-2,-3]})  
  
print(sess.run([W, b]))

results in the final model parameters:

[array([-0.9999969], dtype=float32), array([ 0.99999082],  
 dtype=float32)]

Now we have done actual machine learning! Although doing this simple linear regression doesn't require much TensorFlow core code, more complicated models and methods to feed data into your model necessitate more code. Thus TensorFlow provides higher level abstractions for common patterns, structures, and functionality. We will learn how to use some of these abstractions in the next section.

### Complete program

The completed trainable linear regression model is shown here:

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))

When run, it produces

W: [-0.9999969] b: [ 0.99999082] loss: 5.69997e-11

This more complicated program can still be visualized in TensorBoard

## tf.contrib.learn

tf.contrib.learn is a high-level TensorFlow library that simplifies the mechanics of machine learning, including the following:

* running training loops
* running evaluation loops
* managing data sets
* managing feeding

tf.contrib.learn defines many common models.

### Basic usage

Notice how much simpler the linear regression program becomes with tf.contrib.learn:

import tensorflow as tf  
# NumPy is often used to load, manipulate and preprocess data.  
import numpy as np  
  
# Declare list of features. We only have one real-valued feature. There are many  
# other types of columns that are more complicated and useful.  
features = [tf.contrib.layers.real\_valued\_column("x", dimension=1)]  
  
# An estimator is the front end to invoke training (fitting) and evaluation  
# (inference). There are many predefined types like linear regression,  
# logistic regression, linear classification, logistic classification, and  
# many neural network classifiers and regressors. The following code  
# provides an estimator that does linear regression.  
estimator = tf.contrib.learn.LinearRegressor(feature\_columns=features)  
  
# TensorFlow provides many helper methods to read and set up data sets.  
# Here we use `numpy\_input\_fn`. We have to tell the function how many batches  
# of data (num\_epochs) we want and how big each batch should be.  
x = np.array([1., 2., 3., 4.])  
y = np.array([0., -1., -2., -3.])  
input\_fn = tf.contrib.learn.io.numpy\_input\_fn({"x":x}, y, batch\_size=4,  
                                              num\_epochs=1000)  
  
# We can invoke 1000 training steps by invoking the `fit` method and passing the  
# training data set.  
estimator.fit(input\_fn=input\_fn, steps=1000)  
  
# Here we evaluate how well our model did. In a real example, we would want  
# to use a separate validation and testing data set to avoid overfitting.  
print(estimator.evaluate(input\_fn=input\_fn))

When run, it produces

    {'global\_step': 1000, 'loss': 1.9650059e-11}

### A custom model

tf.contrib.learn does not lock you into its predefined models. Suppose we wanted to create a custom model that is not built into TensorFlow. We can still retain the high level abstraction of data set, feeding, training, etc. oftf.contrib.learn. For illustration, we will show how to implement our own equivalent model to LinearRegressor using our knowledge of the lower level TensorFlow API.

To define a custom model that works with tf.contrib.learn, we need to use tf.contrib.learn.Estimator. tf.contrib.learn.LinearRegressor is actually a sub-class of tf.contrib.learn.Estimator. Instead of sub-classing Estimator, we simply provide Estimator a function model\_fn that tells tf.contrib.learnhow it can evaluate predictions, training steps, and loss. The code is as follows:

import numpy as np  
import tensorflow as tf  
# Declare list of features, we only have one real-valued feature  
def model(features, labels, mode):  
  # Build a linear model and predict values  
  W = tf.get\_variable("W", [1], dtype=tf.float64)  
  b = tf.get\_variable("b", [1], dtype=tf.float64)  
  y = W\*features['x'] + b  
  # Loss sub-graph  
  loss = tf.reduce\_sum(tf.square(y - labels))  
  # Training sub-graph  
  global\_step = tf.train.get\_global\_step()  
  optimizer = tf.train.GradientDescentOptimizer(0.01)  
  train = tf.group(optimizer.minimize(loss),  
                   tf.assign\_add(global\_step, 1))  
  # ModelFnOps connects subgraphs we built to the  
  # appropriate functionality.  
  return tf.contrib.learn.ModelFnOps(  
      mode=mode, predictions=y,  
      loss=loss,  
      train\_op=train)  
  
estimator = tf.contrib.learn.Estimator(model\_fn=model)  
# define our data set  
x = np.array([1., 2., 3., 4.])  
y = np.array([0., -1., -2., -3.])  
input\_fn = tf.contrib.learn.io.numpy\_input\_fn({"x": x}, y, 4, num\_epochs=1000)  
  
# train  
estimator.fit(input\_fn=input\_fn, steps=1000)  
# evaluate our model  
print(estimator.evaluate(input\_fn=input\_fn, steps=10))

When run, it produces

{'loss': 5.9819476e-11, 'global\_step': 1000}

Notice how the contents of the custom model() function are very similar to our manual model training loop from the lower level API.

## Next steps

Now you have a working knowledge of the basics of TensorFlow. We have several more tutorials that you can look at to learn more. If you are a beginner in machine learning see [MNIST for beginners](https://www.tensorflow.org/get_started/mnist/beginners), otherwise see [Deep MNIST for experts](https://www.tensorflow.org/get_started/mnist/pros).

# 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*](https://www.tensorflow.org/get_started/mnist/pros). Be sure to[*install TensorFlow*](https://www.tensorflow.org/install/index)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](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/mnist/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](http://yann.lecun.com/exdb/mnist/). 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](http://colah.github.io/posts/2014-10-Visualizing-MNIST/) (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:

evidencei=∑jWi, jxj+bi

where Wi is the weights and bi 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=softmax(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:

softmax(x)=normalize(exp⁡(x))

If you expand that equation out, you get:

softmax(x)i=exp⁡(xi)∑jexp⁡(xj)

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](http://neuralnetworksanddeeplearning.com/chap3.html#softmax) 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:

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.)

More compactly, we can just write:

y=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](http://www.numpy.org/) 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 Nonemeans 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 Variables.

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

We create these Variables 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:

Hy′(y)=−∑iyi′log⁡(yi)

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](http://colah.github.io/posts/2015-09-Visual-Information/).

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, −∑y′log⁡(y):

cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(y), reduction\_indices=[1]))

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 applytf.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.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](http://colah.github.io/posts/2015-08-Backprop/) 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](https://en.wikipedia.org/wiki/Gradient_descent) 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](https://www.tensorflow.org/api_guides/python/train#Optimizers): 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.

We can now launch the model in an InteractiveSession:

sess = tf.InteractiveSession()

We first have to create an operation to initialize the variables we created:

tf.global\_variables\_initializer().run()

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

for \_ 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 placeholders.

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](http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html).)

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](https://www.tensorflow.org/get_started/mnist/pros) 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*](https://www.tensorflow.org/get_started/mnist/beginners). Be sure to[*install TensorFlow*](https://www.tensorflow.org/install/index)before starting.

## About this tutorial

The first part of this tutorial explains what is happening in the [mnist\_softmax.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/mnist/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](https://www.tensorflow.org/get_started/index#the_computational_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](https://www.tensorflow.org/get_started/index#the_computational_graph).

import tensorflow as tf  
sess = tf.InteractiveSession()

#### Computation Graph

To do efficient numerical computing in Python, we typically use libraries like [NumPy](http://www.numpy.org/) 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](https://www.tensorflow.org/get_started/index#the_computational_graph) section of [Getting Started](https://www.tensorflow.org/get_started/index) 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 Variables.

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 bas 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 Variables 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 eachVariable. This can be done for all Variables at once:

sess.run(tf.global\_variables\_initializer())

### Predicted Class and Loss Function

We can now implement our regression model. It only takes one line! We multiply the vectorized input images xby 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(labels=y\_, logits=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](https://www.tensorflow.org/api_guides/python/train#optimizers). 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 \_ 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 placeholders.

### 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.equalto 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](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))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 convolution 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. The max\_pool\_2x2 method will reduce the image size to 14x14.

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](https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf) 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](https://www.tensorflow.org/get_started/mnist/pros#f1)

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(labels=y\_, logits=y\_conv))  
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.global\_variables\_initializer())  
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

# TensorFlow Mechanics 101

Code: [tensorflow/examples/tutorials/mnist/](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/mnist/)

The goal of this tutorial is to show how to use TensorFlow to train and evaluate a simple feed-forward neural network for handwritten digit classification using the (classic) MNIST data set. The intended audience for this tutorial is experienced machine learning users interested in using TensorFlow.

These tutorials are not intended for teaching Machine Learning in general.

Please ensure you have followed the instructions to [install TensorFlow](https://www.tensorflow.org/install/index).

## Tutorial Files

This tutorial references the following files:

| File | Purpose |
| --- | --- |
| [mnist.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/mnist/mnist.py) | The code to build a fully-connected MNIST model. |
| [fully\_connected\_feed.py](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/mnist/fully_connected_feed.py) | The main code to train the built MNIST model against the downloaded dataset using a feed dictionary. |

Simply run the fully\_connected\_feed.py file directly to start training:

python fully\_connected\_feed.py

## Prepare the Data

MNIST is a classic problem in machine learning. The problem is to look at greyscale 28x28 pixel images of handwritten digits and determine which digit the image represents, for all the digits from zero to nine.

For more information, refer to [Yann LeCun's MNIST page](http://yann.lecun.com/exdb/mnist/) or [Chris Olah's visualizations of MNIST](http://colah.github.io/posts/2014-10-Visualizing-MNIST/).

### Download

At the top of the run\_training() method, the input\_data.read\_data\_sets() function will ensure that the correct data has been downloaded to your local training folder and then unpack that data to return a dictionary of DataSet instances.

data\_sets = input\_data.read\_data\_sets(FLAGS.train\_dir, FLAGS.fake\_data)

**NOTE**: The fake\_data flag is used for unit-testing purposes and may be safely ignored by the reader.

| Dataset | Purpose |
| --- | --- |
| data\_sets.train | 55000 images and labels, for primary training. |
| data\_sets.validation | 5000 images and labels, for iterative validation of training accuracy. |
| data\_sets.test | 10000 images and labels, for final testing of trained accuracy. |

### Inputs and Placeholders

The placeholder\_inputs() function creates two [tf.placeholder](https://www.tensorflow.org/api_docs/python/tf/placeholder) ops that define the shape of the inputs, including the batch\_size, to the rest of the graph and into which the actual training examples will be fed.

images\_placeholder = tf.placeholder(tf.float32, shape=(batch\_size,  
                                                       mnist.IMAGE\_PIXELS))  
labels\_placeholder = tf.placeholder(tf.int32, shape=(batch\_size))

Further down, in the training loop, the full image and label datasets are sliced to fit the batch\_size for each step, matched with these placeholder ops, and then passed into the sess.run() function using the feed\_dictparameter.

## Build the Graph

After creating placeholders for the data, the graph is built from the mnist.py file according to a 3-stage pattern: inference(), loss(), and training().

1. inference() - Builds the graph as far as is required for running the network forward to make predictions.
2. loss() - Adds to the inference graph the ops required to generate loss.
3. training() - Adds to the loss graph the ops required to compute and apply gradients.

### Inference

The inference() function builds the graph as far as needed to return the tensor that would contain the output predictions.

It takes the images placeholder as input and builds on top of it a pair of fully connected layers with [ReLU](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))activation followed by a ten  node linear layer specifying the output logits.

Each layer is created beneath a unique [tf.name\_scope](https://www.tensorflow.org/api_docs/python/tf/name_scope) that acts as a prefix to the items created within that scope.

with tf.name\_scope('hidden1'):

Within the defined scope, the weights and biases to be used by each of these layers are generated into [tf.Variable](https://www.tensorflow.org/api_docs/python/tf/Variable) instances, with their desired shapes:

weights = tf.Variable(  
    tf.truncated\_normal([IMAGE\_PIXELS, hidden1\_units],  
                        stddev=1.0 / math.sqrt(float(IMAGE\_PIXELS))),  
    name='weights')  
biases = tf.Variable(tf.zeros([hidden1\_units]),  
                     name='biases')

When, for instance, these are created under the hidden1 scope, the unique name given to the weights variable would be "hidden1/weights".

Each variable is given initializer ops as part of their construction.

In this most common case, the weights are initialized with the [tf.truncated\_normal](https://www.tensorflow.org/api_docs/python/tf/truncated_normal) and given their shape of a 2-D tensor with the first dim representing the number of units in the layer from which the weights connect and the second dim representing the number of units in the layer to which the weights connect. For the first layer, named hidden1, the dimensions are [IMAGE\_PIXELS, hidden1\_units] because the weights are connecting the image inputs to the hidden1 layer. The tf.truncated\_normal initializer generates a random distribution with a given mean and standard deviation.

Then the biases are initialized with [tf.zeros](https://www.tensorflow.org/api_docs/python/tf/zeros) to ensure they start with all zero values, and their shape is simply the number of units in the layer to which they connect.

The graph's three primary ops -- two [tf.nn.relu](https://www.tensorflow.org/api_docs/python/tf/nn/relu) ops wrapping [tf.matmul](https://www.tensorflow.org/api_docs/python/tf/matmul) for the hidden layers and one extra tf.matmul for the logits -- are then created, each in turn, with separate tf.Variable instances connected to each of the input placeholders or the output tensors of the previous layer.

hidden1 = tf.nn.relu(tf.matmul(images, weights) + biases)

hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)

logits = tf.matmul(hidden2, weights) + biases

Finally, the logits tensor that will contain the output is returned.

### Loss

The loss() function further builds the graph by adding the required loss ops.

First, the values from the labels\_placeholder are converted to 64-bit integers. Then, a [tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits](https://www.tensorflow.org/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits) op is added to automatically produce 1-hot labels from the labels\_placeholder and compare the output logits from the inference() function with those 1-hot labels.

labels = tf.to\_int64(labels)  
cross\_entropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(  
    labels=labels, logits=logits, name='xentropy')

It then uses [tf.reduce\_mean](https://www.tensorflow.org/api_docs/python/tf/reduce_mean) to average the cross entropy values across the batch dimension (the first dimension) as the total loss.

loss = tf.reduce\_mean(cross\_entropy, name='xentropy\_mean')

And the tensor that will then contain the loss value is returned.

**Note:** Cross-entropy is an idea from information theory that allows us to describe how bad it is to believe the predictions of the neural network, given what is actually true. For more information, read the blog post Visual Information Theory (http://colah.github.io/posts/2015-09-Visual-Information/)

### Training

The training() function adds the operations needed to minimize the loss via [Gradient Descent](https://en.wikipedia.org/wiki/Gradient_descent).

Firstly, it takes the loss tensor from the loss() function and hands it to a [tf.summary.scalar](https://www.tensorflow.org/api_docs/python/tf/summary/scalar), an op for generating summary values into the events file when used with a [tf.summary.FileWriter](https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter) (see below). In this case, it will emit the snapshot value of the loss every time the summaries are written out.

tf.summary.scalar('loss', loss)

Next, we instantiate a [tf.train.GradientDescentOptimizer](https://www.tensorflow.org/api_docs/python/tf/train/GradientDescentOptimizer) responsible for applying gradients with the requested learning rate.

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

We then generate a single variable to contain a counter for the global training step and the [tf.train.Optimizer.minimize](https://www.tensorflow.org/api_docs/python/tf/train/Optimizer#minimize) op is used to both update the trainable weights in the system and increment the global step. This op is, by convention, known as the train\_op and is what must be run by a TensorFlow session in order to induce one full step of training (see below).

global\_step = tf.Variable(0, name='global\_step', trainable=False)  
train\_op = optimizer.minimize(loss, global\_step=global\_step)

## Train the Model

Once the graph is built, it can be iteratively trained and evaluated in a loop controlled by the user code in fully\_connected\_feed.py.

### The Graph

At the top of the run\_training() function is a python with command that indicates all of the built ops are to be associated with the default global [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph) instance.

with tf.Graph().as\_default():

A tf.Graph is a collection of ops that may be executed together as a group. Most TensorFlow uses will only need to rely on the single default graph.

More complicated uses with multiple graphs are possible, but beyond the scope of this simple tutorial.

### The Session

Once all of the build preparation has been completed and all of the necessary ops generated, a [tf.Session](https://www.tensorflow.org/api_docs/python/tf/Session) is created for running the graph.

sess = tf.Session()

Alternately, a Session may be generated into a with block for scoping:

with tf.Session() as sess:

The empty parameter to session indicates that this code will attach to (or create if not yet created) the default local session.

Immediately after creating the session, all of the tf.Variable instances are initialized by calling [tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session#run) on their initialization op.

init = tf.global\_variables\_initializer()  
sess.run(init)

The [tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session#run) method will run the complete subset of the graph that corresponds to the op(s) passed as parameters. In this first call, the init op is a [tf.group](https://www.tensorflow.org/api_docs/python/tf/group) that contains only the initializers for the variables. None of the rest of the graph is run here; that happens in the training loop below.

### Train Loop

After initializing the variables with the session, training may begin.

The user code controls the training per step, and the simplest loop that can do useful training is:

for step in xrange(FLAGS.max\_steps):  
    sess.run(train\_op)

However, this tutorial is slightly more complicated in that it must also slice up the input data for each step to match the previously generated placeholders.

#### Feed the Graph

For each step, the code will generate a feed dictionary that will contain the set of examples on which to train for the step, keyed by the placeholder ops they represent.

In the fill\_feed\_dict() function, the given DataSet is queried for its next batch\_size set of images and labels, and tensors matching the placeholders are filled containing the next images and labels.

images\_feed, labels\_feed = data\_set.next\_batch(FLAGS.batch\_size,  
                                               FLAGS.fake\_data)

A python dictionary object is then generated with the placeholders as keys and the representative feed tensors as values.

feed\_dict = {  
    images\_placeholder: images\_feed,  
    labels\_placeholder: labels\_feed,  
}

This is passed into the sess.run() function's feed\_dict parameter to provide the input examples for this step of training.

#### Check the Status

The code specifies two values to fetch in its run call: [train\_op, loss].

for step in xrange(FLAGS.max\_steps):  
    feed\_dict = fill\_feed\_dict(data\_sets.train,  
                               images\_placeholder,  
                               labels\_placeholder)  
    \_, loss\_value = sess.run([train\_op, loss],  
                             feed\_dict=feed\_dict)

Because there are two values to fetch, sess.run() returns a tuple with two items. Each Tensor in the list of values to fetch corresponds to a numpy array in the returned tuple, filled with the value of that tensor during this step of training. Since train\_op is an Operation with no output value, the corresponding element in the returned tuple is None and, thus, discarded. However, the value of the loss tensor may become NaN if the model diverges during training, so we capture this value for logging.

Assuming that the training runs fine without NaNs, the training loop also prints a simple status text every 100 steps to let the user know the state of training.

if step % 100 == 0:  
    print('Step %d: loss = %.2f (%.3f sec)' % (step, loss\_value, duration))

#### Visualize the Status

In order to emit the events files used by [TensorBoard](https://www.tensorflow.org/get_started/summaries_and_tensorboard), all of the summaries (in this case, only one) are collected into a single Tensor during the graph building phase.

summary = tf.summary.merge\_all()

And then after the session is created, a [tf.summary.FileWriter](https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter) may be instantiated to write the events files, which contain both the graph itself and the values of the summaries.

summary\_writer = tf.summary.FileWriter(FLAGS.train\_dir, sess.graph)

Lastly, the events file will be updated with new summary values every time the summary is evaluated and the output passed to the writer's add\_summary() function.

summary\_str = sess.run(summary, feed\_dict=feed\_dict)  
summary\_writer.add\_summary(summary\_str, step)

When the events files are written, TensorBoard may be run against the training folder to display the values from the summaries.

**NOTE**: For more info about how to build and run Tensorboard, please see the accompanying tutorial [Tensorboard: Visualizing Learning](https://www.tensorflow.org/get_started/summaries_and_tensorboard).

#### Save a Checkpoint

In order to emit a checkpoint file that may be used to later restore a model for further training or evaluation, we instantiate a [tf.train.Saver](https://www.tensorflow.org/api_docs/python/tf/train/Saver).

saver = tf.train.Saver()

In the training loop, the [tf.train.Saver.save](https://www.tensorflow.org/api_docs/python/tf/train/Saver#save) method will periodically be called to write a checkpoint file to the training directory with the current values of all the trainable variables.

saver.save(sess, FLAGS.train\_dir, global\_step=step)

At some later point in the future, training might be resumed by using the [tf.train.Saver.restore](https://www.tensorflow.org/api_docs/python/tf/train/Saver#restore) method to reload the model parameters.

saver.restore(sess, FLAGS.train\_dir)

## Evaluate the Model

Every thousand steps, the code will attempt to evaluate the model against both the training and test datasets. The do\_eval() function is called thrice, for the training, validation, and test datasets.

print('Training Data Eval:')  
do\_eval(sess,  
        eval\_correct,  
        images\_placeholder,  
        labels\_placeholder,  
        data\_sets.train)  
print('Validation Data Eval:')  
do\_eval(sess,  
        eval\_correct,  
        images\_placeholder,  
        labels\_placeholder,  
        data\_sets.validation)  
print('Test Data Eval:')  
do\_eval(sess,  
        eval\_correct,  
        images\_placeholder,  
        labels\_placeholder,  
        data\_sets.test)

Note that more complicated usage would usually sequester the data\_sets.test to only be checked after significant amounts of hyperparameter tuning. For the sake of a simple little MNIST problem, however, we evaluate against all of the data.

### Build the Eval Graph

Before entering the training loop, the Eval op should have been built by calling the evaluation() function from mnist.py with the same logits/labels parameters as the loss() function.

eval\_correct = mnist.evaluation(logits, labels\_placeholder)

The evaluation() function simply generates a [tf.nn.in\_top\_k](https://www.tensorflow.org/api_docs/python/tf/nn/in_top_k) op that can automatically score each model output as correct if the true label can be found in the K most-likely predictions. In this case, we set the value of K to 1 to only consider a prediction correct if it is for the true label.

eval\_correct = tf.nn.in\_top\_k(logits, labels, 1)

### Eval Output

One can then create a loop for filling a feed\_dict and calling sess.run() against the eval\_correct op to evaluate the model on the given dataset.

for step in xrange(steps\_per\_epoch):  
    feed\_dict = fill\_feed\_dict(data\_set,  
                               images\_placeholder,  
                               labels\_placeholder)  
    true\_count += sess.run(eval\_correct, feed\_dict=feed\_dict)

The true\_count variable simply accumulates all of the predictions that the in\_top\_k op has determined to be correct. From there, the precision may be calculated from simply dividing by the total number of examples.

precision = true\_count / num\_examples  
print('  Num examples: %d  Num correct: %d  Precision @ 1: %0.04f' %  
      (num\_examples, true\_count, precision))

# tf.contrib.learn Quickstart

TensorFlow’s high-level machine learning API (tf.contrib.learn) makes it easy to configure, train, and evaluate a variety of machine learning models. In this tutorial, you’ll use tf.contrib.learn to construct a[neural network](https://en.wikipedia.org/wiki/Artificial_neural_network) classifier and train it on the [Iris data set](https://en.wikipedia.org/wiki/Iris_flower_data_set) to predict flower species based on sepal/petal geometry. You'll write code to perform the following five steps:

1. Load CSVs containing Iris training/test data into a TensorFlow Dataset
2. Construct a [neural network classifier](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/DNNClassifier)
3. Fit the model using the training data
4. Evaluate the accuracy of the model
5. Classify new samples

NOTE: Remember to [install TensorFlow on your machine](https://www.tensorflow.org/install/index) before getting started with this tutorial.

## Complete Neural Network Source Code

Here is the full code for the neural network classifier:

from \_\_future\_\_ import absolute\_import  
from \_\_future\_\_ import division  
from \_\_future\_\_ import print\_function  
  
import os  
import urllib  
  
import numpy as np  
import tensorflow as tf  
  
# Data sets  
IRIS\_TRAINING = "iris\_training.csv"  
IRIS\_TRAINING\_URL = "http://download.tensorflow.org/data/iris\_training.csv"  
  
IRIS\_TEST = "iris\_test.csv"  
IRIS\_TEST\_URL = "http://download.tensorflow.org/data/iris\_test.csv"  
  
def main():  
  # If the training and test sets aren't stored locally, download them.  
  if not os.path.exists(IRIS\_TRAINING):  
    raw = urllib.urlopen(IRIS\_TRAINING\_URL).read()  
    with open(IRIS\_TRAINING, "w") as f:  
      f.write(raw)  
  
  if not os.path.exists(IRIS\_TEST):  
    raw = urllib.urlopen(IRIS\_TEST\_URL).read()  
    with open(IRIS\_TEST, "w") as f:  
      f.write(raw)  
  
  # Load datasets.  
  training\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(  
      filename=IRIS\_TRAINING,  
      target\_dtype=np.int,  
      features\_dtype=np.float32)  
  test\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(  
      filename=IRIS\_TEST,  
      target\_dtype=np.int,  
      features\_dtype=np.float32)  
  
  # Specify that all features have real-value data  
  feature\_columns = [tf.contrib.layers.real\_valued\_column("", dimension=4)]  
  
  # Build 3 layer DNN with 10, 20, 10 units respectively.  
  classifier = tf.contrib.learn.DNNClassifier(feature\_columns=feature\_columns,  
                                              hidden\_units=[10, 20, 10],  
                                              n\_classes=3,  
                                              model\_dir="/tmp/iris\_model")  
  # Define the training inputs  
  def get\_train\_inputs():  
    x = tf.constant(training\_set.data)  
    y = tf.constant(training\_set.target)  
  
    return x, y  
  
  # Fit model.  
  classifier.fit(input\_fn=get\_train\_inputs, steps=2000)  
  
  # Define the test inputs  
  def get\_test\_inputs():  
    x = tf.constant(test\_set.data)  
    y = tf.constant(test\_set.target)  
  
    return x, y  
  
  # Evaluate accuracy.  
  accuracy\_score = classifier.evaluate(input\_fn=get\_test\_inputs,  
                                       steps=1)["accuracy"]  
  
  print("\nTest Accuracy: {0:f}\n".format(accuracy\_score))  
  
  # Classify two new flower samples.  
  def new\_samples():  
    return np.array(  
      [[6.4, 3.2, 4.5, 1.5],  
       [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)  
  
  predictions = list(classifier.predict(input\_fn=new\_samples))  
  
  print(  
      "New Samples, Class Predictions:    {}\n"  
      .format(predictions))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
    main()

The following sections walk through the code in detail.

## Load the Iris CSV data to TensorFlow

The [Iris data set](https://en.wikipedia.org/wiki/Iris_flower_data_set) contains 150 rows of data, comprising 50 samples from each of three related Iris species: Iris setosa, Iris virginica, and Iris versicolor.

**From left to right,**[**Iris setosa**](https://commons.wikimedia.org/w/index.php?curid=170298)**(by**[**Radomil**](https://commons.wikimedia.org/wiki/User:Radomil)**, CC BY-SA 3.0),**[**Iris versicolor**](https://commons.wikimedia.org/w/index.php?curid=248095)**(by**[**Dlanglois**](https://commons.wikimedia.org/wiki/User:Dlanglois)**, CC BY-SA 3.0), and**[**Iris virginica**](https://www.flickr.com/photos/33397993@N05/3352169862)**(by**[**Frank Mayfield**](https://www.flickr.com/photos/33397993@N05)**, CC BY-SA 2.0).**

Each row contains the following data for each flower sample: [sepal](https://en.wikipedia.org/wiki/Sepal) length, sepal width, [petal](https://en.wikipedia.org/wiki/Petal) length, petal width, and flower species. Flower species are represented as integers, with 0 denoting Iris setosa, 1 denoting Iris versicolor, and 2 denoting Iris virginica.

| Sepal Length | Sepal Width | Petal Length | Petal Width | Species |
| --- | --- | --- | --- | --- |
| 5.1 | 3.5 | 1.4 | 0.2 | 0 |
| 4.9 | 3.0 | 1.4 | 0.2 | 0 |
| 4.7 | 3.2 | 1.3 | 0.2 | 0 |
| … | … | … | … | … |
| 7.0 | 3.2 | 4.7 | 1.4 | 1 |
| 6.4 | 3.2 | 4.5 | 1.5 | 1 |
| 6.9 | 3.1 | 4.9 | 1.5 | 1 |
| … | … | … | … | … |
| 6.5 | 3.0 | 5.2 | 2.0 | 2 |
| 6.2 | 3.4 | 5.4 | 2.3 | 2 |
| 5.9 | 3.0 | 5.1 | 1.8 | 2 |

For this tutorial, the Iris data has been randomized and split into two separate CSVs:

* A training set of 120 samples ([iris\_training.csv](http://download.tensorflow.org/data/iris_training.csv))
* A test set of 30 samples ([iris\_test.csv](http://download.tensorflow.org/data/iris_test.csv)).

To get started, first import all the necessary modules, and define where to download and store the dataset:

from \_\_future\_\_ import absolute\_import  
from \_\_future\_\_ import division  
from \_\_future\_\_ import print\_function  
  
import os  
import urllib  
  
import tensorflow as tf  
import numpy as np  
  
IRIS\_TRAINING = "iris\_training.csv"  
IRIS\_TRAINING\_URL = "http://download.tensorflow.org/data/iris\_training.csv"  
  
IRIS\_TEST = "iris\_test.csv"  
IRIS\_TEST\_URL = "http://download.tensorflow.org/data/iris\_test.csv"

Then, if the training and test sets aren't already stored locally, download them.

if not os.path.exists(IRIS\_TRAINING):  
  raw = urllib.urlopen(IRIS\_TRAINING\_URL).read()  
  with open(IRIS\_TRAINING,'w') as f:  
    f.write(raw)  
  
if not os.path.exists(IRIS\_TEST):  
  raw = urllib.urlopen(IRIS\_TEST\_URL).read()  
  with open(IRIS\_TEST,'w') as f:  
    f.write(raw)

Next, load the training and test sets into Datasets using the [load\_csv\_with\_header()](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/contrib/learn/python/learn/datasets/base.py) method in learn.datasets.base. The load\_csv\_with\_header() method takes three required arguments:

* filename, which takes the filepath to the CSV file
* target\_dtype, which takes the [numpy datatype](http://docs.scipy.org/doc/numpy/user/basics.types.html) of the dataset's target value.
* features\_dtype, which takes the [numpy datatype](http://docs.scipy.org/doc/numpy/user/basics.types.html) of the dataset's feature values.

Here, the target (the value you're training the model to predict) is flower species, which is an integer from 0–2, so the appropriate numpy datatype is np.int:

# Load datasets.  
training\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(  
    filename=IRIS\_TRAINING,  
    target\_dtype=np.int,  
    features\_dtype=np.float32)  
test\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(  
    filename=IRIS\_TEST,  
    target\_dtype=np.int,  
    features\_dtype=np.float32)

Datasets in tf.contrib.learn are [named tuples](https://docs.python.org/2/library/collections.html#collections.namedtuple); you can access feature data and target values via the data and target fields. Here, training\_set.data and training\_set.target contain the feature data and target values for the training set, respectively, and test\_set.data and test\_set.targetcontain feature data and target values for the test set.

Later on, in ["Fit the DNNClassifier to the Iris Training Data,"](https://www.tensorflow.org/get_started/tflearn#fit_dnnclassifier) you'll use training\_set.data andtraining\_set.target to train your model, and in ["Evaluate Model Accuracy,"](https://www.tensorflow.org/get_started/tflearn#evaluate_accuracy) you'll use test\_set.data and test\_set.target. But first, you'll construct your model in the next section.

## Construct a Deep Neural Network Classifier

tf.contrib.learn offers a variety of predefined models, called [Estimators](https://www.tensorflow.org/api_guides/python/contrib.learn#estimators), which you can use "out of the box" to run training and evaluation operations on your data. Here, you'll configure a Deep Neural Network Classifier model to fit the Iris data. Using tf.contrib.learn, you can instantiate your[tf.contrib.learn.DNNClassifier](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/DNNClassifier) with just a couple lines of code:

# Specify that all features have real-value data  
feature\_columns = [tf.contrib.layers.real\_valued\_column("", dimension=4)]  
  
# Build 3 layer DNN with 10, 20, 10 units respectively.  
classifier = tf.contrib.learn.DNNClassifier(feature\_columns=feature\_columns,  
                                            hidden\_units=[10, 20, 10],  
                                            n\_classes=3,  
                                            model\_dir="/tmp/iris\_model")

The code above first defines the model's feature columns, which specify the data type for the features in the data set. All the feature data is continuous, so tf.contrib.layers.real\_valued\_column is the appropriate function to use to construct the feature columns. There are four features in the data set (sepal width, sepal height, petal width, and petal height), so accordingly dimension must be set to 4to hold all the data.

Then, the code creates a DNNClassifier model using the following arguments:

* feature\_columns=feature\_columns. The set of feature columns defined above.
* hidden\_units=[10, 20, 10]. Three [hidden layers](http://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw), containing 10, 20, and 10 neurons, respectively.
* n\_classes=3. Three target classes, representing the three Iris species.
* model\_dir=/tmp/iris\_model. The directory in which TensorFlow will save checkpoint data during model training. For more on logging and monitoring with TensorFlow, see [Logging and Monitoring Basics with tf.contrib.learn](https://www.tensorflow.org/get_started/monitors).

## Describe the training input pipeline

The tf.contrib.learn API uses input functions, which create the TensorFlow operations that generate data for the model. In this case, the data is small enough that it can be stored in [tf.constant TensorFlow constants](https://www.tensorflow.org/get_started/BROKEN_LINK). The following code produces the simplest possible input pipeline:

# Define the test inputs  
def get\_train\_inputs():  
  x = tf.constant(training\_set.data)  
  y = tf.constant(training\_set.target)  
  
  return x, y

## Fit the DNNClassifier to the Iris Training Data

Now that you've configured your DNN classifier model, you can fit it to the Iris training data using the [fit](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/BaseEstimator#fit) method. Pass get\_train\_inputs as the input\_fn, and the number of steps to train (here, 2000):

# Fit model.  
classifier.fit(input\_fn=get\_train\_inputs, steps=2000)

The state of the model is preserved in the classifier, which means you can train iteratively if you like. For example, the above is equivalent to the following:

classifier.fit(x=training\_set.data, y=training\_set.target, steps=1000)  
classifier.fit(x=training\_set.data, y=training\_set.target, steps=1000)

However, if you're looking to track the model while it trains, you'll likely want to instead use a TensorFlow [monitor](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/monitors) to perform logging operations. See the tutorial [“Logging and Monitoring Basics with tf.contrib.learn”](https://www.tensorflow.org/get_started/monitors) for more on this topic.

## Evaluate Model Accuracy

You've fit your DNNClassifier model on the Iris training data; now, you can check its accuracy on the Iris test data using the [evaluate](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/BaseEstimator#evaluate) method. Like fit, evaluate takes an input function that builds its input pipeline. evaluate returns a dict with the evaluation results. The following code passes the Iris test data—test\_set.data and test\_set.target—to evaluate and prints the accuracy from the results:

# Define the test inputs  
def get\_test\_inputs():  
  x = tf.constant(test\_set.data)  
  y = tf.constant(test\_set.target)  
  
  return x, y  
  
# Evaluate accuracy.  
accuracy\_score = classifier.evaluate(input\_fn=get\_test\_inputs,  
                                     steps=1)["accuracy"]  
  
print("\nTest Accuracy: {0:f}\n".format(accuracy\_score))

**Note:** The **steps** argument to **evaluate** is important here. [**evaluate**](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/Evaluable#evaluate) normally runs until it reaches the end of the input. This is perfect for evaluating over a set of files, but the constants being used here will never throw the **OutOfRangeError** or **StopIteration** that it is expecting.

When you run the full script, it will print something close to:

Test Accuracy: 0.966667

Your accuracy result may vary a bit, but should be higher than 90%. Not bad for a relatively small data set!

## Classify New Samples

Use the estimator's predict() method to classify new samples. For example, say you have these two new flower samples:

| Sepal Length | Sepal Width | Petal Length | Petal Width |
| --- | --- | --- | --- |
| 6.4 | 3.2 | 4.5 | 1.5 |
| 5.8 | 3.1 | 5.0 | 1.7 |

You can predict their species using the predict() method. predict returns a generator, which can easily be converted to a list. The following code retrieves and prints the class predictions:

# Classify two new flower samples.  
def new\_samples():  
  return np.array(  
    [[6.4, 3.2, 4.5, 1.5],  
     [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)  
  
predictions = list(classifier.predict(input\_fn=new\_samples))  
  
print(  
    "New Samples, Class Predictions:    {}\n"  
    .format(predictions))

Your results should look as follows:

New Samples, Class Predictions:    [1 2]

The model thus predicts that the first sample is Iris versicolor, and the second sample is Iris virginica.

## Additional Resources

* For further reference materials on tf.contrib.learn, see the official [API docs](https://www.tensorflow.org/api_guides/python/contrib.learn).
* To learn more about using tf.contrib.learn to create linear models, see [Large-scale Linear Models with TensorFlow](https://www.tensorflow.org/tutorials/linear).
* To build your own Estimator using tf.contrib.learn APIs, check out [Creating Estimators in tf.contrib.learn](https://www.tensorflow.org/extend/estimators).
* To experiment with neural network modeling and visualization in the browser, check out [Deep Playground](http://playground.tensorflow.org/).
* For more advanced tutorials on neural networks, see [Convolutional Neural Networks](https://www.tensorflow.org/tutorials/deep_cnn) and [Recurrent Neural Networks](https://www.tensorflow.org/tutorials/recurrent).

# Building Input Functions with tf.contrib.learn

This tutorial introduces you to creating input functions in tf.contrib.learn. You'll get an overview of how to construct an input\_fn to preprocess and feed data into your models. Then, you'll implement an input\_fn that feeds training, evaluation, and prediction data into a neural network regressor for predicting median house values.

## Custom Input Pipelines with input\_fn

When training a neural network using tf.contrib.learn, it's possible to pass your feature and target data directly into your fit, evaluate, or predict operations. Here's an example taken from the [tf.contrib.learn quickstart tutorial](https://www.tensorflow.org/get_started/tflearn):

training\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(  
    filename=IRIS\_TRAINING, target\_dtype=np.int, features\_dtype=np.float32)  
test\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(  
    filename=IRIS\_TEST, target\_dtype=np.int, features\_dtype=np.float32)  
...  
  
classifier.fit(x=training\_set.data,  
               y=training\_set.target,  
               steps=2000)

This approach works well when little to no manipulation of source data is required. But in cases where more feature engineering is needed, tf.contrib.learn supports using a custom input function (input\_fn) to encapsulate the logic for preprocessing and piping data into your models.

### Anatomy of an input\_fn

The following code illustrates the basic skeleton for an input function:

def my\_input\_fn():  
  
    # Preprocess your data here...  
  
    # ...then return 1) a mapping of feature columns to Tensors with  
    # the corresponding feature data, and 2) a Tensor containing labels  
    return feature\_cols, labels

The body of the input function contains the specific logic for preprocessing your input data, such as scrubbing out bad examples or [feature scaling](https://en.wikipedia.org/wiki/Feature_scaling).

Input functions must return the following two values containing the final feature and label data to be fed into your model (as shown in the above code skeleton):

feature\_cols

A dict containing key/value pairs that map feature column names to Tensors (or SparseTensors) containing the corresponding feature data.

labels

A Tensor containing your label (target) values: the values your model aims to predict.

### Converting Feature Data to Tensors

If your feature/label data is stored in [pandas](http://pandas.pydata.org/) dataframes or [numpy](http://www.numpy.org/) arrays, you'll need to convert it to Tensors before returning it from your input\_fn.

For continuous data, you can create and populate a Tensor using tf.constant:

feature\_column\_data = [1, 2.4, 0, 9.9, 3, 120]  
feature\_tensor = tf.constant(feature\_column\_data)

For [sparse, categorical data](https://en.wikipedia.org/wiki/Sparse_matrix) (data where the majority of values are 0), you'll instead want to populate aSparseTensor, which is instantiated with three arguments:

dense\_shape

The shape of the tensor. Takes a list indicating the number of elements in each dimension. For example, dense\_shape=[3,6] specifies a two-dimensional 3x6 tensor, dense\_shape=[2,3,4]specifies a three-dimensional 2x3x4 tensor, and dense\_shape=[9] specifies a one-dimensional tensor with 9 elements.

indices

The indices of the elements in your tensor that contain nonzero values. Takes a list of terms, where each term is itself a list containing the index of a nonzero element. (Elements are zero-indexed—i.e., [0,0] is the index value for the element in the first column of the first row in a two-dimensional tensor.) For example, indices=[[1,3], [2,4]] specifies that the elements with indexes of [1,3] and [2,4] have nonzero values.

values

A one-dimensional tensor of values. Term i in values corresponds to term i in indices and specifies its value. For example, given indices=[[1,3], [2,4]], the parameter values=[18, 3.6] specifies that element [1,3] of the tensor has a value of 18, and element [2,4] of the tensor has a value of 3.6.

The following code defines a two-dimensional SparseTensor with 3 rows and 5 columns. The element with index [0,1] has a value of 6, and the element with index [2,4] has a value of 0.5 (all other values are 0):

sparse\_tensor = tf.SparseTensor(indices=[[0,1], [2,4]],  
                                values=[6, 0.5],  
                                dense\_shape=[3, 5])

This corresponds to the following dense tensor:

[[0, 6, 0, 0, 0]  
 [0, 0, 0, 0, 0]  
 [0, 0, 0, 0, 0.5]]

For more on SparseTensor, see the [tf.SparseTensor](https://www.tensorflow.org/api_docs/python/tf/SparseTensor).

### Passing input\_fn Data to Your Model

To feed data to your model for training, you simply pass the input function you've created to your fitoperation as the value of the input\_fn parameter, e.g.:

classifier.fit(input\_fn=my\_input\_fn, steps=2000)

Note that the input\_fn is responsible for supplying both feature and label data to the model, and replaces both the x and y parameters in fit. If you supply an input\_fn value to fit that is not None in conjunction with either an x or y parameter that is not None, it will result in a ValueError.

Also note that the input\_fn parameter must receive a function object (i.e., input\_fn=my\_input\_fn), not the return value of a function call (input\_fn=my\_input\_fn()). This means that if you try to pass parameters to the input function in your fit call, as in the following code, it will result in a TypeError:

classifier.fit(input\_fn=my\_input\_fn(training\_set), steps=2000)

However, if you'd like to be able to parameterize your input function, there are other methods for doing so. You can employ a wrapper function that takes no arguments as your input\_fn and use it to invoke your input function with the desired parameters. For example:

def my\_input\_function\_training\_set():  
  return my\_input\_function(training\_set)  
  
classifier.fit(input\_fn=my\_input\_fn\_training\_set, steps=2000)

Alternatively, you can use Python's [functools.partial](https://docs.python.org/2/library/functools.html#functools.partial) function to construct a new function object with all parameter values fixed:

classifier.fit(input\_fn=functools.partial(my\_input\_function,  
                                          data\_set=training\_set), steps=2000)

A third option is to wrap your input\_fn invocation in a [lambda](https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions) and pass it to the input\_fn parameter:

classifier.fit(input\_fn=lambda: my\_input\_fn(training\_set), steps=2000)

One big advantage of architecting your input pipeline as shown above—to accept a parameter for data set—is that you can pass the same input\_fn to evaluate and predict operations by just changing the data set argument, e.g.:

classifier.evaluate(input\_fn=lambda: my\_input\_fn(test\_set), steps=2000)

This approach enhances code maintainability: no need to capture x and y values in separate variables (e.g., x\_train, x\_test, y\_train, y\_test) for each type of operation.

### A Neural Network Model for Boston House Values

In the remainder of this tutorial, you'll write an input function for preprocessing a subset of Boston housing data pulled from the [UCI Housing Data Set](https://archive.ics.uci.edu/ml/datasets/Housing) and use it to feed data to a neural network regressor for predicting median house values.

The [Boston CSV data sets](https://www.tensorflow.org/get_started/input_fn#setup) you'll use to train your neural network contain the following [feature data](https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names) for Boston suburbs:

| Feature | Description |
| --- | --- |
| CRIM | Crime rate per capita |
| ZN | Fraction of residential land zoned to permit 25,000+ sq ft lots |
| INDUS | Fraction of land that is non-retail business |
| NOX | Concentration of nitric oxides in parts per 10 million |
| RM | Average Rooms per dwelling |
| AGE | Fraction of owner-occupied residences built before 1940 |
| DIS | Distance to Boston-area employment centers |
| TAX | Property tax rate per $10,000 |
| PTRATIO | Student-teacher ratio |

And the label your model will predict is MEDV, the median value of owner-occupied residences in thousands of dollars.

## Setup

Download the following data sets: [boston\_train.csv](http://download.tensorflow.org/data/boston_train.csv), [boston\_test.csv](http://download.tensorflow.org/data/boston_test.csv), and [boston\_predict.csv](http://download.tensorflow.org/data/boston_predict.csv).

The following sections provide a step-by-step walkthrough of how to create an input function, feed these data sets into a neural network regressor, train and evaluate the model, and make house value predictions. The full, final code is [available here](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/input_fn/boston.py).

### Importing the Housing Data

To start, set up your imports (including pandas and tensorflow) and [set logging verbosity](https://www.tensorflow.org/get_started/monitors#enabling_logging_with_tensorflow) to INFOfor more detailed log output:

from \_\_future\_\_ import absolute\_import  
from \_\_future\_\_ import division  
from \_\_future\_\_ import print\_function  
  
import itertools  
  
import pandas as pd  
import tensorflow as tf  
  
tf.logging.set\_verbosity(tf.logging.INFO)

Define the column names for the data set in COLUMNS. To distinguish features from the label, also define FEATURES and LABEL. Then read the three CSVs ([tf.train](https://www.tensorflow.org/api_docs/python/tf/train), [tf.test](https://www.tensorflow.org/api_docs/python/tf/test), and [predict](http://download.tensorflow.org/data/boston_predict.csv)) into pandas DataFrames:

COLUMNS = ["crim", "zn", "indus", "nox", "rm", "age",  
           "dis", "tax", "ptratio", "medv"]  
FEATURES = ["crim", "zn", "indus", "nox", "rm",  
            "age", "dis", "tax", "ptratio"]  
LABEL = "medv"  
  
training\_set = pd.read\_csv("boston\_train.csv", skipinitialspace=True,  
                           skiprows=1, names=COLUMNS)  
test\_set = pd.read\_csv("boston\_test.csv", skipinitialspace=True,  
                       skiprows=1, names=COLUMNS)  
prediction\_set = pd.read\_csv("boston\_predict.csv", skipinitialspace=True,  
                             skiprows=1, names=COLUMNS)

### Defining FeatureColumns and Creating the Regressor

Next, create a list of FeatureColumns for the input data, which formally specify the set of features to use for training. Because all features in the housing data set contain continuous values, you can create their FeatureColumns using the tf.contrib.layers.real\_valued\_column() function:

feature\_cols = [tf.contrib.layers.real\_valued\_column(k)  
                  for k in FEATURES]

NOTE: For a more in-depth overview of feature columns, see [this introduction](https://www.tensorflow.org/tutorials/linear#feature_columns_and_transformations), and for an example that illustrates how to define FeatureColumns for categorical data, see the [Linear Model Tutorial](https://www.tensorflow.org/tutorials/wide).

Now, instantiate a DNNRegressor for the neural network regression model. You'll need to provide two arguments here: hidden\_units, a hyperparameter specifying the number of nodes in each hidden layer (here, two hidden layers with 10 nodes each), and feature\_columns, containing the list ofFeatureColumns you just defined:

regressor = tf.contrib.learn.DNNRegressor(feature\_columns=feature\_cols,  
                                          hidden\_units=[10, 10],  
                                          model\_dir="/tmp/boston\_model")

### Building the input\_fn

To pass input data into the regressor, create an input function, which will accept a pandasDataframe and return feature column and label values as Tensors:

def input\_fn(data\_set):  
  feature\_cols = {k: tf.constant(data\_set[k].values)  
                  for k in FEATURES}  
  labels = tf.constant(data\_set[LABEL].values)  
  return feature\_cols, labels

Note that the input data is passed into input\_fn in the data\_set argument, which means the function can process any of the DataFrames you've imported: training\_set, test\_set, and prediction\_set.

### Training the Regressor

To train the neural network regressor, run fit with the training\_set passed to the input\_fn as follows:

regressor.fit(input\_fn=lambda: input\_fn(training\_set), steps=5000)

You should see log output similar to the following, which reports training loss for every 100 steps:

INFO:tensorflow:Step 1: loss = 483.179  
INFO:tensorflow:Step 101: loss = 81.2072  
INFO:tensorflow:Step 201: loss = 72.4354  
...  
INFO:tensorflow:Step 1801: loss = 33.4454  
INFO:tensorflow:Step 1901: loss = 32.3397  
INFO:tensorflow:Step 2001: loss = 32.0053  
INFO:tensorflow:Step 4801: loss = 27.2791  
INFO:tensorflow:Step 4901: loss = 27.2251  
INFO:tensorflow:Saving checkpoints for 5000 into /tmp/boston\_model/model.ckpt.  
INFO:tensorflow:Loss for final step: 27.1674.

### Evaluating the Model

Next, see how the trained model performs against the test data set. Run evaluate, and this time pass the test\_set to the input\_fn:

ev = regressor.evaluate(input\_fn=lambda: input\_fn(test\_set), steps=1)

Retrieve the loss from the ev results and print it to output:

loss\_score = ev["loss"]  
print("Loss: {0:f}".format(loss\_score))

You should see results similar to the following:

INFO:tensorflow:Eval steps [0,1) for training step 5000.  
INFO:tensorflow:Saving evaluation summary for 5000 step: loss = 11.9221  
Loss: 11.922098

### Making Predictions

Finally, you can use the model to predict median house values for the prediction\_set, which contains feature data but no labels for six examples:

y = regressor.predict(input\_fn=lambda: input\_fn(prediction\_set))  
# .predict() returns an iterator; convert to a list and print predictions  
predictions = list(itertools.islice(y, 6))  
print ("Predictions: {}".format(str(predictions)))

Your results should contain six house-value predictions in thousands of dollars, e.g:

Predictions: [ 33.30348587  17.04452896  22.56370163  34.74345398  14.55953979  
  19.58005714]

## Additional Resources

This tutorial focused on creating an input\_fn for a neural network regressor. To learn more about using input\_fns for other types of models, check out the following resources:

* [Large-scale Linear Models with TensorFlow](https://www.tensorflow.org/tutorials/linear): This introduction to linear models in TensorFlow provides a high-level overview of feature columns and techniques for transforming input data.
* [TensorFlow Linear Model Tutorial](https://www.tensorflow.org/tutorials/wide): This tutorial covers creating FeatureColumns and an input\_fn for a linear classification model that predicts income range based on census data.
* [TensorFlow Wide & Deep Learning Tutorial](https://www.tensorflow.org/tutorials/wide_and_deep): Building on the [Linear Model Tutorial](https://www.tensorflow.org/tutorials/wide), this tutorial covers FeatureColumn and input\_fn creation for a "wide and deep" model that combines a linear model and a neural network using DNNLinearCombinedClassifier.

# Logging and Monitoring Basics with tf.contrib.learn

When training a model, it’s often valuable to track and evaluate progress in real time. In this tutorial, you’ll learn how to use TensorFlow’s logging capabilities and the Monitor API to audit the in-progress training of a neural network classifier for categorizing irises. This tutorial builds on the code developed in [tf.contrib.learn Quickstart](https://www.tensorflow.org/get_started/tflearn) so if you haven't yet completed that tutorial, you may want to explore it first, especially if you're looking for an intro/refresher on tf.contrib.learn basics.

## Setup

For this tutorial, you'll be building upon the following code from [tf.contrib.learn Quickstart](https://www.tensorflow.org/get_started/tflearn):

from \_\_future\_\_ import absolute\_import  
from \_\_future\_\_ import division  
from \_\_future\_\_ import print\_function  
  
import os  
  
import numpy as np  
import tensorflow as tf  
  
# Data sets  
IRIS\_TRAINING = os.path.join(os.path.dirname(\_\_file\_\_), "iris\_training.csv")  
IRIS\_TEST = os.path.join(os.path.dirname(\_\_file\_\_), "iris\_test.csv")  
  
def main(unused\_argv):  
    # Load datasets.  
    training\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(  
        filename=IRIS\_TRAINING, target\_dtype=np.int, features\_dtype=np.float32)  
    test\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(  
        filename=IRIS\_TEST, target\_dtype=np.int, features\_dtype=np.float32)  
  
    # Specify that all features have real-value data  
    feature\_columns = [tf.contrib.layers.real\_valued\_column("", dimension=4)]  
  
    # Build 3 layer DNN with 10, 20, 10 units respectively.  
    classifier = tf.contrib.learn.DNNClassifier(feature\_columns=feature\_columns,  
                                                hidden\_units=[10, 20, 10],  
                                                n\_classes=3,  
                                                model\_dir="/tmp/iris\_model")  
  
    # Fit model.  
    classifier.fit(x=training\_set.data,  
                   y=training\_set.target,  
                   steps=2000)  
  
    # Evaluate accuracy.  
    accuracy\_score = classifier.evaluate(x=test\_set.data,  
                                         y=test\_set.target)["accuracy"]  
    print('Accuracy: {0:f}'.format(accuracy\_score))  
  
    # Classify two new flower samples.  
    new\_samples = np.array(  
        [[6.4, 3.2, 4.5, 1.5], [5.8, 3.1, 5.0, 1.7]], dtype=float)  
    y = list(classifier.predict(new\_samples, as\_iterable=True))  
    print('Predictions: {}'.format(str(y)))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  tf.app.run()

Copy the above code into a file, and download the corresponding [training](http://download.tensorflow.org/data/iris_training.csv) and [tf.test](https://www.tensorflow.org/api_docs/python/tf/test) data sets to the same directory.

In the following sections, you'll progressively make updates to the above code to add logging and monitoring capabilities. Final code incorporating all updates is [available for download here](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/monitors/iris_monitors.py).

## Overview

The [tf.contrib.learn Quickstart tutorial](https://www.tensorflow.org/get_started/tflearn) walked through how to implement a neural net classifier to categorize iris examples into one of three species.

But when [the code](https://www.tensorflow.org/get_started/monitors#setup) from this tutorial is run, the output contains no logging tracking how model training is progressing—only the results of the print statements that were included:

Accuracy: 0.933333  
Predictions: [1 2]

Without any logging, model training feels like a bit of a black box; you can't see what's happening as TensorFlow steps through gradient descent, get a sense of whether the model is converging appropriately, or audit to determine whether [early stopping](https://en.wikipedia.org/wiki/Early_stopping) might be appropriate.

One way to address this problem would be to split model training into multiple fit calls with smaller numbers of steps in order to evaluate accuracy more progressively. However, this is not recommended practice, as it greatly slows down model training. Fortunately, tf.contrib.learn offers another solution: a[Monitor API](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/monitors) designed to help you log metrics and evaluate your model while training is in progress. In the following sections, you'll learn how to enable logging in TensorFlow, set up a ValidationMonitor to do streaming evaluations, and visualize your metrics using TensorBoard.

## Enabling Logging with TensorFlow

TensorFlow uses five different levels for log messages. In order of ascending severity, they are DEBUG, INFO, WARN, ERROR, and FATAL. When you configure logging at any of these levels, TensorFlow will output all log messages corresponding to that level and all levels of higher severity. For example, if you set a logging level of ERROR, you'll get log output containing ERROR and FATAL messages, and if you set a level of DEBUG, you'll get log messages from all five levels.

By default, TensorFlow is configured at a logging level of WARN, but when tracking model training, you'll want to adjust the level to INFO, which will provide additional feedback as fit operations are in progress.

Add the following line to the beginning of your code (right after your imports):

tf.logging.set\_verbosity(tf.logging.INFO)

Now when you run the code, you'll see additional log output like the following:

INFO:tensorflow:loss = 1.18812, step = 1  
INFO:tensorflow:loss = 0.210323, step = 101  
INFO:tensorflow:loss = 0.109025, step = 201

With INFO-level logging, tf.contrib.learn automatically outputs [training-loss metrics](https://en.wikipedia.org/wiki/Loss_function) to stderr after every 100 steps.

## Configuring a ValidationMonitor for Streaming Evaluation

Logging training loss is helpful to get a sense whether your model is converging, but what if you want further insight into what's happening during training? tf.contrib.learn provides several high-level Monitors you can attach to your fit operations to further track metrics and/or debug lower-level TensorFlow operations during model training, including:

| Monitor | Description |
| --- | --- |
| CaptureVariable | Saves a specified variable's values into a collection at every n steps of training |
| PrintTensor | Logs a specified tensor's values at every n steps of training |
| SummarySaver | Saves [tf.Summary](https://www.tensorflow.org/api_docs/python/tf/Summary) [protocol buffers](https://developers.google.com/protocol-buffers/) for a given tensor using a [tf.summary.FileWriter](https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter) at every n steps of training |
| ValidationMonitor | Logs a specified set of evaluation metrics at every n steps of training, and, if desired, implements early stopping under certain conditions |

### Evaluating Every N Steps

For the iris neural network classifier, while logging training loss, you might also want to simultaneously evaluate against test data to see how well the model is generalizing. You can accomplish this by configuring a ValidationMonitor with the test data (test\_set.data and test\_set.target), and setting how often to evaluate with every\_n\_steps. The default value of every\_n\_steps is 100; here, set every\_n\_steps to 50 to evaluate after every 50 steps of model training:

validation\_monitor = tf.contrib.learn.monitors.ValidationMonitor(  
    test\_set.data,  
    test\_set.target,  
    every\_n\_steps=50)

Place this code right before the line instantiating the classifier.

ValidationMonitors rely on saved checkpoints to perform evaluation operations, so you'll want to modify instantiation of the classifier to add a [tf.contrib.learn.RunConfig](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/RunConfig) that includessave\_checkpoints\_secs, which specifies how many seconds should elapse between checkpoint saves during training. Because the iris data set is quite small, and thus trains quickly, it makes sense to set save\_checkpoints\_secs to 1 (saving a checkpoint every second) to ensure a sufficient number of checkpoints:

classifier = tf.contrib.learn.DNNClassifier(  
    feature\_columns=feature\_columns,  
    hidden\_units=[10, 20, 10],  
    n\_classes=3,  
    model\_dir="/tmp/iris\_model",  
    config=tf.contrib.learn.RunConfig(save\_checkpoints\_secs=1))

NOTE: The model\_dir parameter specifies an explicit directory (/tmp/iris\_model) for model data to be stored; this directory path will be easier to reference later on than an autogenerated one. Each time you run the code, any existing data in /tmp/iris\_model will be loaded, and model training will continue where it left off in the last run (e.g., running the script twice in succession will execute 4000 steps during training—2000 during each fit operation). To start over model training from scratch, delete /tmp/iris\_model before running the code.

Finally, to attach your validation\_monitor, update the fit call to include a monitors param, which takes a list of all monitors to run during model training:

classifier.fit(x=training\_set.data,  
               y=training\_set.target,  
               steps=2000,  
               monitors=[validation\_monitor])

Now, when you rerun the code, you should see validation metrics in your log output, e.g.:

INFO:tensorflow:Validation (step 50): loss = 1.71139, global\_step = 0, accuracy = 0.266667  
...  
INFO:tensorflow:Validation (step 300): loss = 0.0714158, global\_step = 268, accuracy = 0.966667  
...  
INFO:tensorflow:Validation (step 1750): loss = 0.0574449, global\_step = 1729, accuracy = 0.966667

### Customizing the Evaluation Metrics with MetricSpec

By default, if no evaluation metrics are specified, ValidationMonitor will log both [loss](https://en.wikipedia.org/wiki/Loss_function) and accuracy, but you can customize the list of metrics that will be run every 50 steps. To specify the exact metrics you'd like to run in each evaluation pass, you can add a metrics param to the ValidationMonitorconstructor. metrics takes a dict of key/value pairs, where each key is the name you'd like logged for the metric, and the corresponding value is a [MetricSpec](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/contrib/learn/python/learn/metric_spec.py) object.

The MetricSpec constructor accepts four parameters:

* metric\_fn. The function that calculates and returns the value of a metric. This can be a predefined function available in the [tf.contrib.metrics](https://www.tensorflow.org/api_docs/python/tf/contrib/metrics) module, such as[tf.contrib.metrics.streaming\_precision](https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_precision) or [tf.contrib.metrics.streaming\_recall](https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_recall).

Alternatively, you can define your own custom metric function, which must take predictionsand labels tensors as arguments (a weights argument can also optionally be supplied). The function must return the value of the metric in one of two formats:

* + A single tensor
  + A pair of ops (value\_op, update\_op), where value\_op returns the metric value and update\_op performs a corresponding operation to update internal model state.
* prediction\_key. The key of the tensor containing the predictions returned by the model. This argument may be omitted if the model returns either a single tensor or a dict with a single entry. For a DNNClassifier model, class predictions will be returned in a tensor with the key[tf.contrib.learn.PredictionKey.CLASSES](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/PredictionKey#CLASSES).
* label\_key. The key of the tensor containing the labels returned by the model, as specified by the model's [input\_fn](https://www.tensorflow.org/get_started/input_fn). As with prediction\_key, this argument may be omitted if the input\_fnreturns either a single tensor or a dict with a single entry. In the iris example in this tutorial, the DNNClassifier does not have an input\_fn (x,y data is passed directly to fit), so it's not necessary to provide a label\_key.
* weights\_key. Optional. The key of the tensor (returned by the [input\_fn](https://www.tensorflow.org/get_started/input_fn)) containing weights inputs for the metric\_fn.

The following code creates a validation\_metrics dict that defines three metrics to log during model evaluation:

* "accuracy", using [tf.contrib.metrics.streaming\_accuracy](https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_accuracy) as the metric\_fn
* "precision", using [tf.contrib.metrics.streaming\_precision](https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_precision) as the metric\_fn
* "recall", using [tf.contrib.metrics.streaming\_recall](https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_recall) as the metric\_fn

validation\_metrics = {  
    "accuracy":  
        tf.contrib.learn.MetricSpec(  
            metric\_fn=tf.contrib.metrics.streaming\_accuracy,  
            prediction\_key=tf.contrib.learn.prediction\_key.PredictionKey.  
            CLASSES),  
    "precision":  
        tf.contrib.learn.MetricSpec(  
            metric\_fn=tf.contrib.metrics.streaming\_precision,  
            prediction\_key=tf.contrib.learn.prediction\_key.PredictionKey.  
            CLASSES),  
    "recall":  
        tf.contrib.learn.MetricSpec(  
            metric\_fn=tf.contrib.metrics.streaming\_recall,  
            prediction\_key=tf.contrib.learn.prediction\_key.PredictionKey.  
            CLASSES)  
}

Add the above code before the ValidationMonitor constructor. Then revise theValidationMonitor constructor as follows to add a metrics parameter to log the accuracy, precision, and recall metrics specified in validation\_metrics (loss is always logged, and doesn't need to be explicitly specified):

validation\_monitor = tf.contrib.learn.monitors.ValidationMonitor(  
    test\_set.data,  
    test\_set.target,  
    every\_n\_steps=50,  
    metrics=validation\_metrics)

Rerun the code, and you should see precision and recall included in your log output, e.g.:

INFO:tensorflow:Validation (step 50): recall = 0.0, loss = 1.20626, global\_step = 1, precision = 0.0, accuracy = 0.266667  
...  
INFO:tensorflow:Validation (step 600): recall = 1.0, loss = 0.0530696, global\_step = 571, precision = 1.0, accuracy = 0.966667  
...  
INFO:tensorflow:Validation (step 1500): recall = 1.0, loss = 0.0617403, global\_step = 1452, precision = 1.0, accuracy = 0.966667

### Early Stopping with ValidationMonitor

Note that in the above log output, by step 600, the model has already achieved precision and recall rates of 1.0. This raises the question as to whether model training could benefit from [early stopping](https://en.wikipedia.org/wiki/Early_stopping).

In addition to logging eval metrics, ValidationMonitors make it easy to implement early stopping when specified conditions are met, via three params:

| Param | Description |
| --- | --- |
| early\_stopping\_metric | Metric that triggers early stopping (e.g., loss or accuracy) under conditions specified in early\_stopping\_rounds andearly\_stopping\_metric\_minimize. Default is "loss". |
| early\_stopping\_metric\_minimize | True if desired model behavior is to minimize the value ofearly\_stopping\_metric; False if desired model behavior is to maximize the value of early\_stopping\_metric. Default is True. |
| early\_stopping\_rounds | Sets a number of steps during which if the early\_stopping\_metricdoes not decrease (if early\_stopping\_metric\_minimize is True) or increase (if early\_stopping\_metric\_minimize is False), training will be stopped. Default is None, which means early stopping will never occur. |

Make the following revision to the ValidationMonitor constructor, which specifies that if loss (early\_stopping\_metric="loss") does not decrease (early\_stopping\_metric\_minimize=True) over a period of 200 steps (early\_stopping\_rounds=200), model training will stop immediately at that point, and not complete the full 2000 steps specified in fit:

validation\_monitor = tf.contrib.learn.monitors.ValidationMonitor(  
    test\_set.data,  
    test\_set.target,  
    every\_n\_steps=50,  
    metrics=validation\_metrics,  
    early\_stopping\_metric="loss",  
    early\_stopping\_metric\_minimize=True,  
    early\_stopping\_rounds=200)

Rerun the code to see if model training stops early:

...  
INFO:tensorflow:Validation (step 1150): recall = 1.0, loss = 0.056436, global\_step = 1119, precision = 1.0, accuracy = 0.966667  
INFO:tensorflow:Stopping. Best step: 800 with loss = 0.048313818872.

Indeed, here training stops at step 1150, indicating that for the past 200 steps, loss did not decrease, and that overall, step 800 produced the smallest loss value against the test data set. This suggests that additional calibration of hyperparameters by decreasing the step count might further improve the model.

## Visualizing Log Data with TensorBoard

Reading through the log produced by ValidationMonitor provides plenty of raw data on model performance during training, but it may also be helpful to see visualizations of this data to get further insight into trends—for example, how accuracy is changing over step count. You can use TensorBoard (a separate program packaged with TensorFlow) to plot graphs like this by setting the logdircommand-line argument to the directory where you saved your model training data (here, /tmp/iris\_model). Run the following on your command line:

**$ tensorboard --logdir=/tmp/iris\_model/**

Starting TensorBoard 39 on port 6006

Then navigate to http://0.0.0.0:<port\_number> in your browser, where <port\_number> is the port specified in the command-line output (here, 6006).

If you click on the accuracy field, you'll see an image like the following, which shows accuracy plotted against step count:

For more on using TensorBoard, see [TensorBoard: Visualizing Learning](https://www.tensorflow.org/get_started/summaries_and_tensorboard) and [TensorBoard: Graph Visualization](https://www.tensorflow.org/get_started/graph_viz).

# TensorBoard: Visualizing Learning

The computations you'll use TensorFlow for - like training a massive deep neural network - can be complex and confusing. To make it easier to understand, debug, and optimize TensorFlow programs, we've included a suite of visualization tools called TensorBoard. You can use TensorBoard to visualize your TensorFlow graph, plot quantitative metrics about the execution of your graph, and show additional data like images that pass through it. When TensorBoard is fully configured, it looks like this:

This tutorial is intended to get you started with simple TensorBoard usage. There are other resources available as well! The [TensorBoard README](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/tensorboard/README.md) has a lot more information on TensorBoard usage, including tips & tricks, and debugging information.

## Serializing the data

TensorBoard operates by reading TensorFlow events files, which contain summary data that you can generate when running TensorFlow. Here's the general lifecycle for summary data within TensorBoard.

First, create the TensorFlow graph that you'd like to collect summary data from, and decide which nodes you would like to annotate with [summary operations](https://www.tensorflow.org/api_guides/python/summary).

For example, suppose you are training a convolutional neural network for recognizing MNIST digits. You'd like to record how the learning rate varies over time, and how the objective function is changing. Collect these by attaching [tf.summary.scalar](https://www.tensorflow.org/api_docs/python/tf/summary/scalar) ops to the nodes that output the learning rate and loss respectively. Then, give each scalar\_summary a meaningful tag, like 'learning rate' or 'loss function'.

Perhaps you'd also like to visualize the distributions of activations coming off a particular layer, or the distribution of gradients or weights. Collect this data by attaching [tf.summary.histogram](https://www.tensorflow.org/api_docs/python/tf/summary/histogram) ops to the gradient outputs and to the variable that holds your weights, respectively.

For details on all of the summary operations available, check out the docs on [summary operations](https://www.tensorflow.org/api_guides/python/summary).

Operations in TensorFlow don't do anything until you run them, or an op that depends on their output. And the summary nodes that we've just created are peripheral to your graph: none of the ops you are currently running depend on them. So, to generate summaries, we need to run all of these summary nodes. Managing them by hand would be tedious, so use [tf.summary.merge\_all](https://www.tensorflow.org/api_docs/python/tf/summary/merge_all) to combine them into a single op that generates all the summary data.

Then, you can just run the merged summary op, which will generate a serialized Summary protobuf object with all of your summary data at a given step. Finally, to write this summary data to disk, pass the summary protobuf to a [tf.summary.FileWriter](https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter).

The FileWriter takes a logdir in its constructor - this logdir is quite important, it's the directory where all of the events will be written out. Also, the FileWriter can optionally take a Graph in its constructor. If it receives a Graph object, then TensorBoard will visualize your graph along with tensor shape information. This will give you a much better sense of what flows through the graph: see [Tensor shape information](https://www.tensorflow.org/get_started/graph_viz#tensor_shape_information).

Now that you've modified your graph and have a FileWriter, you're ready to start running your network! If you want, you could run the merged summary op every single step, and record a ton of training data. That's likely to be more data than you need, though. Instead, consider running the merged summary op every n steps.

The code example below is a modification of the [simple MNIST tutorial](https://www.tensorflow.org/get_started/mnist/beginners), in which we have added some summary ops, and run them every ten steps. If you run this and then launch tensorboard --logdir=/tmp/mnist\_logs, you'll be able to visualize statistics, such as how the weights or accuracy varied during training. The code below is an excerpt; full source is [here](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/mnist/mnist_with_summaries.py).

def variable\_summaries(var):  
  """Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""  
  with tf.name\_scope('summaries'):  
    mean = tf.reduce\_mean(var)  
    tf.summary.scalar('mean', mean)  
    with tf.name\_scope('stddev'):  
      stddev = tf.sqrt(tf.reduce\_mean(tf.square(var - mean)))  
    tf.summary.scalar('stddev', stddev)  
    tf.summary.scalar('max', tf.reduce\_max(var))  
    tf.summary.scalar('min', tf.reduce\_min(var))  
    tf.summary.histogram('histogram', var)  
  
def nn\_layer(input\_tensor, input\_dim, output\_dim, layer\_name, act=tf.nn.relu):  
  """Reusable code for making a simple neural net layer.  
  
  It does a matrix multiply, bias add, and then uses relu to nonlinearize.  
  It also sets up name scoping so that the resultant graph is easy to read,  
  and adds a number of summary ops.  
  """  
  # Adding a name scope ensures logical grouping of the layers in the graph.  
  with tf.name\_scope(layer\_name):  
    # This Variable will hold the state of the weights for the layer  
    with tf.name\_scope('weights'):  
      weights = weight\_variable([input\_dim, output\_dim])  
      variable\_summaries(weights)  
    with tf.name\_scope('biases'):  
      biases = bias\_variable([output\_dim])  
      variable\_summaries(biases)  
    with tf.name\_scope('Wx\_plus\_b'):  
      preactivate = tf.matmul(input\_tensor, weights) + biases  
      tf.summary.histogram('pre\_activations', preactivate)  
    activations = act(preactivate, name='activation')  
    tf.summary.histogram('activations', activations)  
    return activations  
  
hidden1 = nn\_layer(x, 784, 500, 'layer1')  
  
with tf.name\_scope('dropout'):  
  keep\_prob = tf.placeholder(tf.float32)  
  tf.summary.scalar('dropout\_keep\_probability', keep\_prob)  
  dropped = tf.nn.dropout(hidden1, keep\_prob)  
  
# Do not apply softmax activation yet, see below.  
y = nn\_layer(dropped, 500, 10, 'layer2', act=tf.identity)  
  
with tf.name\_scope('cross\_entropy'):  
  # The raw formulation of cross-entropy,  
  #  
  # tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(tf.softmax(y)),  
  #                               reduction\_indices=[1]))  
  #  
  # can be numerically unstable.  
  #  
  # So here we use tf.nn.softmax\_cross\_entropy\_with\_logits on the  
  # raw outputs of the nn\_layer above, and then average across  
  # the batch.  
  diff = tf.nn.softmax\_cross\_entropy\_with\_logits(targets=y\_, logits=y)  
  with tf.name\_scope('total'):  
    cross\_entropy = tf.reduce\_mean(diff)  
tf.summary.scalar('cross\_entropy', cross\_entropy)  
  
with tf.name\_scope('train'):  
  train\_step = tf.train.AdamOptimizer(FLAGS.learning\_rate).minimize(  
      cross\_entropy)  
  
with tf.name\_scope('accuracy'):  
  with tf.name\_scope('correct\_prediction'):  
    correct\_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y\_, 1))  
  with tf.name\_scope('accuracy'):  
    accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  
tf.summary.scalar('accuracy', accuracy)  
  
# Merge all the summaries and write them out to /tmp/mnist\_logs (by default)  
merged = tf.summary.merge\_all()  
train\_writer = tf.summary.FileWriter(FLAGS.summaries\_dir + '/train',  
                                      sess.graph)  
test\_writer = tf.summary.FileWriter(FLAGS.summaries\_dir + '/test')  
tf.global\_variables\_initializer().run()

After we've initialized the FileWriters, we have to add summaries to the FileWriters as we train and test the model.

# Train the model, and also write summaries.  
# Every 10th step, measure test-set accuracy, and write test summaries  
# All other steps, run train\_step on training data, & add training summaries  
  
def feed\_dict(train):  
  """Make a TensorFlow feed\_dict: maps data onto Tensor placeholders."""  
  if train or FLAGS.fake\_data:  
    xs, ys = mnist.train.next\_batch(100, fake\_data=FLAGS.fake\_data)  
    k = FLAGS.dropout  
  else:  
    xs, ys = mnist.test.images, mnist.test.labels  
    k = 1.0  
  return {x: xs, y\_: ys, keep\_prob: k}  
  
for i in range(FLAGS.max\_steps):  
  if i % 10 == 0:  # Record summaries and test-set accuracy  
    summary, acc = sess.run([merged, accuracy], feed\_dict=feed\_dict(False))  
    test\_writer.add\_summary(summary, i)  
    print('Accuracy at step %s: %s' % (i, acc))  
  else:  # Record train set summaries, and train  
    summary, \_ = sess.run([merged, train\_step], feed\_dict=feed\_dict(True))  
    train\_writer.add\_summary(summary, i)

You're now all set to visualize this data using TensorBoard.

## Launching TensorBoard

To run TensorBoard, use the following command (alternatively python -m tensorflow.tensorboard)

tensorboard --logdir=path/to/log-directory

where logdir points to the directory where the FileWriter serialized its data. If this logdirdirectory contains subdirectories which contain serialized data from separate runs, then TensorBoard will visualize the data from all of those runs. Once TensorBoard is running, navigate your web browser to localhost:6006 to view the TensorBoard.

When looking at TensorBoard, you will see the navigation tabs in the top right corner. Each tab represents a set of serialized data that can be visualized.

For in depth information on how to use the graph tab to visualize your graph, see [TensorBoard: Graph Visualization](https://www.tensorflow.org/get_started/graph_viz).

For more usage information on TensorBoard in general, see the [TensorBoard README](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/tensorboard/README.md).

# TensorBoard: Embedding Visualization

Embeddings are ubiquitous in machine learning, appearing in recommender systems, NLP, and many other applications. Indeed, in the context of TensorFlow, it's natural to view tensors (or slices of tensors) as points in space, so almost any TensorFlow system will naturally give rise to various embeddings.

TensorBoard has a built-in visualizer, called the *Embedding Projector*, for interactive visualization and analysis of high-dimensional data like embeddings. The embedding projector will read the embeddings from your model checkpoint file. Although it's most useful for embeddings, it will load any 2D tensor, including your training weights.

To learn more about embeddings and how to train them, see the [Vector Representations of Words](https://www.tensorflow.org/tutorials/word2vec)tutorial. If you are interested in embeddings of images, check out [this article](http://colah.github.io/posts/2014-10-Visualizing-MNIST/) for interesting visualizations of MNIST images. On the other hand, if you are interested in word embeddings, [this article](http://colah.github.io/posts/2015-01-Visualizing-Representations/) gives a good introduction.

By default, the Embedding Projector projects the high-dimensional data into 3 dimensions using[principal component analysis](https://en.wikipedia.org/wiki/Principal_component_analysis). For a visual explanation of PCA, see [this article](http://setosa.io/ev/principal-component-analysis/). Another very useful projection you can use is [t-SNE](https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding). We talk about more t-SNE later in the tutorial.

If you are working with an embedding, you'll probably want to attach labels/images to the data points. You can do this by generating a [metadata file](https://www.tensorflow.org/get_started/embedding_viz#metadata) containing the labels for each point and configuring the projector either by using our Python API, or manually constructing and saving a[projector\_config.pbtxt](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/tensorboard/plugins/projector/projector_config.proto) in the same directory as your checkpoint file.

## Setup

For in depth information on how to run TensorBoard and make sure you are logging all the necessary information, see [TensorBoard: Visualizing Learning](https://www.tensorflow.org/get_started/summaries_and_tensorboard).

To visualize your embeddings, there are 3 things you need to do:

1) Setup a 2D tensor that holds your embedding(s).

embedding\_var = tf.Variable(....)

2) Periodically save your model variables in a checkpoint in LOG\_DIR.

saver = tf.train.Saver()  
saver.save(session, os.path.join(LOG\_DIR, "model.ckpt"), step)

3) (Optional) Associate metadata with your embedding.

If you have any metadata (labels, images) associated with your embedding, you can tell TensorBoard about it either by directly storing a [projector\_config.pbtxt](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/tensorboard/plugins/projector/projector_config.proto) in the LOG\_DIR, or use our python API.

For instance, the following projector\_config.ptxt associates the word\_embedding tensor with metadata stored in $LOG\_DIR/metadata.tsv:

embeddings {  
  tensor\_name: 'word\_embedding'  
  metadata\_path: '$LOG\_DIR/metadata.tsv'  
}

The same config can be produced programmatically using the following code snippet:

from tensorflow.contrib.tensorboard.plugins import projector  
  
# Create randomly initialized embedding weights which will be trained.  
N = 10000 # Number of items (vocab size).  
D = 200 # Dimensionality of the embedding.  
embedding\_var = tf.Variable(tf.random\_normal([N,D]), name='word\_embedding')  
  
# Format: tensorflow/contrib/tensorboard/plugins/projector/projector\_config.proto  
config = projector.ProjectorConfig()  
  
# You can add multiple embeddings. Here we add only one.  
embedding = config.embeddings.add()  
embedding.tensor\_name = embedding\_var.name  
# Link this tensor to its metadata file (e.g. labels).  
embedding.metadata\_path = os.path.join(LOG\_DIR, 'metadata.tsv')  
  
# Use the same LOG\_DIR where you stored your checkpoint.  
summary\_writer = tf.summary.FileWriter(LOG\_DIR)  
  
# The next line writes a projector\_config.pbtxt in the LOG\_DIR. TensorBoard will  
# read this file during startup.  
projector.visualize\_embeddings(summary\_writer, config)

After running your model and training your embeddings, run TensorBoard and point it to the LOG\_DIR of the job.

tensorboard --logdir=LOG\_DIR

Then click on the Embeddings tab on the top pane and select the appropriate run (if there are more than one run).

## Metadata

Usually embeddings have metadata associated with it (e.g. labels, images). The metadata should be stored in a separate file outside of the model checkpoint since the metadata is not a trainable parameter of the model. The format should be a [TSV file](https://en.wikipedia.org/wiki/Tab-separated_values) (tab characters shown in red) with the first line containing column headers (shown in bold) and subsequent lines contain the metadata values:

**Word\tFrequency**  
Airplane\t345  
Car\t241  
...

There is no explicit key shared with the main data file; instead, the order in the metadata file is assumed to match the order in the embedding tensor. In other words, the first line is the header information and the (i+1)-th line in the metadata file corresponds to the i-th row of the embedding tensor stored in the checkpoint.

**Note:** If the TSV metadata file has only a single column, then we don’t expect a header row, and assume each row is the label of the embedding. We include this exception because it matches the commonly-used "vocab file" format.

### Images

If you have images associated with your embeddings, you will need to produce a single image consisting of small thumbnails of each data point. This is known as the [sprite image](https://www.google.com/webhp#q=what+is+a+sprite+image). The sprite should have the same number of rows and columns with thumbnails stored in row-first order: the first data point placed in the top left and the last data point in the bottom right:

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 |  |

Note in the example above that the last row doesn't have to be filled. For a concrete example of a sprite, see [this sprite image](https://www.tensorflow.org/images/mnist_10k_sprite.png) of 10,000 MNIST digits (100x100).

**Note:** We currently support sprites up to 8192px X 8192px.

After constructing the sprite, you need to tell the Embedding Projector where to find it:

embedding.sprite.image\_path = PATH\_TO\_SPRITE\_IMAGE  
# Specify the width and height of a single thumbnail.  
embedding.sprite.single\_image\_dim.extend([w, h])

## Interaction

The Embedding Projector has three panels:

1. Data panel on the top left, where you can choose the run, the embedding tensor and data columns to color and label points by.
2. Projections panel on the bottom left, where you choose the type of projection (e.g. PCA, t-SNE).
3. Inspector panel on the right side, where you can search for particular points and see a list of nearest neighbors.

### Projections

The Embedding Projector has three methods of reducing the dimensionality of a data set: two linear and one nonlinear. Each method can be used to create either a two- or three-dimensional view.

**Principal Component Analysis** A straightforward technique for reducing dimensions is Principal Component Analysis (PCA). The Embedding Projector computes the top 10 principal components. The menu lets you project those components onto any combination of two or three. PCA is a linear projection, often effective at examining global geometry.

**t-SNE** A popular non-linear dimensionality reduction technique is t-SNE. The Embedding Projector offers both two- and three-dimensional t-SNE views. Layout is performed client-side animating every step of the algorithm. Because t-SNE often preserves some local structure, it is useful for exploring local neighborhoods and finding clusters. Although extremely useful for visualizing high-dimensional data, t-SNE plots can sometimes be mysterious or misleading. See this [great article](http://distill.pub/2016/misread-tsne/) for how to use t-SNE effectively.

**Custom** You can also construct specialized linear projections based on text searches for finding meaningful directions in space. To define a projection axis, enter two search strings or regular expressions. The program computes the centroids of the sets of points whose labels match these searches, and uses the difference vector between centroids as a projection axis.

### Navigation

To explore a data set, you can navigate the views in either a 2D or a 3D mode, zooming, rotating, and panning using natural click-and-drag gestures. Clicking on a point causes the right pane to show an explicit textual list of nearest neighbors, along with distances to the current point. The nearest-neighbor points themselves are highlighted on the projection.

Zooming into the cluster gives some information, but it is sometimes more helpful to restrict the view to a subset of points and perform projections only on those points. To do so, you can select points in multiple ways:

1. After clicking on a point, its nearest neighbors are also selected.
2. After a search, the points matching the query are selected.
3. Enabling selection, clicking on a point and dragging defines a selection sphere.

After selecting a set of points, you can isolate those points for further analysis on their own with the "Isolate Points" button in the Inspector pane on the right hand side.

Selection of the nearest neighbors of “important” in a word embedding dataset.

The combination of filtering with custom projection can be powerful. Below, we filtered the 100 nearest neighbors of “politics” and projected them onto the “best” - “worst” vector as an x axis. The y axis is random.

You can see that on the right side we have “ideas”, “science”, “perspective”, “journalism” while on the left we have “crisis”, “violence” and “conflict”.

|  |  |
| --- | --- |
|  |  |
| Custom projection controls. | Custom projection of neighbors of "politics" onto "best" - "worst" vector. |

### Collaborative Features

To share your findings, you can use the bookmark panel in the bottom right corner and save the current state (including computed coordinates of any projection) as a small file. The Projector can then be pointed to a set of one or more of these files, producing the panel below. Other users can then walk through a sequence of bookmarks.

# TensorBoard: Graph Visualization

TensorFlow computation graphs are powerful but complicated. The graph visualization can help you understand and debug them. Here's an example of the visualization at work.

Visualization of a TensorFlow graph.

To see your own graph, run TensorBoard pointing it to the log directory of the job, click on the graph tab on the top pane and select the appropriate run using the menu at the upper left corner. For in depth information on how to run TensorBoard and make sure you are logging all the necessary information, see [TensorBoard: Visualizing Learning](https://www.tensorflow.org/get_started/summaries_and_tensorboard).

## Name scoping and nodes

Typical TensorFlow graphs can have many thousands of nodes--far too many to see easily all at once, or even to lay out using standard graph tools. To simplify, variable names can be scoped and the visualization uses this information to define a hierarchy on the nodes in the graph. By default, only the top of this hierarchy is shown. Here is an example that defines three operations under the hidden name scope using [tf.name\_scope](https://www.tensorflow.org/api_docs/python/tf/name_scope):

import tensorflow as tf  
  
with tf.name\_scope('hidden') as scope:  
  a = tf.constant(5, name='alpha')  
  W = tf.Variable(tf.random\_uniform([1, 2], -1.0, 1.0), name='weights')  
  b = tf.Variable(tf.zeros([1]), name='biases')

This results in the following three op names:

* hidden/alpha
* hidden/weights
* hidden/biases

By default, the visualization will collapse all three into a node labeled hidden. The extra detail isn't lost. You can double-click, or click on the orange + sign in the top right to expand the node, and then you'll see three subnodes for alpha, weights and biases.

Here's a real-life example of a more complicated node in its initial and expanded states.

|  |  |
| --- | --- |
|  |  |
| Initial view of top-level name scope pool\_1. Clicking on the orange + button on the top right or double-clicking on the node itself will expand it. | Expanded view of pool\_1 name scope. Clicking on the orange - button on the top right or double-clicking on the node itself will collapse the name scope. |

Grouping nodes by name scopes is critical to making a legible graph. If you're building a model, name scopes give you control over the resulting visualization. **The better your name scopes, the better your visualization.**

The figure above illustrates a second aspect of the visualization. TensorFlow graphs have two kinds of connections: data dependencies and control dependencies. Data dependencies show the flow of tensors between two ops and are shown as solid arrows, while control dependencies use dotted lines. In the expanded view (right side of the figure above) all the connections are data dependencies with the exception of the dotted line connecting CheckNumerics and control\_dependency.

There's a second trick to simplifying the layout. Most TensorFlow graphs have a few nodes with many connections to other nodes. For example, many nodes might have a control dependency on an initialization step. Drawing all edges between the init node and its dependencies would create a very cluttered view.

To reduce clutter, the visualization separates out all high-degree nodes to an auxiliary area on the right and doesn't draw lines to represent their edges. Instead of lines, we draw small node icons to indicate the connections. Separating out the auxiliary nodes typically doesn't remove critical information since these nodes are usually related to bookkeeping functions. See [Interaction](https://www.tensorflow.org/get_started/graph_viz#interaction) for how to move nodes between the main graph and the auxiliary area.

|  |  |
| --- | --- |
|  |  |
| Node conv\_1 is connected to save. Note the little save node icon on its right. | save has a high degree, and will appear as an auxiliary node. The connection with conv\_1 is shown as a node icon on its left. To further reduce clutter, since save has a lot of connections, we show the first 5 and abbreviate the others as ... 12 more. |

One last structural simplification is series collapsing. Sequential motifs--that is, nodes whose names differ by a number at the end and have isomorphic structures--are collapsed into a single stack of nodes, as shown below. For networks with long sequences, this greatly simplifies the view. As with hierarchical nodes, double-clicking expands the series. See [Interaction](https://www.tensorflow.org/get_started/graph_viz#interaction) for how to disable/enable series collapsing for a specific set of nodes.

|  |  |
| --- | --- |
|  |  |
| A collapsed view of a node sequence. | A small piece of the expanded view, after double-click. |

Finally, as one last aid to legibility, the visualization uses special icons for constants and summary nodes. To summarize, here's a table of node symbols:

| Symbol | Meaning |
| --- | --- |
|  | High-level node representing a name scope. Double-click to expand a high-level node. |
|  | Sequence of numbered nodes that are not connected to each other. |
|  | Sequence of numbered nodes that are connected to each other. |
|  | An individual operation node. |
|  | A constant. |
|  | A summary node. |
|  | Edge showing the data flow between operations. |
|  | Edge showing the control dependency between operations. |
|  | A reference edge showing that the outgoing operation node can mutate the incoming tensor. |

## Interaction

Navigate the graph by panning and zooming. Click and drag to pan, and use a scroll gesture to zoom. Double-click on a node, or click on its + button, to expand a name scope that represents a group of operations. To easily keep track of the current viewpoint when zooming and panning, there is a minimap in the bottom right corner.

To close an open node, double-click it again or click its - button. You can also click once to select a node. It will turn a darker color, and details about it and the nodes it connects to will appear in the info card at upper right corner of the visualization.

|  |  |
| --- | --- |
|  |  |
| Info card showing detailed information for the conv2 name scope. The inputs and outputs are combined from the inputs and outputs of the operation nodes inside the name scope. For name scopes no attributes are shown. | Info card showing detailed information for the DecodeRawoperation node. In addition to inputs and outputs, the card shows the device and the attributes associated with the current operation. |

TensorBoard provides several ways to change the visual layout of the graph. This doesn't change the graph's computational semantics, but it can bring some clarity to the network's structure. By right clicking on a node or pressing buttons on the bottom of that node's info card, you can make the following changes to its layout:

* Nodes can be moved between the main graph and the auxiliary area.
* A series of nodes can be ungrouped so that the nodes in the series do not appear grouped together. Ungrouped series can likewise be regrouped.

Selection can also be helpful in understanding high-degree nodes. Select any high-degree node, and the corresponding node icons for its other connections will be selected as well. This makes it easy, for example, to see which nodes are being saved--and which aren't.

Clicking on a node name in the info card will select it. If necessary, the viewpoint will automatically pan so that the node is visible.

Finally, you can choose two color schemes for your graph, using the color menu above the legend. The default Structure View shows structure: when two high-level nodes have the same structure, they appear in the same color of the rainbow. Uniquely structured nodes are gray. There's a second view, which shows what device the different operations run on. Name scopes are colored proportionally to the fraction of devices for the operations inside them.

The images below give an illustration for a piece of a real-life graph.

|  |  |
| --- | --- |
|  |  |
| Structure view: The gray nodes have unique structure. The orange conv1 and conv2 nodes have the same structure, and analogously for nodes with other colors. | Device view: Name scopes are colored proportionally to the fraction of devices of the operation nodes inside them. Here, purple means GPU and the green is CPU. |

## Tensor shape information

When the serialized GraphDef includes tensor shapes, the graph visualizer labels edges with tensor dimensions, and edge thickness reflects total tensor size. To include tensor shapes in the GraphDef pass the actual graph object (as in sess.graph) to the FileWriter when serializing the graph. The images below show the CIFAR-10 model with tensor shape information:

|  |
| --- |
|  |
| CIFAR-10 model with tensor shape information. |

## Runtime statistics

Often it is useful to collect runtime metadata for a run, such as total memory usage, total compute time, and tensor shapes for nodes. The code example below is a snippet from the train and test section of a modification of the [simple MNIST tutorial](https://www.tensorflow.org/get_started/mnist/beginners), in which we have recorded summaries and runtime statistics. See the [Summaries Tutorial](https://www.tensorflow.org/get_started/summaries_and_tensorboard#serializing_the_data) for details on how to record summaries. Full source is [here](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/mnist/mnist_with_summaries.py).

  # Train the model, and also write summaries.  
  # Every 10th step, measure test-set accuracy, and write test summaries  
  # All other steps, run train\_step on training data, & add training summaries  
  
  def feed\_dict(train):  
    """Make a TensorFlow feed\_dict: maps data onto Tensor placeholders."""  
    if train or FLAGS.fake\_data:  
      xs, ys = mnist.train.next\_batch(100, fake\_data=FLAGS.fake\_data)  
      k = FLAGS.dropout  
    else:  
      xs, ys = mnist.test.images, mnist.test.labels  
      k = 1.0  
    return {x: xs, y\_: ys, keep\_prob: k}  
  
  for i in range(FLAGS.max\_steps):  
    if i % 10 == 0:  # Record summaries and test-set accuracy  
      summary, acc = sess.run([merged, accuracy], feed\_dict=feed\_dict(False))  
      test\_writer.add\_summary(summary, i)  
      print('Accuracy at step %s: %s' % (i, acc))  
    else:  # Record train set summaries, and train  
      if i % 100 == 99:  # Record execution stats  
        run\_options = tf.RunOptions(trace\_level=tf.RunOptions.FULL\_TRACE)  
        run\_metadata = tf.RunMetadata()  
        summary, \_ = sess.run([merged, train\_step],  
                              feed\_dict=feed\_dict(True),  
                              options=run\_options,  
                              run\_metadata=run\_metadata)  
        train\_writer.add\_run\_metadata(run\_metadata, 'step%d' % i)  
        train\_writer.add\_summary(summary, i)  
        print('Adding run metadata for', i)  
      else:  # Record a summary  
        summary, \_ = sess.run([merged, train\_step], feed\_dict=feed\_dict(True))  
        train\_writer.add\_summary(summary, i)

This code will emit runtime statistics for every 100th step starting at step99.

When you launch tensorboard and go to the Graph tab, you will now see options under "Session runs" which correspond to the steps where run metadata was added. Selecting one of these runs will show you the snapshot of the network at that step, fading out unused nodes. In the controls on the left hand side, you will be able to color the nodes by total memory or total compute time. Additionally, clicking on a node will display the exact total memory, compute time, and tensor output sizes.

|  |  |  |
| --- | --- | --- |
|  |  |  |

# Versions of TensorFlow

The docs at root (i.e. in [tensorflow.org/api\_docs](https://www.tensorflow.org/api_docs/)) refer to the most recent stable branch (in this case, r1.1).

These branches are available on this site.

* [master](https://www.tensorflow.org/versions/master/)
* [r1.2](https://www.tensorflow.org/versions/r1.2/)
* [r1.1](https://www.tensorflow.org/api_docs/)
* [r1.0](https://www.tensorflow.org/versions/r1.0/)
* [r0.12](https://www.tensorflow.org/versions/r0.12/)
* [r0.11](https://www.tensorflow.org/versions/r0.11/)
* [r0.10](https://www.tensorflow.org/versions/r0.10/)

Earlier branches of the documentation can be found on [GitHub](http://github.com/tensorflow/tensorflow).