A close up of a sign

Description automatically generated

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.

A close up of a logo

Description automatically generated

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**

**OS X, Linux, Windows**

**Prerequisites**

*Intro to TensorFlow* requires [Python 3.4 or higher](https://www.python.org/downloads/) and [Anaconda](https://www.anaconda.com/distribution/). If you don't meet all of these requirements, please install the appropriate package(s).

**Install TensorFlow**

You're going to use an Anaconda environment for this class. If you're unfamiliar with Anaconda environments, check out the [official documentation](https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html). More information, tips, and troubleshooting for installing tensorflow on Windows can be found [here](https://www.tensorflow.org/install/install_windows).

**Note:** If you've already created the environment for Term 1, you shouldn't need to do so again here!

Run the following commands to setup your environment:

conda create --name=IntroToTensorFlow python=3 anaconda

source activate IntroToTensorFlow

conda install -c conda-forge tensorflow

That's it! You have a working environment with TensorFlow. Test it out with the code in the *Hello, world!* section below.

**Docker on Windows**

Docker instructions were offered prior to the availability of a stable Windows installation via pip or Anaconda. Please try Anaconda first, Docker instructions have been retained as an alternative to an installation via Anaconda.

**Install Docker**

Download and install Docker from the [official Docker website](https://docs.docker.com/engine/installation/windows/).

**Run the Docker Container**

Run the command below to start a jupyter notebook server with TensorFlow:

docker run -it -p 8888:8888 gcr.io/tensorflow/tensorflow

*Users in China should use the b.gcr.io/tensorflow/tensorflow instead of gcr.io/tensorflow/tensorflow*

You can access the jupyter notebook at [localhost:8888](http://localhost:8888/). The server includes 3 examples of TensorFlow notebooks, but you can create a new notebook to test all your code.

**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)

**Errors**

If you're getting the error tensorflow.python.framework.errors.InvalidArgumentError: Placeholder:0 is both fed and fetched, you're running an older version of TensorFlow. Uninstall TensorFlow, and reinstall it using the instructions above. For more solutions, check out the [Common Problems](https://www.tensorflow.org/get_started/os_setup#common_problems) section.

**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](https://medium.com/tebs-lab/deep-neural-networks-as-computational-graphs-867fcaa56c9), a way of visualizing a mathematical process. Let’s take the TensorFlow code you ran and turn that into a graph:

A picture containing screenshot

Description automatically generated

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)

print(output)

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.

After you run the above, you will see the following printed out:

'Hello World!'

# 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/tf/math).)

**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.

A picture containing player, clock, ball

Description automatically generated

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_guides/python/constant_op).

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.

# TensorFlow Linear Function

Let’s derive the function y = Wx + b. We want to translate our input, x, to labels, y.

For example, imagine we want to classify images as digits.

x would be our list of pixel values, and y would be the logits, one for each digit. Let's take a look at y = Wx, where the weights, W, determine the influence of x at predicting each y.

A picture containing clock

Description automatically generated

y = Wx allows us to segment the data into their respective labels using a line.

However, this line has to pass through the origin, because whenever x equals 0, then y is also going to equal 0.

We want the ability to shift the line away from the origin to fit more complex data. The simplest solution is to add a number to the function, which we call “bias”.

A close up of a logo

Description automatically generated

Function y = Wx + b

Our new function becomes Wx + b, allowing us to create predictions on linearly separable data. Let’s use a concrete example and calculate the logits.

## Matrix Multiplication Quiz

Calculate the logits a and b for the following formula.

A picture containing clock

Description automatically generated

y = Wx + b

### Transposition

We've been using the y = Wx + b function for our linear function.

But there's another function that does the same thing, y = xW + b. These functions do the same thing and are interchangeable, except for the dimensions of the matrices involved.

To shift from one function to the other, you simply have to swap the row and column dimensions of each matrix. This is called transposition.

For rest of this lesson, we actually use xW + b, because this is what TensorFlow uses.

A picture containing clock

Description automatically generated

y = xW + b

The above example is identical to the quiz you just completed, except that the matrices are transposed.

x now has the dimensions 1x3, W now has the dimensions 3x2, and b now has the dimensions 1x2. Calculating this will produce a matrix with the dimension of 1x2.

You'll notice that the elements in this 1x2 matrix are the same as the elements in the 2x1 matrix from the quiz. Again, these matrices are simply transposed.

A picture containing drawing

Description automatically generated

We now have our logits! The columns represent the logits for our two labels.

Now you can learn how to train this function in TensorFlow.

## 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/programmers_guide/variables) 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/programmers_guide/variables) 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 picture containing clock

Description automatically generated

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.

A close up of a tiled wall

Description automatically generated

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/versions/r0.11/api_docs/python/state_ops/variables#Variable) of weights
   2. Implement get\_biases to return a [**tf.Variable**](https://www.tensorflow.org/versions/r0.11/api_docs/python/state_ops/variables#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).

You can’t train a neural network on a single sample. Let’s apply n samples of x to the function y = Wx + b, which becomes Y = WX + B.

A picture containing clock, meter

Description automatically generated

Y = WX + B

For every sample of X (X1, X2, X3), we get logits for label 1 (Y1) and label 2 (Y2).

In order to add the bias to the product of WX, we had to turn b into a matrix of the same shape. This is a bit unnecessary, since the bias is only two numbers. It should really be a vector.

We can take advantage of an operation called broadcasting used in TensorFlow and Numpy. This operation allows arrays of different dimension to be added or multiplied with each other. For example:

import numpy as np

t = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])

u = np.array([1, 2, 3])

print(t + u)

The code above will print...

[[ 2 4 6]

[ 5 7 9]

[ 8 10 12]

[11 13 15]]

This is because u is the same dimension as the last dimension in t.

A picture containing player, meter

Description automatically generated

Softmax Function

# Softmax

Congratulations on successfully implementing a linear function that outputs logits. You're one step closer to a working classifier.

The next step is to assign a probability to each label, which you can then use to classify the data. Use the softmax function to turn your logits into probabilities.

We can do this by using the formula above, which uses the input of y values and the mathematical constant "e" which is approximately equal to 2.718. By taking "e" to the power of any real value we always get back a positive value, this then helps us scale when having negative y values. The summation symbol on the bottom of the divisor indicates that we add together all the e^(input y value) elements in order to get our calculated probability outputs.

## Quiz

For the next quiz, you'll implement a softmax(x) function that takes in x, a one or two dimensional array of logits.

In the one dimensional case, the array is just a single set of logits. In the two dimensional case, each column in the array is a set of logits. The softmax(x) function should return a NumPy array of the same shape as x.

For example, given a one-dimensional array:

*# logits is a one-dimensional array with 3 elements*

logits = [1.0, 2.0, 3.0]

*# softmax will return a one-dimensional array with 3 elements*

**print** softmax(logits)

$ [ 0.09003057 0.24472847 0.66524096]

Given a two-dimensional array where each column represents a set of logits:

*# logits is a two-dimensional array*

logits = np.array([

[1, 2, 3, 6],

[2, 4, 5, 6],

[3, 8, 7, 6]])

*# softmax will return a two-dimensional array with the same shape*

**print** softmax(logits)

$ [

[ 0.09003057 0.00242826 0.01587624 0.33333333]

[ 0.24472847 0.01794253 0.11731043 0.33333333]

[ 0.66524096 0.97962921 0.86681333 0.33333333]

]

Implement the softmax function, which is specified by the formula at the top of the page.

The probabilities for each column must sum to 1. Feel free to test your function with the inputs above.

# TensorFlow Softmax

Now that you've built a softmax function from scratch, let's see how softmax is done in TensorFlow.

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.

## 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

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

*# 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.

## 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.

...

Epoch: 79 - Cost: 0.111 Valid Accuracy: 0.86

Epoch: 80 - Cost: 0.11 Valid Accuracy: 0.869

Epoch: 81 - Cost: 0.109 Valid Accuracy: 0.869

....

Epoch: 85 - Cost: 0.107 Valid Accuracy: 0.869

Epoch: 86 - Cost: 0.107 Valid Accuracy: 0.869

Epoch: 87 - Cost: 0.106 Valid Accuracy: 0.869

Epoch: 88 - Cost: 0.106 Valid Accuracy: 0.869

Epoch: 89 - Cost: 0.105 Valid Accuracy: 0.869

Epoch: 90 - Cost: 0.105 Valid Accuracy: 0.869

Epoch: 91 - Cost: 0.104 Valid Accuracy: 0.869

Epoch: 92 - Cost: 0.103 Valid Accuracy: 0.869

Epoch: 93 - Cost: 0.103 Valid Accuracy: 0.869

Epoch: 94 - Cost: 0.102 Valid Accuracy: 0.869

Epoch: 95 - Cost: 0.102 Valid Accuracy: 0.869

Epoch: 96 - Cost: 0.101 Valid Accuracy: 0.869

Epoch: 97 - Cost: 0.101 Valid Accuracy: 0.869

Epoch: 98 - Cost: 0.1 Valid Accuracy: 0.869

Epoch: 99 - Cost: 0.1 Valid Accuracy: 0.869

Test Accuracy: 0.8696000006198883

From looking at the output above, you can see the model doesn't increase the validation accuracy after epoch 80. Let's see what happens when we increase the learning rate.

learn\_rate = 0.1

Epoch: 76 - Cost: 0.214 Valid Accuracy: 0.752

Epoch: 77 - Cost: 0.21 Valid Accuracy: 0.756

Epoch: 78 - Cost: 0.21 Valid Accuracy: 0.756

...

Epoch: 85 - Cost: 0.207 Valid Accuracy: 0.756

Epoch: 86 - Cost: 0.209 Valid Accuracy: 0.756

Epoch: 87 - Cost: 0.205 Valid Accuracy: 0.756

Epoch: 88 - Cost: 0.208 Valid Accuracy: 0.756

Epoch: 89 - Cost: 0.205 Valid Accuracy: 0.756

Epoch: 90 - Cost: 0.202 Valid Accuracy: 0.756

Epoch: 91 - Cost: 0.207 Valid Accuracy: 0.756

Epoch: 92 - Cost: 0.204 Valid Accuracy: 0.756

Epoch: 93 - Cost: 0.206 Valid Accuracy: 0.756

Epoch: 94 - Cost: 0.202 Valid Accuracy: 0.756

Epoch: 95 - Cost: 0.2974 Valid Accuracy: 0.756

Epoch: 96 - Cost: 0.202 Valid Accuracy: 0.756

Epoch: 97 - Cost: 0.2996 Valid Accuracy: 0.756

Epoch: 98 - Cost: 0.203 Valid Accuracy: 0.756

Epoch: 99 - Cost: 0.2987 Valid Accuracy: 0.756

Test Accuracy: 0.7556000053882599

Looks like the learning rate was increased too much. The final accuracy was lower, and it stopped improving earlier. Let's stick with the previous learning rate, but change the number of epochs to 80.

Epoch: 65 - Cost: 0.122 Valid Accuracy: 0.868

Epoch: 66 - Cost: 0.121 Valid Accuracy: 0.868

Epoch: 67 - Cost: 0.12 Valid Accuracy: 0.868

Epoch: 68 - Cost: 0.119 Valid Accuracy: 0.868

Epoch: 69 - Cost: 0.118 Valid Accuracy: 0.868

Epoch: 70 - Cost: 0.118 Valid Accuracy: 0.868

Epoch: 71 - Cost: 0.117 Valid Accuracy: 0.868

Epoch: 72 - Cost: 0.116 Valid Accuracy: 0.868

Epoch: 73 - Cost: 0.115 Valid Accuracy: 0.868

Epoch: 74 - Cost: 0.115 Valid Accuracy: 0.868

Epoch: 75 - Cost: 0.114 Valid Accuracy: 0.868

Epoch: 76 - Cost: 0.113 Valid Accuracy: 0.868

Epoch: 77 - Cost: 0.113 Valid Accuracy: 0.868

Epoch: 78 - Cost: 0.112 Valid Accuracy: 0.868

Epoch: 79 - Cost: 0.111 Valid Accuracy: 0.868

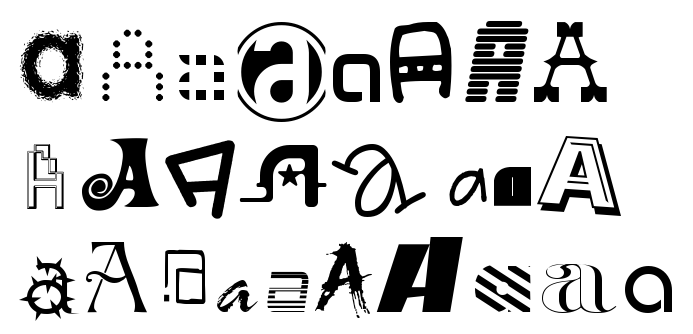
Epoch: 80 - Cost: 0.111 Valid Accuracy: 0.869

Test Accuracy: 0.86909999418258667

The accuracy only reached 0.86, but that could be because the learning rate was too high. Lowering the learning rate would require more epochs, but could ultimately achieve better accuracy.

In the upcoming TensorFLow Lab, you'll get the opportunity to choose your own learning rate, epoch count, and batch size to improve the model's accuracy.

# TensorFlow Neural Network Lab

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

We've prepared a Jupyter notebook that will guide you through the process of creating a single layer neural network in TensorFlow.

#### The Notebook

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 in the **solutions.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 [**Knowledge**](https://knowledge.udacity.com/) or the [**Slack channel**](https://carnd-slack.udacity.com/). You can also review the concepts from the previous lessons.