**Hello, Tensor World!**

Let’s analyze the Hello World script you ran. For reference, I’ve added the code below.

**import** tensorflow **as** tf

*# Create TensorFlow object called hello\_constant*

hello\_constant = tf.constant('Hello World!')

**with** tf.Session() **as** sess:

*# Run the tf.constant operation in the session*

output = sess.run(hello\_constant)

print(output)

**Tensor**

In TensorFlow, data isn’t stored as integers, floats, or strings. These values are encapsulated in an object called a tensor. In the case of hello\_constant = tf.constant('Hello World!'), hello\_constant is a 0-dimensional string tensor, but tensors come in a variety of sizes as shown below:

*# A is a 0-dimensional int32 tensor*

A = tf.constant(1234)

*# B is a 1-dimensional int32 tensor*

B = tf.constant([123,456,789])

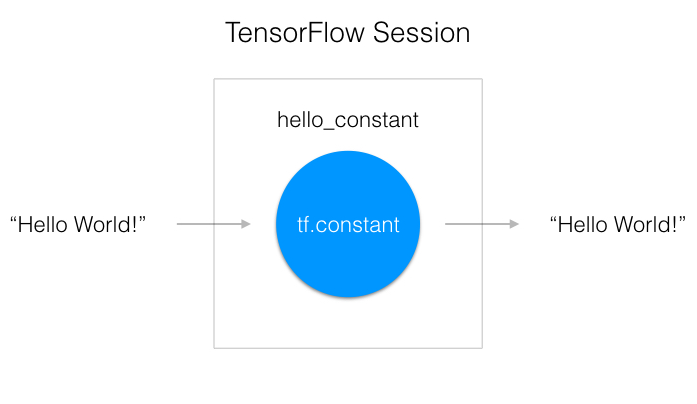
*# C is a 2-dimensional int32 tensor*

C = tf.constant([ [123,456,789], [222,333,444] ])

[**tf.constant()**](https://www.tensorflow.org/api_docs/python/tf/constant) is one of many TensorFlow operations you will use in this lesson. The tensor returned by [**tf.constant()**](https://www.tensorflow.org/api_docs/python/tf/constant) is called a constant tensor, because the value of the tensor never changes.

**Session**

TensorFlow’s api is built around the idea of a computational graph, a way of visualizing a mathematical process which you learned about in the MiniFlow lesson. Let’s take the TensorFlow code you ran and turn that into a graph:



A "TensorFlow Session", as shown above, is an environment for running a graph. The session is in charge of allocating the operations to GPU(s) and/or CPU(s), including remote machines. Let’s see how you use it.

**with** tf.Session() **as** sess:

output = sess.run(hello\_constant)

The code has already created the tensor, hello\_constant, from the previous lines. The next step is to evaluate the tensor in a session.

The code creates a session instance, sess, using [**tf.Session**](https://www.tensorflow.org/api_docs/python/tf/Session). The [**sess.run()**](https://www.tensorflow.org/api_docs/python/tf/Session#run) function then evaluates the tensor and returns the results.

# Input

In the last section, you passed a tensor into a session and it returned the result. What if you want to use a non-constant? This is where [**tf.placeholder()**](https://www.tensorflow.org/api_docs/python/tf/placeholder) and feed\_dict come into place. In this section, you'll go over the basics of feeding data into TensorFlow.

## tf.placeholder()

Sadly you can’t just set x to your dataset and put it in TensorFlow, because over time you'll want your TensorFlow model to take in different datasets with different parameters. You need [**tf.placeholder()**](https://www.tensorflow.org/api_docs/python/tf/placeholder)!

[**tf.placeholder()**](https://www.tensorflow.org/api_docs/python/tf/placeholder) returns a tensor that gets its value from data passed to the [**tf.session.run()**](https://www.tensorflow.org/api_docs/python/tf/Session#run)function, allowing you to set the input right before the session runs.

## Session’s feed\_dict

x = tf.placeholder(tf.string)

**with** tf.Session() **as** sess:

output = sess.run(x, feed\_dict={x: 'Hello World'})

Use the feed\_dict parameter in [**tf.session.run()**](https://www.tensorflow.org/api_docs/python/tf/Session#run) to set the placeholder tensor. The above example shows the tensor x being set to the string "Hello, world". It's also possible to set more than one tensor using feed\_dict as shown below.

x = tf.placeholder(tf.string)

y = tf.placeholder(tf.int32)

z = tf.placeholder(tf.float32)

**with** tf.Session() **as** sess:

output = sess.run(x, feed\_dict={x: 'Test String', y: 123, z: 45.67})

**Note:** If the data passed to the feed\_dict doesn’t match the tensor type and can’t be cast into the tensor type, you’ll get the error “ValueError: invalid literal for...”.

## Quiz

Let's see how well you understand [**tf.placeholder()**](https://www.tensorflow.org/api_docs/python/tf/placeholder) and feed\_dict. The code below throws an error, but I want you to make it return the number 123. Change line 11, so that the code returns the number 123.

# TensorFlow Math

Getting the input is great, but now you need to use it. You're going to use basic math functions that everyone knows and loves - add, subtract, multiply, and divide - with tensors. (There's many more math functions you can check out in the [**documentation**](https://www.tensorflow.org/api_docs/python/math_ops/).)

## Addition

x = tf.add(5, 2) *# 7*

You’ll start with the add function. The [**tf.add()**](https://www.tensorflow.org/api_guides/python/math_ops) function does exactly what you expect it to do. It takes in two numbers, two tensors, or one of each, and returns their sum as a tensor.

## Subtraction and Multiplication

Here’s an example with subtraction and multiplication.

x = tf.subtract(10, 4) *# 6*

y = tf.multiply(2, 5) *# 10*

The x tensor will evaluate to 6, because 10 - 4 = 6. The y tensor will evaluate to 10, because 2 \* 5 = 10. That was easy!

## Converting types

It may be necessary to convert between types to make certain operators work together. For example, if you tried the following, it would fail with an exception:

tf.subtract(tf.constant(2.0),tf.constant(1)) # Fails with ValueError: Tensor conversion requested dtype float32 for Tensor with dtype int32:

That's because the constant 1 is an integer but the constant 2.0 is a floating point value and subtractexpects them to match.

In cases like these, you can either make sure your data is all of the same type, or you can cast a value to another type. In this case, converting the 2.0 to an integer before subtracting, like so, will give the correct result:

tf.subtract(tf.cast(tf.constant(2.0), tf.int32), tf.constant(1)) # 1

## Quiz

Let's apply what you learned to convert an algorithm to TensorFlow. The code below is a simple algorithm using division and subtraction. Convert the following algorithm in regular Python to TensorFlow and print the results of the session. You can use [**tf.constant()**](https://www.tensorflow.org/api_guides/python/constant_op) for the values 10, 2, and 1.

Good job! You've accomplished a lot. In particular, you did the following:

* Ran operations in [**tf.Session**](https://www.tensorflow.org/api_docs/python/tf/Session).
* Created a constant tensor with [**tf.constant()**](https://www.tensorflow.org/api_docs/python/tf/constant).
* Used [**tf.placeholder()**](https://www.tensorflow.org/api_docs/python/tf/placeholder) and feed\_dict to get input.
* Applied the [**tf.add()**](https://www.tensorflow.org/api_docs/python/tf/add), [**tf.subtract()**](https://www.tensorflow.org/api_docs/python/tf/subtract), [**tf.multiply()**](https://www.tensorflow.org/api_docs/python/tf/multiply), and [**tf.divide()**](https://www.tensorflow.org/api_docs/python/tf/divide) functions using numeric data.
* Learned about casting between types with [**tf.cast()**](https://www.tensorflow.org/api_docs/python/tf/cast)

You know the basics of TensorFlow, so let's take a break and get back to the theory of neural networks. In the next few videos, you're going to learn about one of the most popular applications of neural networks - classification.

### **SCORES aka output values in the case of logistic regressions are also called as LOGITS. They are input to softmax.**

# Linear Function xW + b

logits = linear(features, w, b)

# Softmax

prediction = tf.nn.softmax(logits)

# Linear functions in TensorFlow

The most common operation in neural networks is calculating the linear combination of inputs, weights, and biases. As a reminder, we can write the output of the linear operation as

https://d17h27t6h515a5.cloudfront.net/topher/2017/February/58a4d8b3_linear-equation/linear-equation.gif

Here, **W** is a matrix of the weights connecting two layers. The output **y**, the input **x**, and the biases **b** are all vectors.

## Weights and Bias in TensorFlow

The goal of training a neural network is to modify weights and biases to best predict the labels. In order to use weights and bias, you'll need a Tensor that can be modified. This leaves out [**tf.placeholder()**](https://www.tensorflow.org/api_docs/python/tf/placeholder) and [**tf.constant()**](https://www.tensorflow.org/api_docs/python/tf/constant), since those Tensors can't be modified. This is where [**tf.Variable**](https://www.tensorflow.org/api_docs/python/tf/Variable) class comes in.

### tf.Variable()

x = tf.Variable(5)

The [**tf.Variable**](https://www.tensorflow.org/api_docs/python/tf/Variable) class creates a tensor with an initial value that can be modified, much like a normal Python variable. This tensor stores its state in the session, so you must initialize the state of the tensor manually. You'll use the [**tf.global\_variables\_initializer()**](https://www.tensorflow.org/api_docs/python/tf/global_variables_initializer) function to initialize the state of all the Variable tensors.

##### Initialization

init = tf.global\_variables\_initializer()

**with** tf.Session() **as** sess:

sess.run(init)

The [**tf.global\_variables\_initializer()**](https://www.tensorflow.org/api_docs/python/tf/global_variables_initializer) call returns an operation that will initialize all TensorFlow variables from the graph. You call the operation using a session to initialize all the variables as shown above. Using the [**tf.Variable**](https://www.tensorflow.org/api_docs/python/tf/Variable) class allows us to change the weights and bias, but an initial value needs to be chosen.

Initializing the weights with random numbers from a normal distribution is good practice. Randomizing the weights helps the model from becoming stuck in the same place every time you train it. You'll learn more about this in the next lesson, when you study gradient descent.

Similarly, choosing weights from a normal distribution prevents any one weight from overwhelming other weights. You'll use the [**tf.truncated\_normal()**](https://www.tensorflow.org/api_docs/python/tf/truncated_normal) function to generate random numbers from a normal distribution.

### tf.truncated\_normal()

n\_features = 120

n\_labels = 5

weights = tf.Variable(tf.truncated\_normal((n\_features, n\_labels)))

The [**tf.truncated\_normal()**](https://www.tensorflow.org/api_docs/python/tf/truncated_normal) function returns a tensor with random values from a normal distribution whose magnitude is no more than 2 standard deviations from the mean.

Since the weights are already helping prevent the model from getting stuck, you don't need to randomize the bias. Let's use the simplest solution, setting the bias to 0.

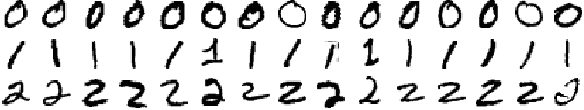
### tf.zeros()

n\_labels = 5

bias = tf.Variable(tf.zeros(n\_labels))

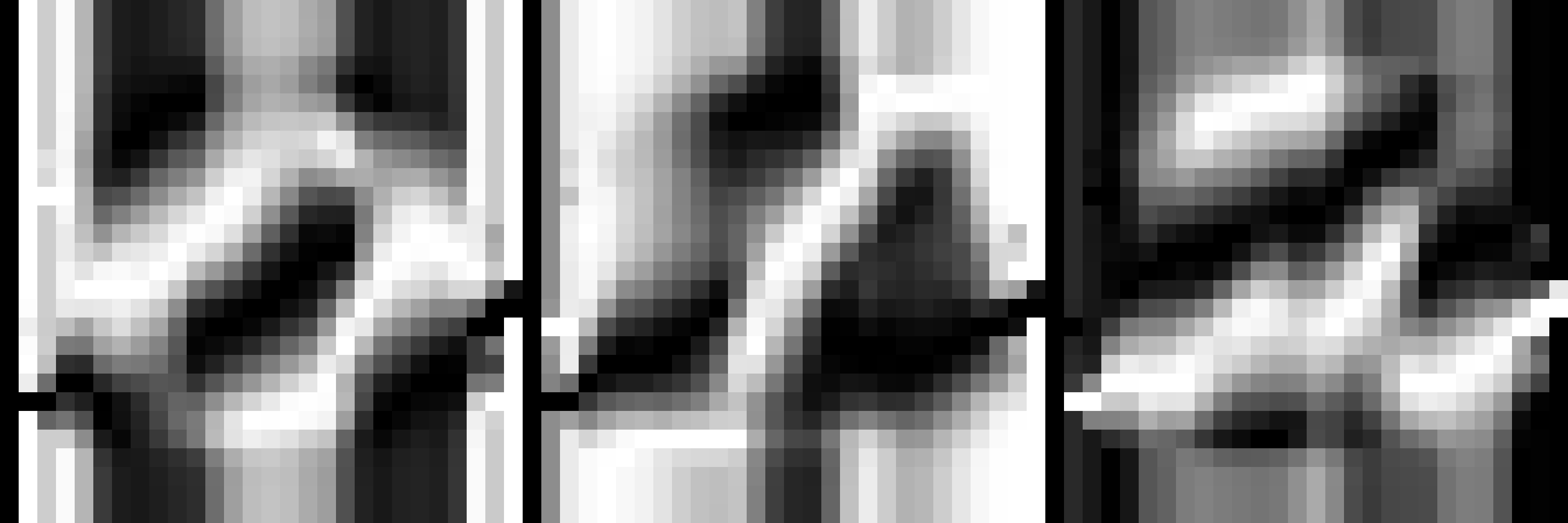
The [**tf.zeros()**](https://www.tensorflow.org/api_docs/python/tf/zeros) function returns a tensor with all zeros.

## Linear Classifier Quiz



A subset of the MNIST dataset

You'll be classifying the handwritten numbers 0, 1, and 2 from the MNIST dataset using TensorFlow. The above is a small sample of the data you'll be training on. Notice how some of the 1s are written with a [**serif**](https://en.wikipedia.org/wiki/Serif) at the top and at different angles. The similarities and differences will play a part in shaping the weights of the model.



Left: Weights for labeling 0. Middle: Weights for labeling 1. Right: Weights for labeling 2.

The images above are trained weights for each label (0, 1, and 2). The weights display the unique properties of each digit they have found. Complete this quiz to train your own weights using the MNIST dataset.

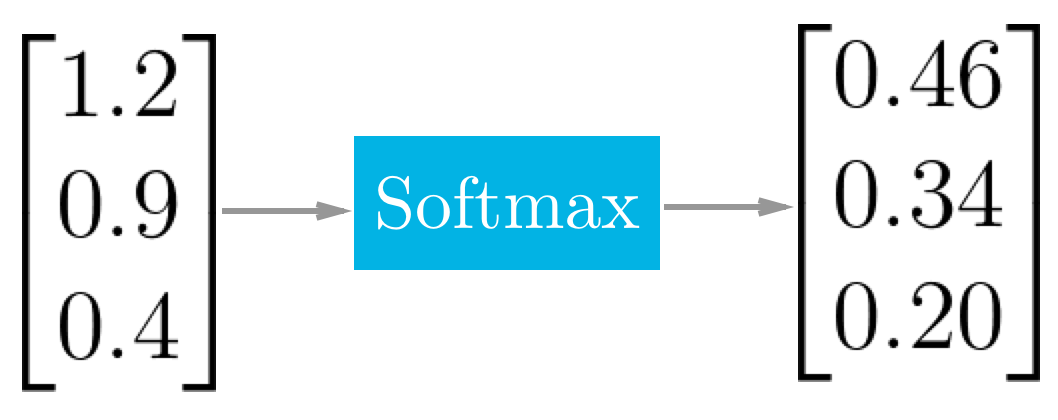
### Instructions

1. Open quiz.py.
   1. Implement get\_weights to return a [**tf.Variable**](https://www.tensorflow.org/api_docs/python/tf/Variable) of weights
   2. Implement get\_biases to return a [**tf.Variable**](https://www.tensorflow.org/api_docs/python/tf/Variable) of biases
   3. Implement xW + b in the linear function
2. Open sandbox.py
   1. Initialize all weights

Since xW in xW + b is matrix multiplication, you have to use the [**tf.matmul()**](https://www.tensorflow.org/api_docs/python/tf/matmul) function instead of [**tf.multiply()**](https://www.tensorflow.org/api_docs/python/tf/multiply). Don't forget that order matters in matrix multiplication, so tf.matmul(a,b) is not the same as tf.matmul(b,a).

# TensorFlow Softmax

You might remember in the Intro to TFLearn lesson we used the softmax function to calculate class probabilities as output from the network. The softmax function squashes it's inputs, typically called **logits** or **logit scores**, to be between 0 and 1 and also normalizes the outputs such that they all sum to 1. This means the output of the softmax function is equivalent to a categorical probability distribution. It's the perfect function to use as the output activation for a network predicting multiple classes.



Example of the softmax function at work.

## TensorFlow Softmax

We're using TensorFlow to build neural networks and, appropriately, there's a function for calculating softmax.

x = tf.nn.softmax([2.0, 1.0, 0.2])

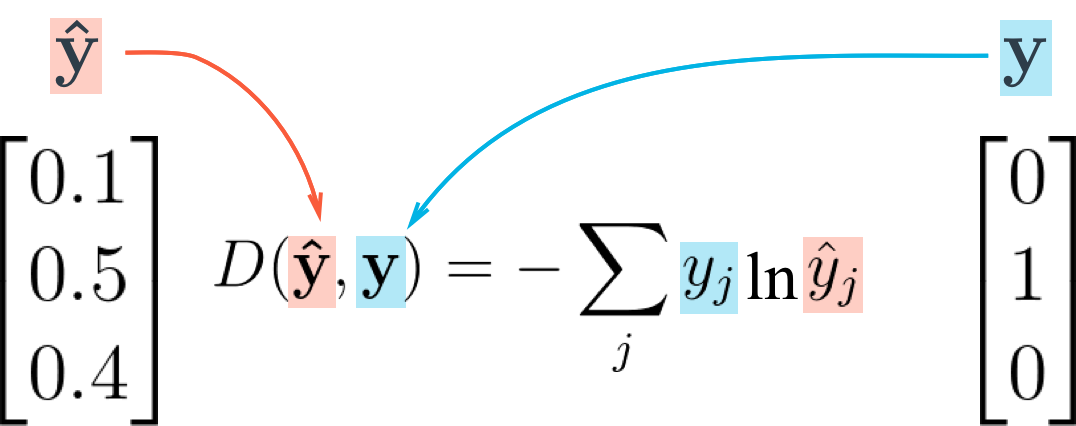
Easy as that! [**tf.nn.softmax()**](https://www.tensorflow.org/api_docs/python/tf/nn/softmax) implements the softmax function for you. It takes in logits and returns softmax activations.

## Quiz

Use the softmax function in the quiz below to return the softmax of the logits.

# Cross Entropy in TensorFlow

In the Intro to TFLearn lesson we discussed using cross entropy as the cost function for classification with one-hot encoded labels. Again, TensorFlow has a function to do the cross entropy calculations for us.



Cross entropy loss function

Let's take what you learned from the video and create a cross entropy function in TensorFlow. To create a cross entropy function in TensorFlow, you'll need to use two new functions:

* [**tf.reduce\_sum()**](https://www.tensorflow.org/api_docs/python/tf/reduce_sum)
* [**tf.log()**](https://www.tensorflow.org/api_docs/python/tf/log)

**Reduce Sum**

x = tf.reduce\_sum([1, 2, 3, 4, 5]) *# 15*

The [**tf.reduce\_sum()**](https://www.tensorflow.org/api_docs/python/tf/reduce_sum) function takes an array of numbers and sums them together.

**Natural Log**

x = tf.log(100) *# 4.60517*

This function does exactly what you would expect it to do. [**tf.log()**](https://www.tensorflow.org/api_docs/python/tf/log) takes the natural log of a number.

**Quiz**

Print the cross entropy using softmax\_data and one\_hot\_encod\_label.

[**(Alternative link for users in China.)**](http://www.tensorfly.cn/)

## Mini-batching

In this section, you'll go over what mini-batching is and how to apply it in TensorFlow.

Mini-batching is a technique for training on subsets of the dataset instead of all the data at one time. This provides the ability to train a model, even if a computer lacks the memory to store the entire dataset.

Mini-batching is computationally inefficient, since you can't calculate the loss simultaneously across all samples. However, this is a small price to pay in order to be able to run the model at all.

It's also quite useful combined with SGD. The idea is to randomly shuffle the data at the start of each epoch, then create the mini-batches. For each mini-batch, you train the network weights with gradient descent. Since these batches are random, you're performing SGD with each batch.

Let's look at the MNIST dataset with weights and a bias to see if your machine can handle it.

**from** tensorflow.examples.tutorials.mnist **import** input\_data

**import** tensorflow **as** tf

n\_input = 784 *# MNIST data input (img shape: 28\*28)*

n\_classes = 10 *# MNIST total classes (0-9 digits)*

*# Import MNIST data*

mnist = input\_data.read\_data\_sets('/datasets/ud730/mnist', one\_hot=**True**)

*# The features are already scaled and the data is shuffled*

train\_features = mnist.train.images

test\_features = mnist.test.images

train\_labels = mnist.train.labels.astype(np.float32)

test\_labels = mnist.test.labels.astype(np.float32)

*# Weights & bias*

weights = tf.Variable(tf.random\_normal([n\_input, n\_classes]))

bias = tf.Variable(tf.random\_normal([n\_classes]))

### Question 1

Calculate the memory size of train\_features, train\_labels, weights, and bias in bytes. Ignore memory for overhead, just calculate the memory required for the stored data.

You may have to look up how much memory a float32 requires, using [**this link**](https://en.wikipedia.org/wiki/Single-precision_floating-point_format).

train\_features Shape: (55000, 784) Type: float32

train\_labels Shape: (55000, 10) Type: float32

weights Shape: (784, 10) Type: float32

bias Shape: (10,) Type: float32

How many bytes of memory does train\_features need?

172480000

RESET

How many bytes of memory does train\_labels need?

2200000

RESET

How many bytes of memory does weights need?

31360

RESET

How many bytes of memory does bias need?

40

RESET

The total memory space required for the inputs, weights and bias is around 174 megabytes, which isn't that much memory. You could train this whole dataset on most CPUs and GPUs.

But larger datasets that you'll use in the future measured in gigabytes or more. It's possible to purchase more memory, but it's expensive. A Titan X GPU with 12 GB of memory costs over $1,000.

Instead, in order to run large models on your machine, you'll learn how to use mini-batching.

Let's look at how you implement mini-batching in TensorFlow.

## TensorFlow Mini-batching

In order to use mini-batching, you must first divide your data into batches.

Unfortunately, it's sometimes impossible to divide the data into batches of exactly equal size. For example, imagine you'd like to create batches of 128 samples each from a dataset of 1000 samples. Since 128 does not evenly divide into 1000, you'd wind up with 7 batches of 128 samples, and 1 batch of 104 samples. (7\*128 + 1\*104 = 1000)

In that case, the size of the batches would vary, so you need to take advantage of TensorFlow's [**tf.placeholder()**](https://www.tensorflow.org/api_docs/python/tf/placeholder) function to receive the varying batch sizes.

Continuing the example, if each sample had n\_input = 784 features and n\_classes = 10 possible labels, the dimensions for features would be [None, n\_input] and labels would be [None, n\_classes].

*# Features and Labels*

features = tf.placeholder(tf.float32, [**None**, n\_input])

labels = tf.placeholder(tf.float32, [**None**, n\_classes])

What does None do here?

The None dimension is a placeholder for the batch size. At runtime, TensorFlow will accept any batch size greater than 0.

Going back to our earlier example, this setup allows you to feed features and labels into the model as either the batches of 128 samples or the single batch of 104 samples.

### Question 2

Use the parameters below, how many batches are there, and what is the last batch size?

features is (50000, 400)

labels is (50000, 10)

batch\_size is 128

How many batches are there?

391

RESET

What is the last batch size?

80

RESET

Now that you know the basics, let's learn how to implement mini-batching.

### Question 3

Implement the batches function to batch features and labels. The function should return each batch with a maximum size of batch\_size. To help you with the quiz, look at the following example output of a working batches function.

*# 4 Samples of features*

example\_features = [

['F11','F12','F13','F14'],

['F21','F22','F23','F24'],

['F31','F32','F33','F34'],

['F41','F42','F43','F44']]

*# 4 Samples of labels*

example\_labels = [

['L11','L12'],

['L21','L22'],

['L31','L32'],

['L41','L42']]

example\_batches = batches(3, example\_features, example\_labels)

The example\_batches variable would be the following:

[

*# 2 batches:*

*# First is a batch of size 3.*

*# Second is a batch of size 1*

[

*# First Batch is size 3*

[

*# 3 samples of features.*

*# There are 4 features per sample.*

['F11', 'F12', 'F13', 'F14'],

['F21', 'F22', 'F23', 'F24'],

['F31', 'F32', 'F33', 'F34']

], [

*# 3 samples of labels.*

*# There are 2 labels per sample.*

['L11', 'L12'],

['L21', 'L22'],

['L31', 'L32']

]

], [

*# Second Batch is size 1.*

*# Since batch size is 3, there is only one sample left from the 4 samples.*

[

*# 1 sample of features.*

['F41', 'F42', 'F43', 'F44']

], [

*# 1 sample of labels.*

['L41', 'L42']

]

]

]

Implement the batches function in the "quiz.py" file below.

