**Intro to TensorFlow**

Now that you are an expert in Neural Networks, Convolutional Neural Networks, and Keras, you're more than ready to learn TensorFlow. In the following sections of this Nanodegree Program, you will be using Keras and TensorFlow alternately. This lesson will teach you what you need to know of TensorFlow, and give you some exercises to practice.

This lesson will build up on the knowledge from the [**Deep Neural Networks**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/75cb91c8-003b-458c-8bac-653a17cd1a97) lesson. If you need to refresh your memory on any of the topics, such as *Linear Functions, Softmax, Cross Entropy, Batching, Epochs*, etc., feel free to go back and watch them again.

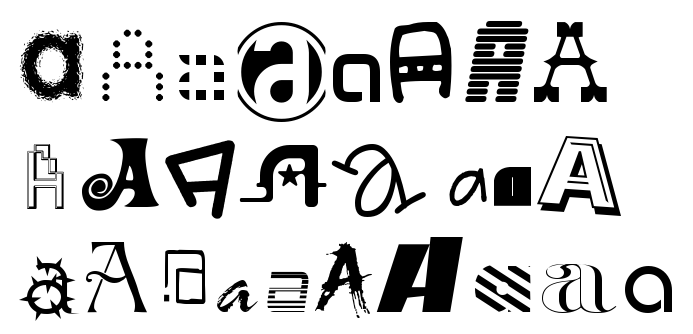
* [**Linear Functions**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/55e267a6-888b-4093-90cb-6b131ad00c6d)
* [**Softmax**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/9e1364a8-e8b4-4eac-be12-4d44a139f721)
* [**Cross Entropy**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/760235e0-a3ec-4e56-8cdb-56d762886690)
* [**Batching and Epochs**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/cad9b57b-ff21-4e06-8a4b-b53bebe5e2b6)

Enjoy!



Throughout this lesson, you'll apply your knowledge of neural networks on real datasets using [**TensorFlow**](https://www.tensorflow.org/) [**(link for China)**](http://www.tensorfly.cn/), an open source Deep Learning library created by Google.

You’ll use TensorFlow to classify images from the notMNIST dataset - a dataset of images of English letters from A to J. You can see a few example images below.



Your goal is to automatically detect the letter based on the image in the dataset. You’ll be working on your own computer for this lab, so, first things first, install TensorFlow!

# Install

As usual, we'll be using Conda to install TensorFlow. You might already have a TensorFlow environment, but check to make sure you have all the necessary packages.

## OS X or Linux

Run the following commands to setup your environment:

conda create -n tensorflow python=3.5

source activate tensorflow

conda install pandas matplotlib jupyter notebook scipy scikit-learn

pip install tensorflow

## Windows

And installing on Windows. In your console or Anaconda shell,

conda create -n tensorflow python=3.5

activate tensorflow

conda install pandas matplotlib jupyter notebook scipy scikit-learn

pip install tensorflow

## Hello, world!

Try running the following code in your Python console to make sure you have TensorFlow properly installed. The console will print "Hello, world!" if TensorFlow is installed. Don’t worry about understanding what it does. You’ll learn about it in the next section.

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

*# Create TensorFlow object called tensor*

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)

# 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])

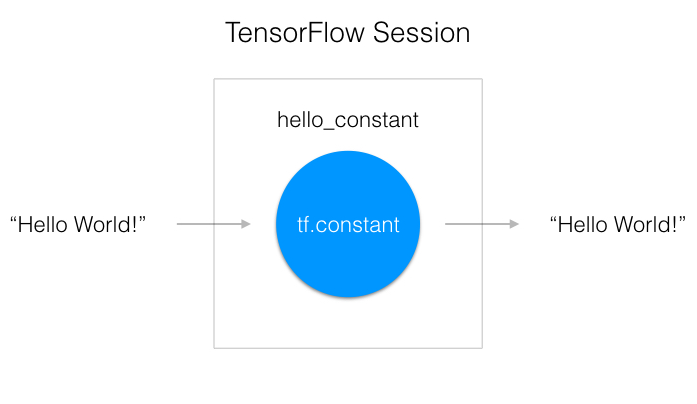
*# C is a 2-dimensional int32 tensor*

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.

**Note:** The quizzes are running TensorFlow version 0.12.1. However, all the code used in this course is compatible with version 1.0. We'll be upgrading our in class quizzes to the newest version in the near future.

# Solution is available in the other "Quiz.py" tab

import tensorflow as tf

def run():

output = None

x = tf.placeholder(tf.int32)

with tf.Session() as sess:

# TODO: Feed the x tensor 123

#x =123

output = sess.run(x,feed\_dict={x: 123})

return output

# 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 subtract expects 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.

# Solution is available in the other "solution.py" tab

import tensorflow as tf

# TODO: Convert the following to TensorFlow:

x = 10

y = 2

z = x/y - 1

x = tf.constant(x)

y= tf.constant(y)

z = tf.subtract(tf.divide(x,y),tf.cast(tf.constant(1),tf.float64))

# TODO: Print z from a session

with tf.Session() as sess:

output = sess.run(z)

print(output)

# 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.

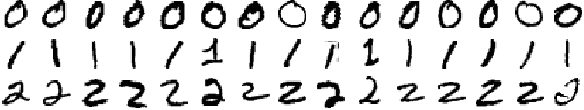
### tf.zeros()

n\_labels = 5

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

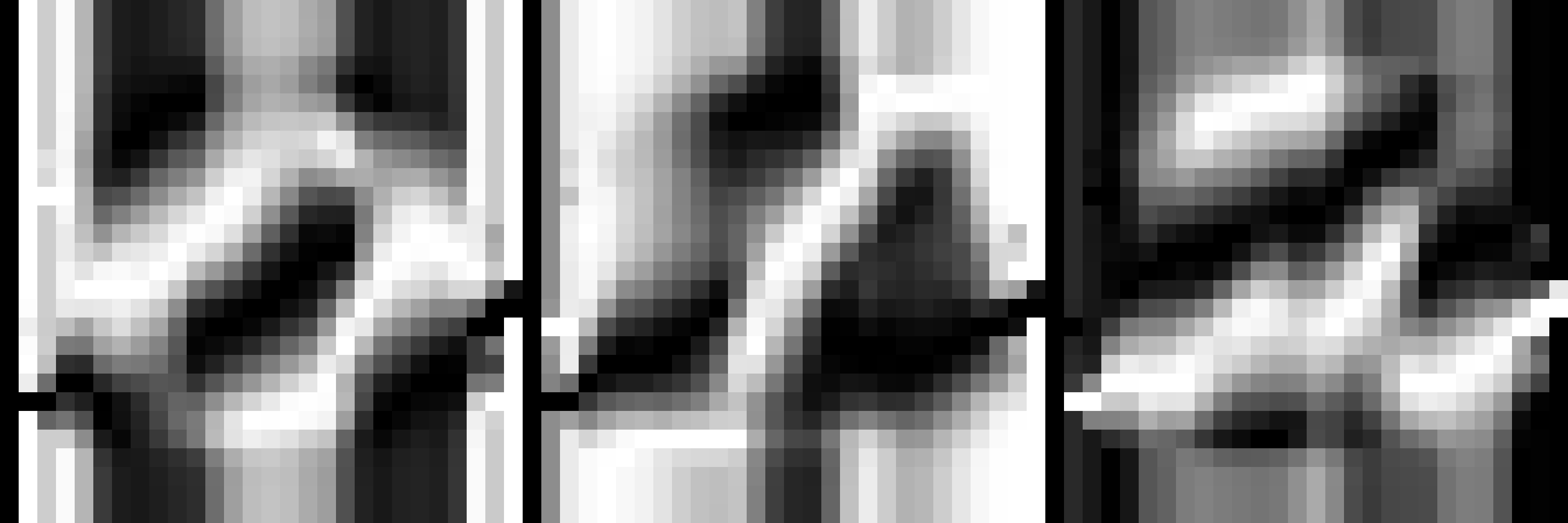
The [**tf.zeros()**](https://www.tensorflow.org/api_docs/python/tf/zeros) function returns a tensor with all zeros.

## 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).

# Solution is available in the other "**quiz.py**" tab

import tensorflow as tf

def get\_weights(n\_features, n\_labels):

"""

Return TensorFlow weights

:param n\_features: Number of features

:param n\_labels: Number of labels

:return: TensorFlow weights

"""

# TODO: Return weights

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

def get\_biases(n\_labels):

"""

Return TensorFlow bias

:param n\_labels: Number of labels

:return: TensorFlow bias

"""

# TODO: Return biases

return tf.Variable(tf.zeros(n\_labels))

def linear(input, w, b):

"""

Return linear function in TensorFlow

:param input: TensorFlow input

:param w: TensorFlow weights

:param b: TensorFlow biases

:return: TensorFlow linear function

"""

# TODO: Linear Function (xW + b)

return tf.add(tf.matmul(input,w), b)

**sandbox.py**

import tensorflow as tf

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

from quiz import get\_weights, get\_biases, linear

def mnist\_features\_labels(n\_labels):

"""

Gets the first <n> labels from the MNIST dataset

:param n\_labels: Number of labels to use

:return: Tuple of feature list and label list

"""

mnist\_features = []

mnist\_labels = []

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

# In order to make quizzes run faster, we're only looking at 10000 images

for mnist\_feature, mnist\_label in zip(\*mnist.train.next\_batch(10000)):

# Add features and labels if it's for the first <n>th labels

if mnist\_label[:n\_labels].any():

mnist\_features.append(mnist\_feature)

mnist\_labels.append(mnist\_label[:n\_labels])

return mnist\_features, mnist\_labels

# Number of features (28\*28 image is 784 features)

n\_features = 784

# Number of labels

n\_labels = 3

# Features and Labels

features = tf.placeholder(tf.float32)

labels = tf.placeholder(tf.float32)

# Weights and Biases

w = get\_weights(n\_features, n\_labels)

b = get\_biases(n\_labels)

# Linear Function xW + b

logits = linear(features, w, b)

# Training data

train\_features, train\_labels = mnist\_features\_labels(n\_labels)

with tf.Session() as session:

# TODO: Initialize session variables

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

# Softmax

prediction = tf.nn.softmax(logits)

# Cross entropy

# This quantifies how far off the predictions were.

# You'll learn more about this in future lessons.

cross\_entropy = -tf.reduce\_sum(labels \* tf.log(prediction), reduction\_indices=1)

# Training loss

# You'll learn more about this in future lessons.

loss = tf.reduce\_mean(cross\_entropy)

# Rate at which the weights are changed

# You'll learn more about this in future lessons.

learning\_rate = 0.08

# Gradient Descent

# This is the method used to train the model

# You'll learn more about this in future lessons.

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

# Run optimizer and get loss

\_, l = session.run(

[optimizer, loss],

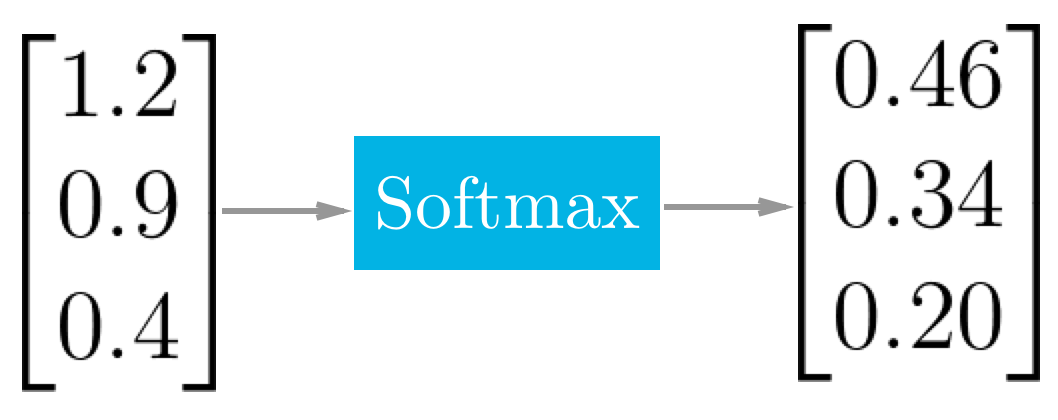
feed\_dict={features: train\_features, labels: train\_labels})

# Print loss

print('Loss: {}'.format(l))

# TensorFlow Softmax

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.

**Softmax.py**

import tensorflow as tf

def run():

output = None

logit\_data = [2.0, 1.0, 0.1]

logits = tf.placeholder(tf.float32)

# TODO: Calculate the softmax of the logits

# softmax =

softmax = tf.nn.softmax(logits)

with tf.Session() as sess:

# TODO: Feed in the logit data

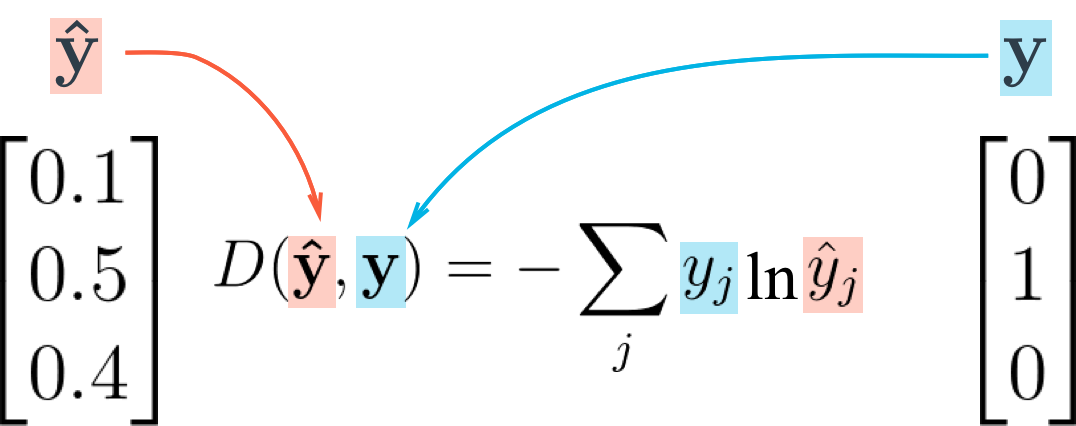
# output = sess.run(softmax, )

output = sess.run(softmax, feed\_dict={logits: logit\_data})

return output

# Cross Entropy in TensorFlow

As with the softmax function, 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.

import tensorflow as tf

softmax\_data = [0.7, 0.2, 0.1]

one\_hot\_data = [1.0, 0.0, 0.0]

softmax = tf.placeholder(tf.float32)

one\_hot = tf.placeholder(tf.float32)

# TODO: Print cross entropy from session

cross\_entropy = -tf.reduce\_sum(tf.multiply(one\_hot,tf.log(softmax)))

with tf.Session() as sess:

print(sess.run(cross\_entropy, feed\_dict={one\_hot : one\_hot\_data ,softmax: softmax\_data}))

## 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 =(55000\*784\*4)

How many bytes of memory does train\_labels need?

2200000 =(55000\*10\*4)

How many bytes of memory does weights need?

31360=(784\*10\*4)

How many bytes of memory does bias need?

40 =(10\*4)

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 = (50000/128 = 390) +1

What is the last batch size?

80 = (50000/128 = 390 whole = 49920 ) 50000-49920=80

**Batch\_size.py**

import math

def batches(batch\_size, features, labels):

assert len(features) == len(labels)

output\_batches = []

sample\_size = len(features)

for start\_i in range(0, sample\_size, batch\_size):

end\_i = start\_i+batch\_size

batch = [features[start\_i:end\_i],labels[start\_i:end\_i] ]

output\_batches.append(batch)

return output\_batches

from pprint import pprint

# 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']]

# PPrint prints data structures like 2d arrays, so they are easier to read

pprint(batches(3, example\_features, example\_labels))

Let's use mini-batching to feed batches of MNIST features and labels into a linear model.

Set the batch size and run the optimizer over all the batches with the batches function. The recommended batch size is 128. If you have memory restrictions, feel free to make it smaller.

Mnist.py

# -\*- coding: utf-8 -\*-

"""

Created on Tue Aug 29 21:17:14 2017

@author: shubra

"""

import math

def batches(batch\_size, features, labels):

"""

Create batches of features and labels

:param batch\_size: The batch size

:param features: List of features

:param labels: List of labels

:return: Batches of (Features, Labels)

"""

assert len(features) == len(labels)

# TODO: Implement batching

output\_batches = []

sample\_size = len(features)

for start\_i in range(0, sample\_size, batch\_size):

end\_i = start\_i + batch\_size

batch = [features[start\_i:end\_i], labels[start\_i:end\_i]]

output\_batches.append(batch)

return output\_batches

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

import tensorflow as tf

import numpy as np

learning\_rate = 0.001

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)

# Features and Labels

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

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

# Weights & bias

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

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

# Logits - xW + b

logits = tf.add(tf.matmul(features, weights), bias)

# Define loss and optimizer

cost = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=labels))

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

# Calculate accuracy

correct\_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

# TODO: Set batch size

batch\_size = 128

assert batch\_size is not None, 'You must set the batch size'

init = tf.global\_variables\_initializer()

with tf.Session() as sess:

sess.run(init)

# TODO: Train optimizer on all batches

# for batch\_features, batch\_labels in \_\_\_\_\_\_

for batch\_features, batch\_labels in batches(batch\_size, train\_features, train\_labels):

sess.run(optimizer, feed\_dict={features: batch\_features, labels: batch\_labels})

# Calculate accuracy for test dataset

test\_accuracy = sess.run(

accuracy,

feed\_dict={features: test\_features, labels: test\_labels})

print('Test Accuracy: {}'.format(test\_accuracy))

## Epochs

An epoch is a single forward and backward pass of the whole dataset. This is used to increase the accuracy of the model without requiring more data. This section will cover epochs in TensorFlow and how to choose the right number of epochs.

The following TensorFlow code trains a model using 10 epochs.

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

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

**import** numpy **as** np

**from** helper **import** batches *# Helper function created in Mini-batching section*

**def** **print\_epoch\_stats**(epoch\_i, sess, last\_features, last\_labels):

"""

Print cost and validation accuracy of an epoch

"""

current\_cost = sess.run(

cost,

feed\_dict={features: last\_features, labels: last\_labels})

valid\_accuracy = sess.run(

accuracy,

feed\_dict={features: valid\_features, labels: valid\_labels})

print('Epoch: {:<4} - Cost: {:<8.3} Valid Accuracy: {:<5.3}'.format(

epoch\_i,

current\_cost,

valid\_accuracy))

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

valid\_features = mnist.validation.images

test\_features = mnist.test.images

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

valid\_labels = mnist.validation.labels.astype(np.float32)

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

*# Features and Labels*

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

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

*# Weights & bias*

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

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

*# Logits - xW + b*

logits = tf.add(tf.matmul(features, weights), bias)

*# Define loss and optimizer*

learning\_rate = tf.placeholder(tf.float32)

cost = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=labels))

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

*# Calculate accuracy*

correct\_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

init = tf.global\_variables\_initializer()

batch\_size = 128

epochs = 10

learn\_rate = 0.001

train\_batches = batches(batch\_size, train\_features, train\_labels)

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

sess.run(init)

*# Training cycle*

**for** epoch\_i **in** range(epochs):

*# Loop over all batches*

**for** batch\_features, batch\_labels **in** train\_batches:

train\_feed\_dict = {

features: batch\_features,

labels: batch\_labels,

learning\_rate: learn\_rate}

sess.run(optimizer, feed\_dict=train\_feed\_dict)

*# Print cost and validation accuracy of an epoch*

print\_epoch\_stats(epoch\_i, sess, batch\_features, batch\_labels)

*# Calculate accuracy for test dataset*

test\_accuracy = sess.run(

accuracy,

feed\_dict={features: test\_features, labels: test\_labels})

print('Test Accuracy: {}'.format(test\_accuracy))

Running the code will output the following:

Epoch: 0 - Cost: 11.0 Valid Accuracy: 0.204

Epoch: 1 - Cost: 9.95 Valid Accuracy: 0.229

Epoch: 2 - Cost: 9.18 Valid Accuracy: 0.246

Epoch: 3 - Cost: 8.59 Valid Accuracy: 0.264

Epoch: 4 - Cost: 8.13 Valid Accuracy: 0.283

Epoch: 5 - Cost: 7.77 Valid Accuracy: 0.301

Epoch: 6 - Cost: 7.47 Valid Accuracy: 0.316

Epoch: 7 - Cost: 7.2 Valid Accuracy: 0.328

Epoch: 8 - Cost: 6.96 Valid Accuracy: 0.342

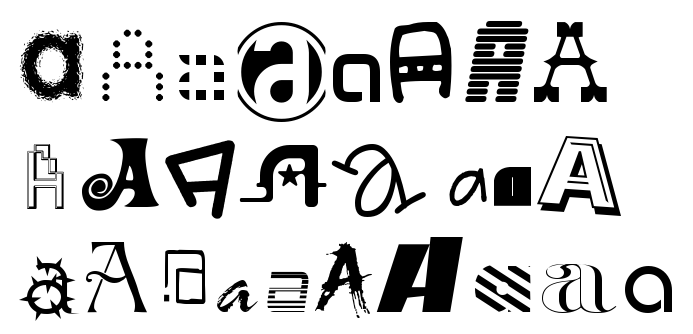
Epoch: 9 - Cost: 6.73 Valid Accuracy: 0.36

Test Accuracy: 0.3801000118255615

Each epoch attempts to move to a lower cost, leading to better accuracy.

This model continues to improve accuracy up to Epoch 9. Let's increase the number of epochs to 100.

# TensorFlow Neural Network Lab

**[](http://yaroslavvb.blogspot.com/2011/09/notmnist-dataset.html)**

# TensorFlow Lab

We've prepared a Jupyter notebook that will guide you through the process of creating a single layer neural network in TensorFlow. You'll implement data normalization, then build and train the network with TensorFlow.

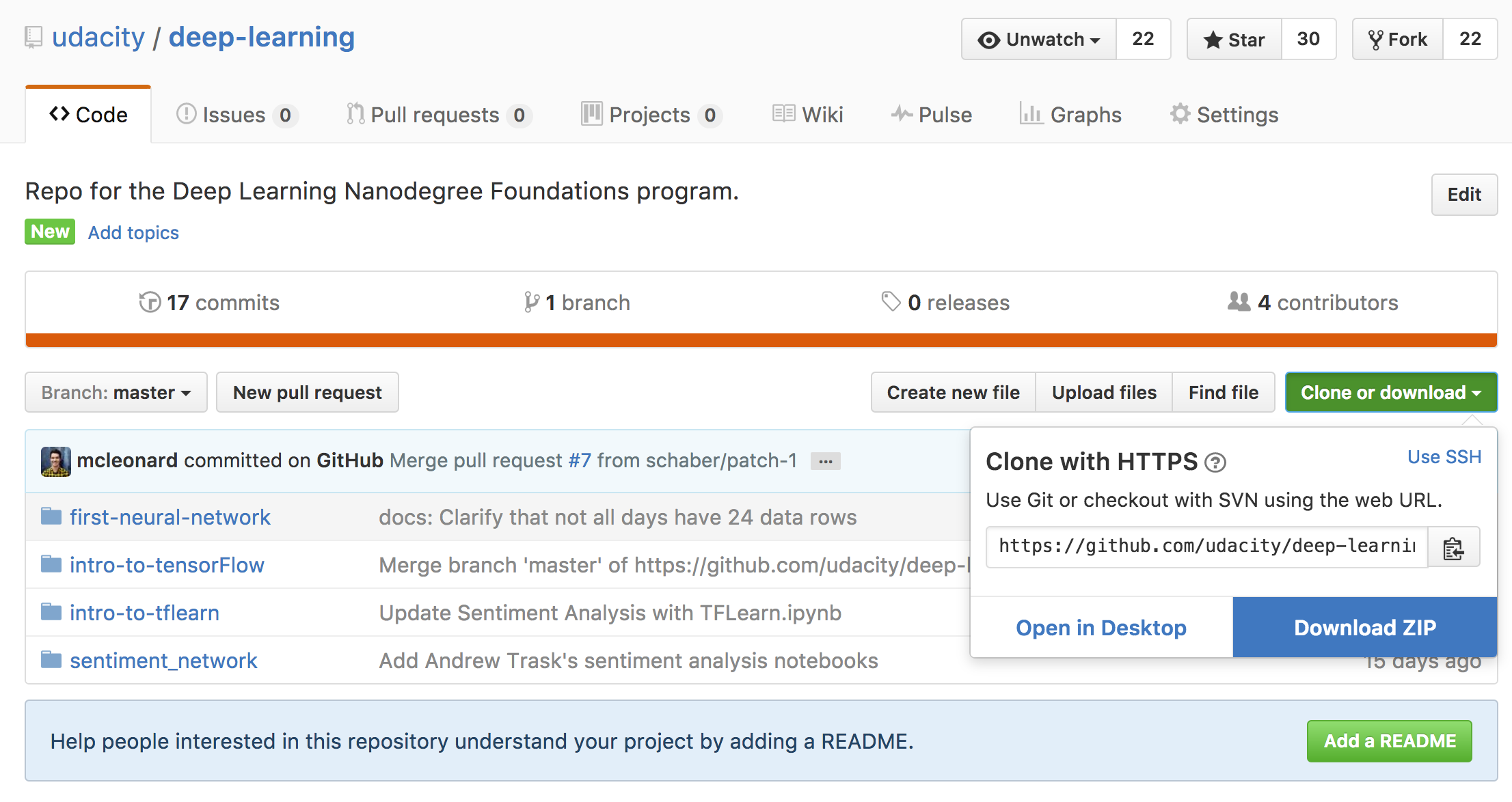
## Getting the notebook

The notebook and all related files are available from [**our GitHub repository**](https://github.com/udacity/deep-learning). Either clone the repository or download it as a Zip file.

Use Git to clone the repository.

git clone https://github.com/udacity/deep-learning.git

If you're unfamiliar with Git and GitHub, I highly recommend checking out [**our course**](https://www.udacity.com/course/how-to-use-git-and-github--ud775). If you'd rather not use Git, you can download the repository as a Zip archive. You can find [**the repo here**](https://github.com/udacity/deep-learning).



Download the repository contents as a Zip file using the green button on the top right.

If you download the Zip file, be sure to extract it (usually just double clicking). The most recent versions of all our code will be available from the repository, so it's the best place to get up-to-date files.

Once you have the repo cloned or downloaded, change directories into the repo, then the intro-to-tensorflowdirectory. In there you'll find the lab notebook, as well as Conda environment files for installing all the necessary packages.

## Windows Instructions

We've provided a Conda environment file for you to easily install all the necessary packages. In the intro-to-tensorflow directory, enter

conda env create -f environment\_win.yml

This will create an environment called dlnd-tf-lab. You can enter the environment with the command

activate dlnd-tf-lab

All the necessary packages should be installed for you.

## OS X and Linux Instructions

We've provided a Conda environment file for you to easily install all the necessary packages. In the intro-to-tensorflow directory, enter

conda env create -f environment.yml

This will create an environment called dlnd-tf-lab. You can enter the environment with the command

source activate dlnd-tf-lab

All the necessary packages should be installed for you.

## View The Notebook

In the directory with the notebook file, start your Jupyter notebook server

jupyter notebook

This should open a browser window for you. If it doesn't, go to [**http://localhost:8888/tree**](http://localhost:8888/tree). Although, the port number might be different if you have other notebook servers running, so try 8889 instead of 8888 if you can't find the right server.

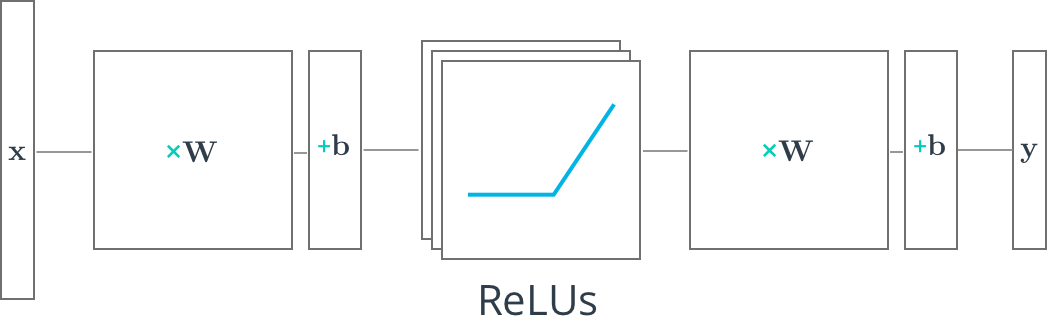
You should see the notebook intro\_to\_tensorflow.ipynb, this is the notebook you'll be working on. The notebook has 3 problems for you to solve:

* Problem 1: Normalize the features
* Problem 2: Use TensorFlow operations to create features, labels, weight, and biases tensors
* Problem 3: Tune the learning rate, number of steps, and batch size for the best accuracy

This is a self-assessed lab. Compare your answers to the solutions [**here**](https://github.com/udacity/deep-learning/blob/master/intro-to-tensorflow/intro_to_tensorflow_solution.ipynb). If you have any difficulty completing the lab, Udacity provides a few services to answer any questions you might have.

## Help

Remember that you can get assistance from your mentor, the Forums (click the link on the left side of the classroom), or the Slack channel. You can also review the concepts from the previous lessons.



**Multilayer Neural Networks**

In the previous lessons and the lab, you learned how to build a neural network of one layer. Now, you'll learn how to build multilayer neural networks with TensorFlow. Adding a hidden layer to a network allows it to model more complex functions. Also, using a non-linear activation function on the hidden layer lets it model non-linear functions.

The first thing we'll learn to implement in TensorFlow is ReLU hidden layer. A ReLU is a non-linear function, or rectified linear unit. The ReLU function is 0 for negative inputs and *x* for all inputs *x*>0.

As before, the following nodes will build up on the knowledge from the [**Deep Neural Networks**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/75cb91c8-003b-458c-8bac-653a17cd1a97) lesson. If you need to refresh your mind, you can go back and watch them again.

* [**ReLU**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/79a26389-34e8-4b3b-9cae-29f6ed792ba6)
* [**Feedforward**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/02c36864-ee71-481c-bb01-a34c35bfc581)
* [**Dropout**](https://classroom.udacity.com/nanodegrees/nd889/parts/16cf5df5-73f0-4afa-93a9-de5974257236/modules/6124bd95-dec2-44f9-bf3b-498ea57699c7/lessons/47f6c25c-7749-4a02-b807-7a5b37f362e8/concepts/700c78a4-20e1-436c-a0cd-43888a006420)

# TensorFlow ReLUs

TensorFlow provides the ReLU function as [**tf.nn.relu()**](https://www.tensorflow.org/api_docs/python/tf/nn/relu), as shown below.

*# Hidden Layer with ReLU activation function*

hidden\_layer = tf.add(tf.matmul(features, hidden\_weights), hidden\_biases)

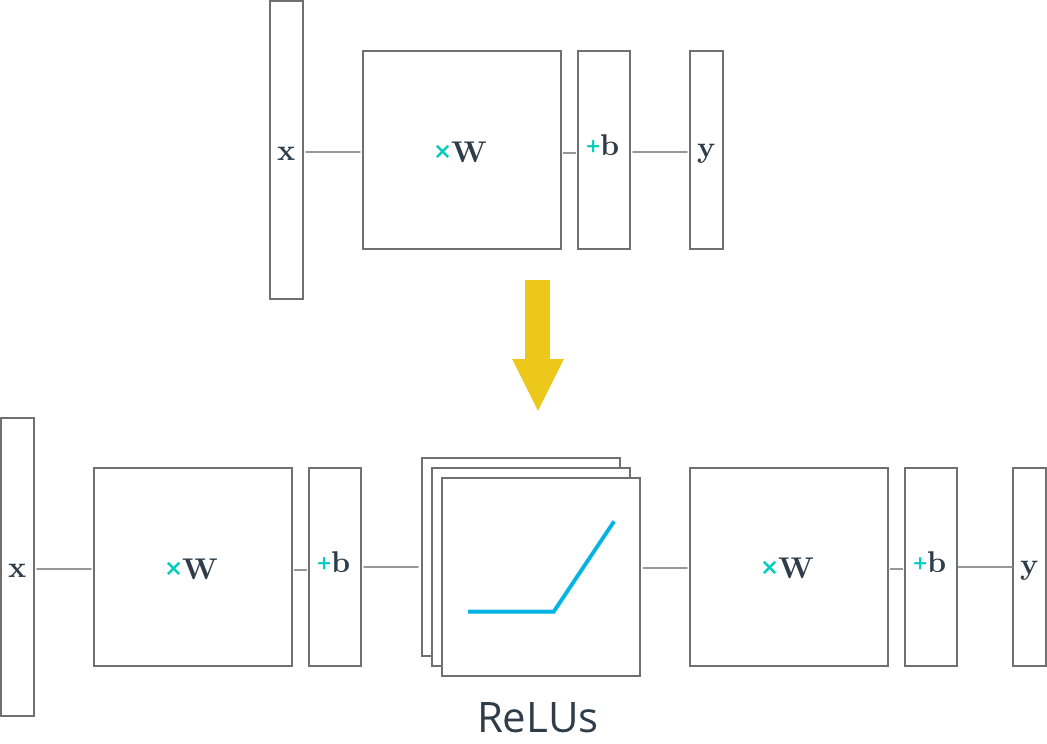
hidden\_layer = tf.nn.relu(hidden\_layer)

output = tf.add(tf.matmul(hidden\_layer, output\_weights), output\_biases)

The above code applies the [**tf.nn.relu()**](https://www.tensorflow.org/api_docs/python/tf/nn/relu) function to the hidden\_layer, effectively turning off any negative weights and acting like an on/off switch. Adding additional layers, like the output layer, after an activation function turns the model into a nonlinear function. This nonlinearity allows the network to solve more complex problems.

## Quiz

Below you'll use the ReLU function to turn a linear single layer network into a non-linear multilayer network.



TensorRELU.py

# -\*- coding: utf-8 -\*-

"""

Created on Wed Aug 30 12:45:00 2017

@author: shubra

"""

import tensorflow as tf

output = None

hidden\_layer\_weights = [[0.1,0.2,0.4],[0.4,0.6,0.6],[0.5,0.9,0.1],[0.8,0.2,0.8]]

output\_weights =[[0.1,0.6],[0.2,0.1],[0.7,0.9]]

weights = [tf.Variable(hidden\_layer\_weights), tf.Variable(output\_weights)]

biases = [tf.Variable(tf.zeros(3)),tf.Variable(tf.zeros(2))]

features = [[1.0,2.0,3.0,4.0],[-1.0,-2.0,-3.0,-4.0],[11.0,12.0,13.0,14.0]]

hidden\_layer = tf.add(tf.matmul(features,weights[0]),biases[0])

hidden\_layer = tf.nn.relu(hidden\_layer)

output = tf.add(tf.matmul(hidden\_layer,weights[1]),biases[1])

with tf.Session() as sess:

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

print(sess.run(output))

# Deep Neural Network in TensorFlow

You've seen how to build a logistic classifier using TensorFlow. Now you're going to see how to use the logistic classifier to build a deep neural network.

## Step by Step

In the following walkthrough, we'll step through TensorFlow code written to classify the letters in the MNIST database. If you would like to run the network on your computer, the file is provided [**here**](https://d17h27t6h515a5.cloudfront.net/topher/2017/February/58a61a3a_multilayer-perceptron/multilayer-perceptron.zip). You can find this and many more examples of TensorFlow at [**Aymeric Damien's GitHub repository**](https://github.com/aymericdamien/TensorFlow-Examples).

## Code

### TensorFlow MNIST

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

mnist = input\_data.read\_data\_sets(".", one\_hot=**True**, reshape=**False**)

You'll use the MNIST dataset provided by TensorFlow, which batches and One-Hot encodes the data for you.

### Learning Parameters

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

*# Parameters*

learning\_rate = 0.001

training\_epochs = 20

batch\_size = 128 *# Decrease batch size if you don't have enough memory*

display\_step = 1

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

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

The focus here is on the architecture of multilayer neural networks, not parameter tuning, so here we'll just give you the learning parameters.

### Hidden Layer Parameters

n\_hidden\_layer = 256 *# layer number of features*

The variable n\_hidden\_layer determines the size of the hidden layer in the neural network. This is also known as the width of a layer.

### Weights and Biases

*# Store layers weight & bias*

weights = {

'hidden\_layer': tf.Variable(tf.random\_normal([n\_input, n\_hidden\_layer])),

'out': tf.Variable(tf.random\_normal([n\_hidden\_layer, n\_classes]))

}

biases = {

'hidden\_layer': tf.Variable(tf.random\_normal([n\_hidden\_layer])),

'out': tf.Variable(tf.random\_normal([n\_classes]))

}

Deep neural networks use multiple layers with each layer requiring it's own weight and bias. The 'hidden\_layer' weight and bias is for the hidden layer. The 'out' weight and bias is for the output layer. If the neural network were deeper, there would be weights and biases for each additional layer.

### Input

*# tf Graph input*

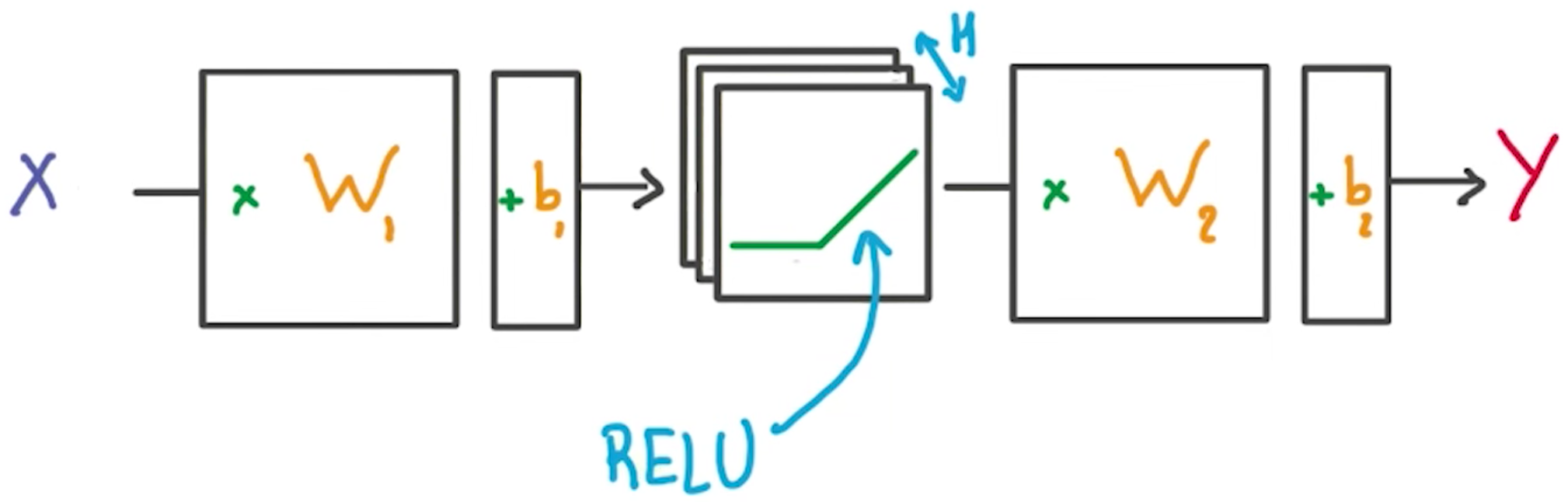
x = tf.placeholder("float", [**None**, 28, 28, 1])

y = tf.placeholder("float", [**None**, n\_classes])

x\_flat = tf.reshape(x, [-1, n\_input])

The MNIST data is made up of 28px by 28px images with a single [**channel**](https://en.wikipedia.org/wiki/Channel_(digital_image%29). The [**tf.reshape()**](https://www.tensorflow.org/versions/master/api_docs/python/tf/reshape) function above reshapes the 28px by 28px matrices in x into row vectors of 784px.

### Multilayer Perceptron



*# Hidden layer with RELU activation*

layer\_1 = tf.add(tf.matmul(x\_flat, weights['hidden\_layer']),\

biases['hidden\_layer'])

layer\_1 = tf.nn.relu(layer\_1)

*# Output layer with linear activation*

logits = tf.add(tf.matmul(layer\_1, weights['out']), biases['out'])

You've seen the linear function tf.add(tf.matmul(x\_flat, weights['hidden\_layer']), biases['hidden\_layer']) before, also known as xw + b. Combining linear functions together using a ReLU will give you a two layer network.

### Optimizer

*# Define loss and optimizer*

cost = tf.reduce\_mean(\

tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=y))

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

.minimize(cost)

This is the same optimization technique used in the Intro to TensorFLow lab.

### Session

*# Initializing the variables*

init = tf.global\_variables\_initializer()

*# Launch the graph*

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

sess.run(init)

*# Training cycle*

**for** epoch **in** range(training\_epochs):

total\_batch = int(mnist.train.num\_examples/batch\_size)

*# Loop over all batches*

**for** i **in** range(total\_batch):

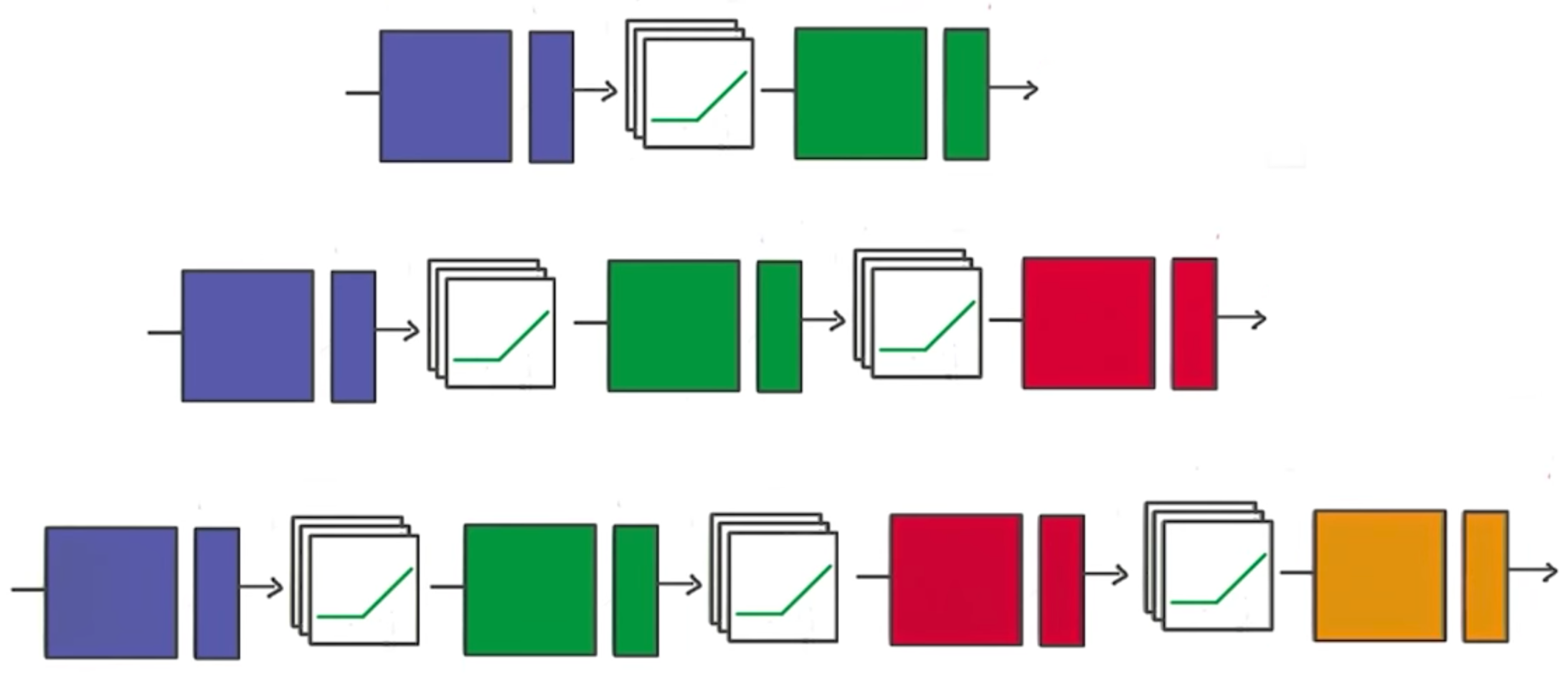
batch\_x, batch\_y = mnist.train.next\_batch(batch\_size)

*# Run optimization op (backprop) and cost op (to get loss value)*

sess.run(optimizer, feed\_dict={x: batch\_x, y: batch\_y})

The MNIST library in TensorFlow provides the ability to receive the dataset in batches. Calling the mnist.train.next\_batch() function returns a subset of the training data.

## Deeper Neural Network



That's it! Going from one layer to two is easy. Adding more layers to the network allows you to solve more complicated problems. In the next video, you'll see how changing the number of layers can affect your network.

**DeepNeuralNetworkTensor.py**

# -\*- coding: utf-8 -\*-

"""

Created on Wed Aug 30 13:41:21 2017

@author: shubra

"""

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

import tensorflow as tf

import numpy as np

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

mnist = input\_data.read\_data\_sets(".", one\_hot=True, reshape=False)

# Parameters

learning\_rate = 0.001

training\_epochs = 20

batch\_size = 128 # Decrease batch size if you don't have enough memory

display\_step = 1

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

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

"""

Layer number of features, it determnes the size of hidden layer in neural network

"""

n\_hidden\_layer = 256 # layer number of features

""" ADDED BY ME """

test\_features = mnist.test.images

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

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

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

# Store layers weight & bias

weights = {

'hidden\_layer': tf.Variable(tf.random\_normal([n\_input, n\_hidden\_layer])),

'out': tf.Variable(tf.random\_normal([n\_hidden\_layer, n\_classes]))

}

biases = {

'hidden\_layer': tf.Variable(tf.random\_normal([n\_hidden\_layer])),

'out': tf.Variable(tf.random\_normal([n\_classes]))

}

# tf Graph input

x = tf.placeholder("float", [None, 28, 28, 1])

y = tf.placeholder("float", [None, n\_classes])

"""

MNIST DB stores 28 by 28 pixel images with single chnnel. tf.reshape() the 28 by 28 matrices

in x into row vector of 784 pixcels

"""

x\_flat = tf.reshape(x, [-1, n\_input])

# Hidden layer with RELU activation

layer\_1 = tf.add(tf.matmul(x\_flat, weights['hidden\_layer']),\

biases['hidden\_layer'])

layer\_1 = tf.nn.relu(layer\_1)

# Output layer with linear activation

logits = tf.add(tf.matmul(layer\_1, weights['out']), biases['out'])

""" Calculate accuracy ADDED BY ME """

correct\_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

#print("accuracy ",accuracy)

"""

cost/loss and optimizer

"""

# Define loss and optimizer

cost = tf.reduce\_mean(\

tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=y))

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

.minimize(cost)

init = tf.global\_variables\_initializer()

print(tf.shape(features))

#print(test\_features[0])

#print(tf.reshape(test\_features,[784]))

#test\_features=tf.reshape(test\_features[0],[1,784])

# Launch the graph

with tf.Session() as sess:

sess.run(init)

# Training cycle

for epoch in range(training\_epochs):

total\_batch = int(mnist.train.num\_examples/batch\_size)

# Loop over all batches

for i in range(total\_batch):

batch\_x, batch\_y = mnist.train.next\_batch(batch\_size)

# Run optimization op (backprop) and cost op (to get loss value)

sess.run(optimizer, feed\_dict={x: batch\_x, y: batch\_y})

""" ADDED BY ME """

# test\_accuracy = sess.run(

# accuracy,

# feed\_dict={features: test\_features, labels: test\_labels})

#

#print('Test Accuracy: {}'.format(test\_accuracy))

#

# Save and Restore TensorFlow Models

Training a model can take hours. But once you close your TensorFlow session, you lose all the trained weights and biases. If you were to reuse the model in the future, you would have to train it all over again!

Fortunately, TensorFlow gives you the ability to save your progress using a class called [**tf.train.Saver**](https://www.tensorflow.org/api_docs/python/tf/train/Saver). This class provides the functionality to save any [**tf.Variable**](https://www.tensorflow.org/api_docs/python/tf/Variable) to your file system.

## Saving Variables

Let's start with a simple example of saving weights and bias Tensors. For the first example you'll just save two variables. Later examples will save all the weights in a practical model.

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

*# The file path to save the data*

save\_file = './model.ckpt'

*# Two Tensor Variables: weights and bias*

weights = tf.Variable(tf.truncated\_normal([2, 3]))

bias = tf.Variable(tf.truncated\_normal([3]))

*# Class used to save and/or restore Tensor Variables*

saver = tf.train.Saver()

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

*# Initialize all the Variables*

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

*# Show the values of weights and bias*

print('Weights:')

print(sess.run(weights))

print('Bias:')

print(sess.run(bias))

*# Save the model*

saver.save(sess, save\_file)

Weights:

[[-0.97990924 1.03016174 0.74119264]

[-0.82581609 -0.07361362 -0.86653847]]

Bias:

[ 1.62978125 -0.37812829 0.64723819]

The Tensors weights and bias are set to random values using the [**tf.truncated\_normal()**](https://www.tensorflow.org/api_docs/python/tf/truncated_normal) function. The values are then saved to the save\_file location, "model.ckpt", using the [**tf.train.Saver.save()**](https://www.tensorflow.org/api_docs/python/tf/train/Saver#save) function. (The ".ckpt" extension stands for "checkpoint".)

If you're using TensorFlow 0.11.0RC1 or newer, a file called "model.ckpt.meta" will also be created. This file contains the TensorFlow graph.

## Loading Variables

Now that the Tensor Variables are saved, let's load them back into a new model.

*# Remove the previous weights and bias*

tf.reset\_default\_graph()

*# Two Variables: weights and bias*

weights = tf.Variable(tf.truncated\_normal([2, 3]))

bias = tf.Variable(tf.truncated\_normal([3]))

*# Class used to save and/or restore Tensor Variables*

saver = tf.train.Saver()

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

*# Load the weights and bias*

saver.restore(sess, save\_file)

*# Show the values of weights and bias*

print('Weight:')

print(sess.run(weights))

print('Bias:')

print(sess.run(bias))

Weights:

[[-0.97990924 1.03016174 0.74119264]

[-0.82581609 -0.07361362 -0.86653847]]

Bias:

[ 1.62978125 -0.37812829 0.64723819]

You'll notice you still need to create the weights and bias Tensors in Python. The [**tf.train.Saver.restore()**](https://www.tensorflow.org/api_docs/python/tf/train/Saver#restore)function loads the saved data into weights and bias.

Since [**tf.train.Saver.restore()**](https://www.tensorflow.org/api_docs/python/tf/train/Saver#restore) sets all the TensorFlow Variables, you don't need to call [**tf.global\_variables\_initializer()**](https://www.tensorflow.org/api_docs/python/tf/global_variables_initializer).

## Save a Trained Model

Let's see how to train a model and save its weights.

First start with a model:

*# Remove previous Tensors and Operations*

tf.reset\_default\_graph()

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

**import** numpy **as** np

learning\_rate = 0.001

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('.', one\_hot=**True**)

*# Features and Labels*

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

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

*# Weights & bias*

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

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

*# Logits - xW + b*

logits = tf.add(tf.matmul(features, weights), bias)

*# Define loss and optimizer*

cost = tf.reduce\_mean(\

tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=labels))

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

.minimize(cost)

*# Calculate accuracy*

correct\_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

Let's train that model, then save the weights:

**import** math

save\_file = './train\_model.ckpt'

batch\_size = 128

n\_epochs = 100

saver = tf.train.Saver()

*# Launch the graph*

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

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

*# Training cycle*

**for** epoch **in** range(n\_epochs):

total\_batch = math.ceil(mnist.train.num\_examples / batch\_size)

*# Loop over all batches*

**for** i **in** range(total\_batch):

batch\_features, batch\_labels = mnist.train.next\_batch(batch\_size)

sess.run(

optimizer,

feed\_dict={features: batch\_features, labels: batch\_labels})

*# Print status for every 10 epochs*

**if** epoch % 10 == 0:

valid\_accuracy = sess.run(

accuracy,

feed\_dict={

features: mnist.validation.images,

labels: mnist.validation.labels})

print('Epoch {:<3} - Validation Accuracy: {}'.format(

epoch,

valid\_accuracy))

*# Save the model*

saver.save(sess, save\_file)

print('Trained Model Saved.')

Epoch 0 - Validation Accuracy: 0.06859999895095825

Epoch 10 - Validation Accuracy: 0.20239999890327454

Epoch 20 - Validation Accuracy: 0.36980000138282776

Epoch 30 - Validation Accuracy: 0.48820000886917114

Epoch 40 - Validation Accuracy: 0.5601999759674072

Epoch 50 - Validation Accuracy: 0.6097999811172485

Epoch 60 - Validation Accuracy: 0.6425999999046326

Epoch 70 - Validation Accuracy: 0.6733999848365784

Epoch 80 - Validation Accuracy: 0.6916000247001648

Epoch 90 - Validation Accuracy: 0.7113999724388123

Trained Model Saved.

## Load a Trained Model

Let's load the weights and bias from memory, then check the test accuracy.

saver = tf.train.Saver()

*# Launch the graph*

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

saver.restore(sess, save\_file)

test\_accuracy = sess.run(

accuracy,

feed\_dict={features: mnist.test.images, labels: mnist.test.labels})

print('Test Accuracy: {}'.format(test\_accuracy))

Test Accuracy: 0.7229999899864197

That's it! You now know how to save and load a trained model in TensorFlow. Let's look at loading weights and biases into modified models in the next section.

SaveTensor.py

# -\*- coding: utf-8 -\*-

"""

Created on Thu Aug 31 08:53:28 2017

@author: shubra

"""

import tensorflow as tf

# The file path to save the data

save\_file = './model.ckpt'

weights = tf.Variable(tf.truncated\_normal([2,3]))

biases = tf.Variable(tf.truncated\_normal([3]))

# Class used to save and/or restore Tensor Variables

saver = tf.train.Saver()

with tf.Session() as sess:

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

print('Weight:')

print(sess.run(weights))

print('Bias:')

print(sess.run(biases))

saver.save(sess,save\_file )

# Remove the previous weights and bias

tf.reset\_default\_graph()

# Two Variables: weights and bias

weights = tf.Variable(tf.truncated\_normal([2, 3]))

bias = tf.Variable(tf.truncated\_normal([3]))

# Class used to save and/or restore Tensor Variables

saver = tf.train.Saver()

with tf.Session() as sess:

# Load the weights and bias

saver.restore(sess, save\_file)

# Show the values of weights and bias

print('Weight:')

print(sess.run(weights))

print('Bias:')

print(sess.run(bias))

# Remove previous Tensors and Operations

tf.reset\_default\_graph()

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

import numpy as np

learning\_rate = 0.001

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('.', one\_hot=True)

# Features and Labels

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

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

# Weights & bias

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

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

# Logits - xW + b

logits = tf.add(tf.matmul(features, weights), bias)

# Define loss and optimizer

cost = tf.reduce\_mean(\

tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=labels))

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

.minimize(cost)

# Calculate accuracy

correct\_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

import math

save\_file = './train\_model.ckpt'

batch\_size = 128

n\_epochs = 100

saver = tf.train.Saver()

# Launch the graph

with tf.Session() as sess:

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

# Training cycle

for epoch in range(n\_epochs):

total\_batch = math.ceil(mnist.train.num\_examples / batch\_size)

# Loop over all batches

for i in range(total\_batch):

batch\_features, batch\_labels = mnist.train.next\_batch(batch\_size)

sess.run(

optimizer,

feed\_dict={features: batch\_features, labels: batch\_labels})

# Print status for every 10 epochs

if epoch % 10 == 0:

valid\_accuracy = sess.run(

accuracy,

feed\_dict={

features: mnist.validation.images,

labels: mnist.validation.labels})

print('Epoch {:<3} - Validation Accuracy: {}'.format(

epoch,

valid\_accuracy))

# Save the model

saver.save(sess, save\_file)

print('Trained Model Saved.')

**TensorLoadSaveTrainedModel.py**

# -\*- coding: utf-8 -\*-

"""

Created on Thu Aug 31 22:57:14 2017

@author: shubra

"""

import tensorflow as tf

save\_file = 'C:/Training/udacity/AI\_NanoDegree/Term2/5. Tensorflow/train\_model.ckpt'

saver = tf.train.Saver()

# Launch the graph

with tf.Session() as sess:

saver.restore(sess, save\_file)

test\_accuracy = sess.run(

accuracy,

feed\_dict={features: mnist.test.images, labels: mnist.test.labels})

print('Test Accuracy: {}'.format(test\_accuracy))

# Loading the Weights and Biases into a New Model

Sometimes you might want to adjust, or "finetune" a model that you have already trained and saved.

However, loading saved Variables directly into a modified model can generate errors. Let's go over how to avoid these problems.

## Naming Error

TensorFlow uses a string identifier for Tensors and Operations called name. If a name is not given, TensorFlow will create one automatically. TensorFlow will give the first node the name <Type>, and then give the name <Type>\_<number> for the subsequent nodes. Let's see how this can affect loading a model with a different order of weights and bias:

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

*# Remove the previous weights and bias*

tf.reset\_default\_graph()

save\_file = 'model.ckpt'

*# Two Tensor Variables: weights and bias*

weights = tf.Variable(tf.truncated\_normal([2, 3]))

bias = tf.Variable(tf.truncated\_normal([3]))

saver = tf.train.Saver()

*# Print the name of Weights and Bias*

print('Save Weights: {}'.format(weights.name))

print('Save Bias: {}'.format(bias.name))

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

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

saver.save(sess, save\_file)

*# Remove the previous weights and bias*

tf.reset\_default\_graph()

*# Two Variables: weights and bias*

bias = tf.Variable(tf.truncated\_normal([3]))

weights = tf.Variable(tf.truncated\_normal([2, 3]))

saver = tf.train.Saver()

*# Print the name of Weights and Bias*

print('Load Weights: {}'.format(weights.name))

print('Load Bias: {}'.format(bias.name))

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

*# Load the weights and bias - ERROR*

saver.restore(sess, save\_file)

The code above prints out the following:

Save Weights: Variable:0

Save Bias: Variable\_1:0

Load Weights: Variable\_1:0

Load Bias: Variable:0

...

InvalidArgumentError (see above for traceback): Assign requires shapes of both tensors to match.

...

You'll notice that the name properties for weights and bias are different than when you saved the model. This is why the code produces the "Assign requires shapes of both tensors to match" error. The code saver.restore(sess, save\_file) is trying to load weight data into bias and bias data into weights.

Instead of letting TensorFlow set the name property, let's set it manually:

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

tf.reset\_default\_graph()

save\_file = 'model.ckpt'

*# Two Tensor Variables: weights and bias*

weights = tf.Variable(tf.truncated\_normal([2, 3]), name='weights\_0')

bias = tf.Variable(tf.truncated\_normal([3]), name='bias\_0')

saver = tf.train.Saver()

*# Print the name of Weights and Bias*

print('Save Weights: {}'.format(weights.name))

print('Save Bias: {}'.format(bias.name))

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

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

saver.save(sess, save\_file)

*# Remove the previous weights and bias*

tf.reset\_default\_graph()

*# Two Variables: weights and bias*

bias = tf.Variable(tf.truncated\_normal([3]), name='bias\_0')

weights = tf.Variable(tf.truncated\_normal([2, 3]) ,name='weights\_0')

saver = tf.train.Saver()

*# Print the name of Weights and Bias*

print('Load Weights: {}'.format(weights.name))

print('Load Bias: {}'.format(bias.name))

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

*# Load the weights and bias - No Error*

saver.restore(sess, save\_file)

print('Loaded Weights and Bias successfully.')

Save Weights: weights\_0:0

Save Bias: bias\_0:0

Load Weights: weights\_0:0

Load Bias: bias\_0:0

Loaded Weights and Bias successfully.

That worked! The Tensor names match and the data loaded correctly.

**TensorSaveAndLoadWtBiasByName.py**

# -\*- coding: utf-8 -\*-

"""

Created on Thu Aug 31 23:11:07 2017

@author: shubra

"""

import tensorflow as tf

tf.reset\_default\_graph()

save\_file = './model\_variables/model\_name.ckpt'

# Two Tensor Variables: weights and bias

weights = tf.Variable(tf.truncated\_normal([2, 3]), name='weights\_0')

bias = tf.Variable(tf.truncated\_normal([3]), name='bias\_0')

saver = tf.train.Saver()

# Print the name of Weights and Bias

print('Save Weights: {}'.format(weights.name))

print('Save Bias: {}'.format(bias.name))

with tf.Session() as sess:

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

saver.save(sess, save\_file)

# Remove the previous weights and bias

tf.reset\_default\_graph()

# Two Variables: weights and bias

bias = tf.Variable(tf.truncated\_normal([3]), name='bias\_0')

weights = tf.Variable(tf.truncated\_normal([2, 3]) ,name='weights\_0')

saver = tf.train.Saver()

# Print the name of Weights and Bias

print('Load Weights: {}'.format(weights.name))

print('Load Bias: {}'.format(bias.name))

with tf.Session() as sess:

# Load the weights and bias - No Error

saver.restore(sess, save\_file)

print('Loaded Weights and Bias successfully.')

# TensorFlow Dropout

Figure 1: Taken from the paper "Dropout: A Simple Way to Prevent Neural Networks from
Overfitting" (https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf)

Figure 1: Taken from the paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" ([**https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf**](https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf))

Dropout is a regularization technique for reducing overfitting. The technique temporarily drops units ([**artificial neurons**](https://en.wikipedia.org/wiki/Artificial_neuron)) from the network, along with all of those units' incoming and outgoing connections. Figure 1 illustrates how dropout works.

TensorFlow provides the [**tf.nn.dropout()**](https://www.tensorflow.org/api_docs/python/tf/nn/dropout) function, which you can use to implement dropout.

Let's look at an example of how to use [**tf.nn.dropout()**](https://www.tensorflow.org/api_docs/python/tf/nn/dropout).

keep\_prob = tf.placeholder(tf.float32) *# probability to keep units*

hidden\_layer = tf.add(tf.matmul(features, weights[0]), biases[0])

hidden\_layer = tf.nn.relu(hidden\_layer)

hidden\_layer = tf.nn.dropout(hidden\_layer, keep\_prob)

logits = tf.add(tf.matmul(hidden\_layer, weights[1]), biases[1])

The code above illustrates how to apply dropout to a neural network.

The [**tf.nn.dropout()**](https://www.tensorflow.org/api_docs/python/tf/nn/dropout) function takes in two parameters:

1. hidden\_layer: the tensor to which you would like to apply dropout
2. keep\_prob: the probability of keeping (i.e. not dropping) any given unit

keep\_prob allows you to adjust the number of units to drop. In order to compensate for dropped units, [**tf.nn.dropout()**](https://www.tensorflow.org/api_docs/python/tf/nn/dropout) multiplies all units that are kept (i.e. not dropped) by 1/keep\_prob.

During training, a good starting value for keep\_prob is 0.5.

During testing, use a keep\_prob value of 1.0 to keep all units and maximize the power of the model.

## Quiz 1

Take a look at the code snippet below. Do you see what's wrong?

There's nothing wrong with the syntax, however the test accuracy is extremely low.

...

keep\_prob = tf.placeholder(tf.float32) *# probability to keep units*

hidden\_layer = tf.add(tf.matmul(features, weights[0]), biases[0])

hidden\_layer = tf.nn.relu(hidden\_layer)

hidden\_layer = tf.nn.dropout(hidden\_layer, keep\_prob)

logits = tf.add(tf.matmul(hidden\_layer, weights[1]), biases[1])

...

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

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

**for** epoch\_i **in** range(epochs):

**for** batch\_i **in** range(batches):

....

sess.run(optimizer, feed\_dict={

features: batch\_features,

labels: batch\_labels,

keep\_prob: 0.5})

validation\_accuracy = sess.run(accuracy, feed\_dict={

features: test\_features,

labels: test\_labels,

keep\_prob: 0.5})

### QUESTION 1 OF 2

What's wrong with the above code?

* 

Dropout doesn't work with batching.

* 

The keep\_prob value of 0.5 is too low.

* 

There shouldn't be a value passed to keep\_prob when testing for accuracy.

* **CORRECT ONE :** keep\_prob should be set to 1.0 when evaluating validation accuracy.

## Quiz 2

This quiz will be starting with the code from the ReLU Quiz and applying a dropout layer. Build a model with a ReLU layer and dropout layer using the keep\_prob placeholder to pass in a probability of 0.5. Print the logits from the model.

Note: Output will be different every time the code is run. This is caused by dropout randomizing the units it drops.

**TensorDropout.py**

# -\*- coding: utf-8 -\*-

"""

Created on Thu Aug 31 23:49:54 2017

@author: shubra

"""

# Solution is available in the other "solution.py" tab

import tensorflow as tf

hidden\_layer\_weights = [

[0.1, 0.2, 0.4],

[0.4, 0.6, 0.6],

[0.5, 0.9, 0.1],

[0.8, 0.2, 0.8]]

out\_weights = [

[0.1, 0.6],

[0.2, 0.1],

[0.7, 0.9]]

# Weights and biases

weights = [

tf.Variable(hidden\_layer\_weights),

tf.Variable(out\_weights)]

biases = [

tf.Variable(tf.zeros(3)),

tf.Variable(tf.zeros(2))]

# Input

features = tf.Variable([[0.0, 2.0, 3.0, 4.0], [0.1, 0.2, 0.3, 0.4], [11.0, 12.0, 13.0, 14.0]])

# TODO: Create Model with Dropout

keep\_prob = tf.placeholder(tf.float32) # probability to keep units

hidden\_layer = tf.add(tf.matmul(features, weights[0]), biases[0])

hidden\_layer = tf.nn.relu(hidden\_layer)

hidden\_layer = tf.nn.dropout(hidden\_layer, keep\_prob)

# TODO: Print logits from a session

logits = tf.add(tf.matmul(hidden\_layer, weights[1]), biases[1])

with tf.Session() as sess:

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

print(sess.run(logits, feed\_dict={keep\_prob: 0.5}))