



Community Experience Distilled

TensorFlow Machine Learning Cookbook

Cook up the machine learning recipe of your choice using the latest numerical computing library—TensorFlow—with the help of this easy-to-follow cookbook

Nick McClure

[PACKT] open source*
PUBLISHING community experience distilled

Table of Contents

Chapter 1: Getting Started	1
Introduction	1
How Tensorflow Works	1
Declaring Tensors	4
Using Placeholders and Variables	6
Working with Matrices	9
Declaring Operations	11
Implmenting Activation Functions	14
Working with Data Sources	17
Additional Resources	22
Chapter 2: The Tensorflow Way	25
Introduction	25
Operations in a Computational Graph	26
Layering Nested Operations	27
Working With Multiple Layers	30
Implementing Loss Functions	33
Implementing Back Propagation	38
Working with Batch and Stochastic Training	43
Combining Everything Together	46
Evaluating Models	50
Chapter 3: Getting Started	57
Introduction	57
Using the Matrix Inverse Method	58
Implementing a Decomposition Method	60
The Tensorflow way of Linear Regression	62
Understanding Loss Functions in Linear Regression	66
Implementing Deming Regression	70

Implementing Lasso and Ridge Regression	73
Implementing Elastic Net Regression	75
Implementing Logistic Regression	79
Chapter 4: Support Vector Machines	85
Introduction	86
Working with a Linear SVM	87
Reduction to Linear Regression	93
Working with Kernels in Tensorflow	97
Implementing a Non-Linear SVM	104
Implementing a Multi-Class SVM	108
Chapter 5: Nearest Neighbor Methods	115
Introduction	115
Working with Nearest Neighbors	116
Working with Text Based Distances	121
Computing with Mixed Distance Functions	124
Using an Address Matching Example	128
Using Nearest Neighbors for Image Recognition	132
Chapter 6: Neural Networks	137
Introduction	137
Implementing Operational Gates	139
Working with Gates and Activation Functions	142
Implementing a One Layer Neural Network	146
Implementing Different Layers	150
Using a Multi-Layer Neural Network	157
Improving the Predictions of Linear Models	162
Learning to play Tic-Tac-Toe	168

1

Getting Started

For this chapter we will cover basic recipes for understanding how Tensorflow works and how to access data for this book and additional resources.

- ▶ How Tensorflow Works
- ▶ Declaring Variables
- ▶ Using Placeholders and Variables
- ▶ Working with Matrices
- ▶ Declaring Operations
- ▶ Implementing Activation Functions
- ▶ Working with Data Sources
- ▶ Additional Resources

Introduction

Google's Tensorflow engine has a unique way of solving problems. This unique way allows for solving machine learning problems very efficiently. We will cover the basic steps to understand how Tensorflow operates. This understanding is essential in understanding recipes for the rest of this book.

How Tensorflow Works

At first, computation in Tensorflow may seem needlessly complicated. But there is a reason for it: because of how Tensorflow treats computation, developing more complicated algorithms is relatively easy. This recipe will talk you through pseudocode of how a Tensorflow algorithm usually works.

Getting ready

Currently, Tensorflow is only supported on Mac and Linux distributions. To use Tensorflow on Windows requires the usage of a virtual machine, see 'Additional Resources' at the end of this chapter. Throughout this book we will only concern ourselves with the Python library wrapper of Tensorflow. This book will use Python 3.4+ (<https://www.python.org>) and Tensorflow 0.7 (<https://www.tensorflow.org>). While Tensorflow can run on the CPU, it runs faster if it runs on the GPU, and it is supported on graphics cards with NVidia Compute Capability 3.0+. To run on a GPU, you will also need to download and install the NVidia Cuda Toolkit (<https://developer.nvidia.com/cuda-downloads>). Some of the recipes will rely on a current installation of the Python packages Scipy, Numpy, and Scikit-Learn as well.

How to do it

Here we will introduce the general flow of Tensorflow Algorithms. Most recipes will follow this outline.

1. Import or generate data.
2. All of our machine learning algorithms will depend on data. In this book we will either generate data or use an outside source of data. Sometimes it is better to rely on generated data because we will want to know the expected outcome. Other times we will access public data sets for the given recipe and details on accessing these are in section 8 of this chapter.
3. Transform and normalize data.
4. The data is usually not in the correct dimension or type that our Tensorflow algorithms expect. We will have to transform our data before we can use it. Most algorithms also expect normalized data and we will do this here as well. Tensorflow has built in functions that can normalize the data for you.

```
data = tf.nn.batch_norm_with_global_normalization(...)
```

5. Set algorithm parameters.
6. Our algorithms usually have a set of parameters that we hold constant throughout the procedure. For example, this can be the number of iterations, the learning rate, or other fixed parameters of our choosing. It is considered good form to initialize these together so the reader or user can easily find them.

```
learning_rate = 0.01
iterations = 1000
```

7. Initialize variables and placeholders.

8. Tensorflow depends on us telling it what it can and cannot modify. Tensorflow will modify the variables during optimization to minimize a loss function. To accomplish this, we feed in data through placeholders. We need to initialize both of these, variables and placeholders with size and type, so that Tensorflow knows what to expect.

```
a_var = tf.constant(42)
x_input = tf.placeholder(tf.float32, [None, input_size])
y_input = tf.placeholder(tf.float32, [None, num_classes])
```

9. Define the model structure.
10. After we have the data, and initialized our variables and placeholders, we have to define the model. This is done by building a computational graph. We tell Tensorflow what operations must be done on the variables and placeholders to arrive at our model predictions. We talk more in depth about computational graphs in chapter two, section one of this book.

```
y_pred = tf.add(tf.mul(x_input, weight_matrix), b_matrix)
```

11. Declare the loss functions.
12. After defining the model, we must be able to evaluate the output. This is where we declare the loss function. The loss function is very important as it tells us how far off our predictions are from the actual values. The different types of loss functions are explored in greater detail in chapter two, section five.
13. `loss = tf.reduce_mean(tf.square(y_actual - y_pred))`
14. Initialize and train the model.
15. Now that we have everything in place, we create an instance of our graph and feed in the data through the placeholders and let Tensorflow change the variables to better predict our training data. Here is one way to initialize the computational graph.

```
with tf.Session(graph=graph) as session:
    ...
    session.run(...)
    ...
```

16. Note that we can also initiate our graph with


```
session = tf.Session(graph=graph)
session.run(...)
```
17. (Optional) Evaluate the model.
18. Once we have built and trained the model, we should evaluate the model by looking at how well it does on new data through some specified criteria.
19. (Optional) Predict new outcomes.
20. It is also important to know how to make predictions on new, unseen, data. We can do this with all of our models, once we have them trained.

How it works

In Tensorflow, we have to setup the data, variables, placeholders, and model before we tell the program to train and change the variables to improve the predictions. Tensorflow accomplishes this through the computational graph. We tell it to minimize a loss function and Tensorflow does this by modifying the variables in the model. Tensorflow knows how to modify the variables because it keeps track of the computations in the model and automatically computes the gradients for every variable. Because of this, we can see how easy it can be to make changes and try different data sources.

See also

A great place to start is the official python api Tensorflow documentation (https://www.tensorflow.org/versions/r0.7/api_docs/python/index.html). There are also tutorials available (<https://www.tensorflow.org/versions/r0.7/tutorials/index.html>).

Declaring Tensors

Getting ready

Tensors are the data structure that Tensorflow operates on in the computational graph. We can declare these tensors as variables or feed them in as placeholders. First we must know how to create tensors. When we create a tensor and declare it to be a variable, Tensorflow creates several graph structures in our computation graph. It is also important to point out that just by creating a tensor, Tensorflow is not adding anything to the computational graph. Tensorflow does this only after creating a Variable out of the tensor. See the next section on variables and placeholders for more information.

How to do it

Here we will cover the main ways to create tensors in Tensorflow.

Fixed tensors:

Creating a zero filled tensor:

```
zero_tsr = tf.zeros([row_dim, col_dim])
```

Creating a one filled tensor:

```
ones_tsr = tf.ones([row_dim, col_dim])
```

Creating a constant filled tensor:

```
filled_tsr = tf.fill([row_dim, col_dim], 42)
```

Creating a tensor out of an existing constant:

```
constant_tsr = tf.constant([1,2,3])
```



Note that the `tf.constant()` function can be used to broadcast a value into an array, mimicking the behaviour of `tf.fill()` by writing `tf.constant(42, [row_dim, col_dim])`

Tensors of similar shape

We can also initialize variables based on the shape of other tensors:

```
zeros_similar = tf.zeros_like(constant_tsr)
ones_similar = tf.ones_like(constant_tsr)
```

Note, that since these tensors depend on prior tensors, we must initialize them in order. Attempting to initialize all the tensors all at once will result in an error. See the section 'There's More' at the end of the next chapter on variables and placeholders.

Sequence tensors:

Tensorflow allows us to specify tensors that contain defined intervals. The following functions behave very similarly to the `range()` outputs and numpy's `linspace()` outputs.

```
linear_tsr = tf.linspace(start=0, stop=1, num=3)
```

The resulting tensor is the sequence `[0.0, 0.5, 1.0]`. Note that this function includes the specified stop value.

```
integer_seq_tsr = tf.range(start=6, limit=15, delta=3)
The result is the sequence [6, 9, 12]. Note that this function does
not include the limit value.
```

Random tensors:

The following generated random numbers from a uniform distribution:

```
randunif_tsr = tf.random_uniform([row_dim, col_dim], minval=0,
maxval=1)
```

Know that this random uniform distribution draws from the interval that includes the `minval` but not the `maxval` (`minval ≤ x < maxval`).

To get a tensor with random draws from a normal distribution:

```
randnorm_tsr = tf.random_normal([row_dim, col_dim], mean=0.0,
stddev=1.0)
```


Getting Started

There are also times when we wish to generate normal random values that are assured within certain bounds. The `truncated_normal()` function always picks normal values within two standard deviations of the specified mean.

```
truncnorm_tsr = tf.truncated_normal([row_dim, col_dim], mean=0.0,
stddev=1.0)
```

We might also be interested in randomizing entries of arrays. To accomplish this there are two functions that help us, `random_shuffle()` and `random_crop()`.

```
shuffled_output = tf.random_shuffle(input_tensor)
cropped_output = tf.random_crop(input_tensor, crop_size)
```

Later on in this book, we will be interested in randomly cropping an image of size (height, width, 3) where there are 3 color spectrums. To fix a dimension in the `cropped_output`, you must give it the maximum size in that dimension:

```
cropped_image = tf.random_crop(my_image, [height/2, width/2, 3])
```

How it works

Once we have decided on how to create the tensors, then we may also create the corresponding variables by wrapping the tensor in the `Variable()` function. More on this in the next section.

```
my_var = tf.Variable(tf.zeros([row_dim, col_dim]))
```

There's more

We are not limited to the built in functions, we can convert any numpy array, python list, or constant to a tensor using the function `convert_to_tensor()`. Know that this function also accepts tensors as an input in case we wish to generalize a computation inside a function.

Using Placeholders and Variables

Getting ready

One of the most important distinctions to make with data is whether it is a placeholder or variable. Variables are parameters of the algorithm that Tensorflow keeps track of how to change these to optimize the algorithm. Placeholders are objects that allow you to feed in data of a specific type and shape or that depend on the results of the computational graph, like the expected outcome of a computation.

```
my_var = tf.Variable(tf.zeros([2,3]))
sess = tf.Session()
initialize_op = tf.initialize_all_variables()
sess.run(initialize_op)
```

Placeholders are just holding the position for data to be fed into the graph. Placeholders get data from a `feed_dict` argument in the session. To put a placeholder in the graph, we must perform at least one operation on the placeholder. We initialize the graph, declare `x` to be a placeholder, and define `y` as the identity operation on `x`, which just returns `x`. We then create data to feed into the `x` placeholder and run the identity operation. It is worth noting that Tensorflow will not return a self-referenced placeholder in the feed dictionary. The code is shown below and the resulting graph is in the next section, "How it works"..

```
sess = tf.Session()
x = tf.placeholder(tf.float32, shape=[2,2])
y = tf.identity(x)
x_vals = np.random.rand(2,2)
sess.run(y, feed_dict={x: x_vals})
# Note that sess.run(x, feed_dict={x: x_vals}) will result in a self-
referencing error.
```

How it works

Diagram illustrating variable access patterns:

- Variable** (Central Node):
 - Accessed by **Assign** and **read**.
 - Accessed by **zeros** and **init**.
 - Accessed by **read**.
- init** (Node):
 - Accessed by **Variable**.

7

Figure 1: Here we can see what the computational graph looks like in detail with just one variable, initialized to all zeros. The grey shaded region is a very detailed view of the operations and constants involved. The main computational graph with less detail is the smaller graph outside of the grey region in the upper right. For more details on creating and visualizing graphs, see chapter X, section X.

Similarly, the computational graph of feeding a numpy array into a placeholder can be seen below, in Figure 2.

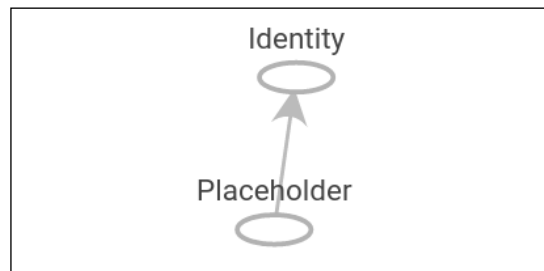


Figure 2: Here is the computational graph of a placeholder initialized. The grey shaded region is a very detailed view of the operations and constants involved. The main computational graph with less detail is the smaller graph outside of the grey region in the upper right.

There's more

During the run of the computational graph, we have to tell Tensorflow when to initialize the variables we have created. While each variable has an initializer method, the most common way to do this is with the helper function `initialize_all_variables()`. This function creates an operation in the graph that initializes all the variables we have created.

```
initializer_op = tf.initialize_all_variables()
```

But if we want to initialize a variable based on the results of initializing another variable. To do this, we have to initialize variables in the order we want.

```
sess = tf.Session()
first_var = tf.Variable(tf.zeros([2,3]))
sess.run(first_var.initializer)
second_var = tf.Variable(tf.zeros_like(first_var))
# Depends on first_var
sess.run(second_var.initializer)
```

Working with Matrices

Getting ready

Many algorithms depend on matrix operations. Tensorflow gives us easy to use operations to perform such matrix calculations. For all of the following examples, we can create a graph session by running the following code.

```
import tensorflow as tf
```

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

How to do it

Creating matrices

We can create two-dimensional matrices from numpy arrays or nested lists, like we described in the earlier section on tensors. We can also use the tensor creation functions and specify a two-dimensional shape for functions like `zeros()`, `ones()`, `truncated_normal()`, etc...

Tensorflow also allows us to create a diagonal matrix from a one dimensional array or list with the function `diag()`.

```
identity_matrix = tf.diag([1.0, 1.0, 1.0]) # Identity matrix
A = tf.truncated_normal([2, 3]) # 2x3 random normal matrix
B = tf.fill([2,3], 5.0) # 2x3 constant matrix of 5's
C = tf.random_uniform([3,2]) # 3x2 random uniform matrix
D = tf.convert_to_tensor(np.array([[1., 2., 3.], [-3., -7., -1.], [0.,
5., -2.])))
print(sess.run(identity_matrix))
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
print(sess.run(A))
[[ 0.96751703  0.11397751 -0.3438891 ]
 [-0.10132604 -0.8432678  0.29810596]]
print(sess.run(B))
[[ 5.  5.  5.]
 [ 5.  5.  5.]]
print(sess.run(C))
[[ 0.33184157  0.08907614]
 [ 0.53189191  0.67605299]
 [ 0.95889051  0.67061249]]
print(sess.run(D))
[[ 1.  2.  3.]
 [-3. -7. -1.]
 [ 0.  5. -2.]]
```

Note that if we were to run `sess.run(C)` again, we would reinitialize the random variables and end up with different random values.

Addition and subtraction

```
print(sess.run(A+B))
[[ 4.61596632  5.39771316  4.43256895 ]
 [ 3.26702736  5.14477345  4.98265553]]
print(sess.run(B-B))
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Multiplication

```
print(sess.run(tf.matmul(B, identity_matrix)))
[[ 5.  5.  5.]
 [ 5.  5.  5.]]
```

Also, the function `matmul()` has arguments that specify whether or not to transpose the arguments before multiplication or whether each matrix is sparse.

Transpose

```
print(sess.run(tf.transpose(C)))
[[ 0.67124544  0.26766731  0.99068872]
 [ 0.25006068  0.86560275  0.58411312]]
```

Again, it is worth mentioning the reinitializing that gives us different values than before.

Determinant

```
print(sess.run(tf.matrix_determinant(D)))
-38.0
Inverse
print(sess.run(tf.matrix_inverse(D)))
[[-0.5         -0.5         -0.5         ]
 [ 0.15789474  0.05263158  0.21052632]
 [ 0.39473684  0.13157895  0.02631579]]
```

Note that the inverse method is based off the Cholesky decomposition if the matrix is symmetric positive definite or the LU decomposition otherwise.

Decompositions

Cholesky decomposition:

```
print(sess.run(tf.cholesky(identity_matrix)))
[[ 1.  0.  1.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Eigenvalues and Eigenvectors

```
print(sess.run(tf.self_adjoint_eig(D)))  
[[-10.65907521  -0.22750691   2.88658212]  
 [  0.21749542   0.63250104  -0.74339638]  
 [  0.84526515   0.2587998   0.46749277]  
 [ -0.4880805    0.73004459   0.47834331]]
```

Note that the function `self_adjoint_eig()` outputs the eigenvalues in the first row and the subsequent vectors in the remaining vectors. In mathematics, this is called the eigendecomposition of a matrix.

How it works

Tensorflow provides all the tools for us to get started with numerical computations and adding such computations to our graphs. This notation might seem quite heavy for simple matrix operations. Remember that we are adding these operations to the graph and telling Tensorflow what tensors to run through those operations. While this might seem verbose now, it helps to understand the notation in later chapters when this way of computation will make it easier to accomplish our goals.

Declaring Operations

Getting ready

Besides the standard arithmetic operations, Tensorflow provides us more operations that we should be aware of and how to use them before proceeding. Again, we can create a graph session by running the following code.

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

How to do it

Tensorflow has the standard operations on tensors, `add()`, `sub()`, `mul()`, and `div()`. Note that all of these operations in this section will evaluate the inputs element-wise unless specified otherwise.

Tensorflow provides some variations of `div()` and relevant functions.

It is worth mentioning that `div()` returns the same type as the inputs. This means it really returns the floor of the division (akin to Python 2) if the inputs are integers. To return the Python 3 version, which casts integers into floats before dividing and always returns a float, Tensorflow provides the function `truediv()`.

```
print(sess.run(tf.div(3,4)))
0
print(sess.run(tf.truediv(3,4)))
0.75
```

If we have floats and want integer division, we can use the function `floordiv()`. Note that this will still return a float, but rounded down to the nearest integer.

```
print(sess.run(tf.floordiv(3.0,4.0)))
0.0
```

Another important function is `mod()`. This function returns the remainder after division.

```
print(sess.run(tf.mod(22.0, 5.0)))
2.0
```

The cross product between two tensors is done by the `cross()` function. Remember that the cross product is only defined for two 3-dimensional vectors, so it only accepts two 3-dimensional tensors.

```
print(sess.run(tf.cross([1., 0., 0.], [0., 1., 0.])))
[ 0.  0.  1.0]
```

Here is a compact list of the more common math functions. All of these functions operate elementwise.

<code>abs()</code>	Absolute value of one input tensor
<code>ceil()</code>	Ceiling function of one input tensor
<code>cos()</code>	Cosine function of one input tensor
<code>exp()</code>	Base e exponential of one input tensor
<code>floor()</code>	Floor function of one input tensor
<code>inv()</code>	Multiplicative inverse ($1/x$) of one input tensor
<code>log()</code>	Natural logarithm of one input tensor
<code>maximum()</code>	Element-wise max of two tensors
<code>minimum()</code>	Element-wise min of two tensors
<code>neg()</code>	Negative of one input tensor
<code>pow()</code>	The first tensor raised to the second tensor element-wise
<code>round()</code>	Rounds one input tensor
<code>rsqrt()</code>	One over the square root of one tensor

<code>sign()</code>	Returns -1, 0, or 1, depending on the sign of the tensor
<code>sin()</code>	Sine function of one input tensor
<code>sqrt()</code>	Square root of one input tensor
<code>square()</code>	Square of one input tensor

Specialty mathematical functions

There are some special math functions that get used in machine learning that are worth mentioning and Tensorflow has built in functions for them. Again, these functions operate element-wise, unless specified otherwise

<code>digamma()</code>	Psi function, the derivative of the <code>lgamma()</code> function
<code>erf()</code>	Gaussian error function, element-wise, of one tensor
<code>erfc()</code>	Complimentary error function of one tensor
<code>igamma()</code>	Lower regularized incomplete gamma function
<code>igammac()</code>	Upper regularized incomplete gamma function
<code>lbeta()</code>	Natural logarithm of the absolute value of the beta function
<code>lgamma()</code>	Natural logarithm of the absolute value of the gamma function
<code>squared_difference()</code>	Computes the square of the differences between two tensors

How it works

It is important to know what functions are available to us to add to our computational graphs. Mostly we will be concerned with the above functions. We can also generate many different custom functions as compositions of the above.

```
# Tangent function (tan(pi/4)=1)
print(sess.run(tf.div(tf.sin(3.1416/4.), tf.cos(3.1416/4.))))
1.0
```

There's more

If we wish to add other operations to our graphs that are not listed here, we must create our own from the above. Here is an example of an operation not listed above that we can add to our graph.

```
# Define a custom polynomial function
def custom_polynomial(value):

    # Return 3 * x^2 - x + 10
```



```
        return(tf.sub(3 * tf.square(value), value) + 10)
    print(sess.run(custom_polynomial(11)))
362
```

Implementing Activation Functions

Getting ready

When we get ready to use neural networks, we will use activation functions regularly. In Tensorflow, activation functions are non-linear operations that act on tensors. They function very similar as the prior mathematical operations. Typically they reduce the output as compared to the input with having horizontal asymptotes, resulting in a squashing effect. Start a Tensorflow graph with the following commands.

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

How to do it

The activation functions live in the nn (neural network) library in Tensorflow. We can either import this by itself (import tensorflow.nn as nn) or be explicit and write .nn in our function calls. Here we choose to be explicit with each function call.

The rectified linear unit, known as ReLU. This is the most common and basic way to introduce a non-linearity into neural networks. This function is just $\max(0, x)$. It is continuous but not smooth.

```
print(sess.run(tf.nn.relu([-3., 3., 10.])))
[ 0.  3. 10.]
```

There are times when we wish to cap the linearly increasing part of the above ReLU activation function. We can do this by nesting the $\max(0, x)$ function into a $\min()$ function. The implementation that Tensorflow has is called the ReLU6 function. This is defined as $\min(\max(0, x), 6)$. This is a version of the hard-sigmoid function and is computationally faster, and does not suffer from vanishing (infinitesimally near zero) or exploding values. This will come in handy when we discuss deeper neural networks in chapters 8 and 9.

```
print(sess.run(tf.nn.relu6([-3., 3., 10.])))
[ 0.  3.  6.]
```

The Sigmoid function is the most common continuous and smooth activation functions. It is also called a logistic function and has the form $1/(1+\exp(-x))$.

```
print(sess.run(tf.nn.sigmoid([-1., 0., 1.])))
[ 0.26894143  0.5          0.7310586 ]
```

Another smooth activation function is the hyper tangent. The hyper tangent function very similar to the sigmoid except that instead of having a range between 0 and 1, it has a range between -1 and 1. The function has the form of the ratio of the hyperbolic sine over the hyperbolic cosine. But another way to write this is $((\exp(x)-\exp(-x))/(\exp(x)+\exp(-x)))$.

```
print(sess.run(tf.nn.tanh([-1., 0., 1.])))
[-0.76159418  0.          0.76159418 ]
```

The softsign function is also gets used as an activation function. The form of this function is $x/(\text{abs}(x) + 1)$.

```
print(sess.run(tf.nn.softsign([-1., 0., -1.])))
[-0.5  0.   0.5]
```

Another function, the softplus is a smooth version of the ReLU function. The form of this function is $\log(\exp(x) + 1)$.

```
print(sess.run(tf.nn.softplus([-1., 0., -1.])))
[ 0.31326166  0.69314718  1.31326163]
```

The exponential linear unit (ELU) is very similar to the softplus function except that the bottom asymptote is -1 instead of 0. The form is $(\exp(x)+1)$ if $x < 0$ else x .

```
print(sess.run(tf.nn.elu([-1., 0., -1.])))
[-0.63212055  0.          1.          ]
```

How it works

These activation functions are the way that we introduce nonlinearities in neural networks or other computational graphs in the future. It is important to note where in our network we are using activation functions. If the activation function has a range between 0 and 1 (sigmoid) then the computational graph can only output values between 0 and 1.

If the activation functions are inside and hidden between nodes, then we want to be aware of the effect that the range can have on our tensors as we pass them through. If our tensors were scaled to have a mean of zero, we will want to use an activation function that preserves as much variance as possible around zero. This would imply we want to choose an activation function like the hyperbolic tangent or softsign. If the tensors are all scaled to be positive, then we would ideally choose an activation function that preserves variance in the positive domain.

There's more

Here are two graphs that illustrate the different activation functions. Figure 3 are the functions ReLU, ReLU6, softplus, and the exponential LU. Figure 4 has the sigmoid, softsign, and the hyperbolic tangent.

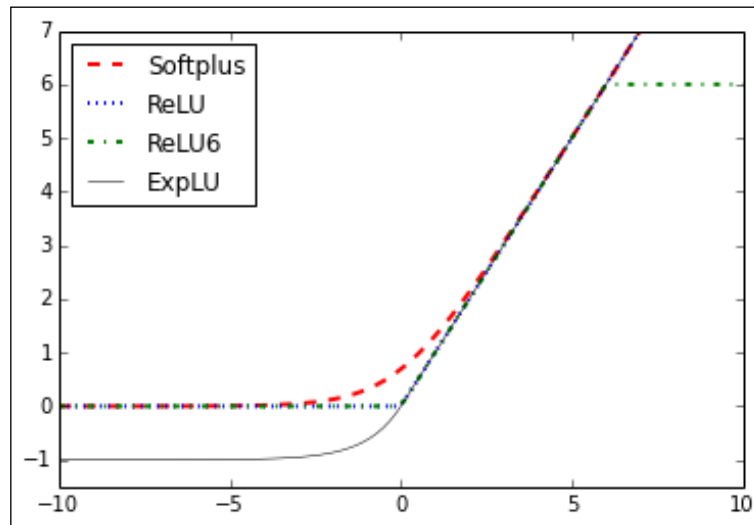


Figure 3: Here we can see four of the activation functions, softplus, ReLU, ReLU6, and Exponential LU. These functions flatten out to the left of zero and linearly increase to the right of zero, with the exception of ReLU6, which has a maximum value of 6.

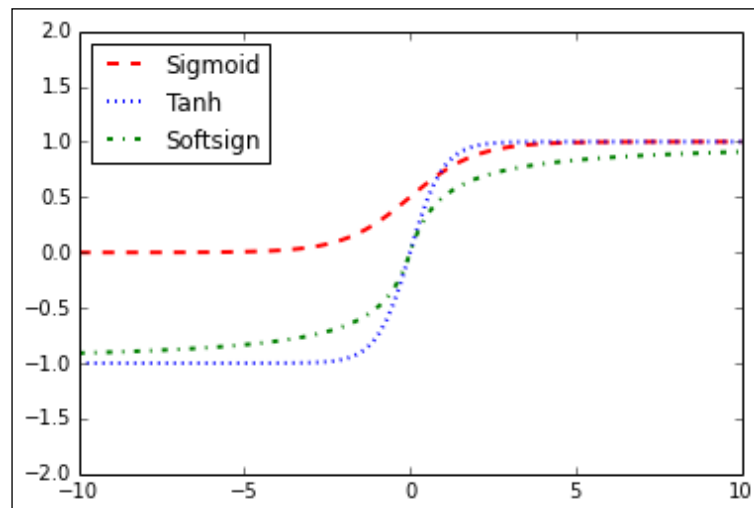


Figure 4: Here are the activation functions sigmoid, hyperbolic tangent (tanh), and softsign. These activation functions are all smooth and have a 'S' shape. Note that there are two horizontal asymptotes for these functions.

Working with Data Sources

Getting ready

Throughout this book, we will rely on access to various data sources to test our algorithms on. This section has instructions on how to access each of those various data sets through Tensorflow and Python. Some data sets are built in to Python libraries, some will require a Python script to download, and some will be manually downloaded through the internet. Almost all of these data sets will require an active internet connection to retrieve.

How to do it

Iris data. This dataset is arguably the most classic dataset used in machine learning and maybe all of statistics. It is a dataset that measures sepal length, sepal width, petal length, and petal width of three different types of iris flowers, *Iris setosa*, *Iris virginica*, and *Iris versicolor*. There are 150 measurements overall, 50 measurements of each species. To load the data set in python, we use Scikit Learn's dataset function.

```
from sklearn import datasets
iris = datasets.load_iris()
print(len(iris.data))
150
print(len(iris.target))
150
print(iris.target[0]) # Sepal length, Sepal width, Petal length, Petal
width
[ 5.1  3.5  1.4  0.2]
print(set(iris.target)) # I. setosa, I. virginica, I. versicolor
{0, 1, 2}
```

Birth weight data. The University of Massachusetts at Amherst has compiled many statistical datasets that are of interest (1). One such dataset is a measure of child birth weight and other demographic and medical measurements of the mother and family history. There are 189 observations of 11 variables. Here is how to access the data in Python.

```
import requests
birthdata_url = 'https://www.umass.edu/statdata/statdata/data/lowbwt.
dat'
birth_file = requests.get(birthdata_url)
birth_data = birth_file.text.split('\r\n') [5:]
```

```
birth_header = [x for x in birth_data[0].split(' ') if len(x)>=1]
birth_data = [[float(x) for x in y.split(' ') if len(x)>=1] for y in
birth_data[1:] if len(y)>=1]
print(len(birth_data))
189
print(len(birth_data[0]))
11
```

Boston Housing data. Carnegie Mellon University maintains a library of datasets in their Statlib Library. This data is easily accessible via The University of California at Irvine's Machine Learning Repository (2). There are 506 observations of house worth along with various demographic data and housing attributes (14 variables).

```
import requests
housing_url = 'https://archive.ics.uci.edu/ml/machine-learning-
databases/housing/housing.data'
```

```
housing_header = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
'B', 'LSTAT', 'MEDV0']
```

```
housing_file = requests.get(housing_url)
housing_data = [[float(x) for x in y.split(' ') if len(x)>=1] for y in
housing_file.text.split('\n') if len(y)>=1]
print(len(housing_data))
```

506

```
print(len(housing_data[0]))
```

14

MNIST handwriting data. MNIST (Mixed National Institute of Standards and Technology) is a subset of the larger NIST handwriting database. The MNIST handwriting dataset is hosted on Yann LeCun's website (<https://yann.lecun.com/exdb/mnist/>). It is a database of 70,000 images of single digit numbers (0-9) with about 60,000 annotated for a training set and 10,000 for a test set. This dataset is used so often in image recognition that Tensorflow provides built in functions to access this data. In machine learning it is also important to provide validation data to prevent target leakage. Because of this Tensorflow sets aside 5,000 of the train set into a validation set.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
print(len(mnist.train.images))
55000
print(len(mnist.test.images))
10000
print(len(mnist.validation.images))
5000
print(mnist.train.labels[1,:]) # The first label is a '3'
[ 0.  0.  0.  1.  0.  0.  0.  0.  0.]
```

Spam-ham text data. UCI's machine learning data set library (2) also holds a spam-ham text message data set. We can access this zip file and get the spam-ham text data as follows.

```
import requests
import io
from zipfile import ZipFile
zip_url = 'http://archive.ics.uci.edu/ml/machine-learning-
databases/00228/smsspamcollection.zip'
r = requests.get(zip_url)
z = ZipFile(io.BytesIO(r.content))
file = z.read('SMSSpamCollection')
text_data = file.decode()
text_data = text_data.encode('ascii',errors='ignore')
text_data = text_data.decode().split('\n')
text_data = [x.split('\t') for x in text_data if len(x)>=1]
[text_data_target, text_data_train] = [list(x) for x in zip(*text_
data)]
print(len(text_data_train))
5574
print(set(text_data_target))
{'ham', 'spam'}
print(text_data_train[1])
Ok lar... Joking wif u oni...
```

Movie review data. Bo Pang from Cornell has released a movie review data set that classifies reviews as good or bad (3). You can find the data on the website, <http://www.cs.cornell.edu/people/pabo/movie-review-data/rt-polaritydata.tar.gz>. To download, extract, and transform this data, we run the following code.

```
import requests
import io
import tarfile

movie_data_url = 'http://www.cs.cornell.edu/people/pabo/movie-review-
data/rt-polaritydata.tar.gz'
r = requests.get(movie_data_url)
# Stream data into temp object
stream_data = io.BytesIO(r.content)
tmp = io.BytesIO()
while True:
    s = stream_data.read(16384)
    if not s:
        break
    tmp.write(s)
stream_data.close()
tmp.seek(0)
```

```
# Extract tar file
tar_file = tarfile.open(fileobj=tmp, mode="r:gz")
pos = tar_file.extractfile('rt-polaritydata/rt-polarity.pos')
neg = tar_file.extractfile('rt-polaritydata/rt-polarity.neg')
# Save pos/neg reviews (Also deal with encoding)
pos_data = []
for line in pos:
    pos_data.append(line.decode('ISO-8859-1').
    encode('ascii',errors='ignore').decode())
neg_data = []
for line in neg:
    neg_data.append(line.decode('ISO-8859-1').
    encode('ascii',errors='ignore').decode())
tar_file.close()
print(len(pos_data))
5331
print(len(neg_data))
5331
print(neg_data[0]) # Print out first negative review
simplistic , silly and tedious .
```

CIFAR-10 image data. The Canadian Institute For Advanced Research has released a image set that contains 80 million labelled colored images (each image is scaled to 32x32 pixels). There 10 different target classes (airplane, automobile, bird, ...). The CIFAR-10 is a subset that is 60,000 images. There are 50,000 images in the training set, and 10,000 in the test set. Since we will be using this data set in multiple ways, and because it is one of our larger data sets, we will not run a script each time we need it. To get this data set, please navigate to <http://www.cs.toronto.edu/~kriz/cifar.html>, and download the CIFAR-10 dataset. We will address how to use this data set in the appropriate chapters.

The works of Shakespeare text data. Project Gutenberg (5) is a project that releases electronic versions of free books. They have compiled all of the works of Shakespeare together and here is how to access the text file through python.

```
import requests
shakespeare_url = 'http://www.gutenberg.org/cache/epub/100/pg100.txt'
# Get Shakespeare text
response = requests.get(shakespeare_url)
shakespeare_file = response.content
# Decode binary into string
shakespeare_text = shakespeare_file.decode('utf-8')
# Drop first few descriptive paragraphs.
shakespeare_text = shakespeare_text[7675:]
print(len(shakespeare_text)) # Number of characters
5582212
```

English-German sentence translation data. The Tatoeba project (<http://tatoeba.org>) collects sentence translations in many languages. Their data has been released under the Creative Commons License. From this data, ManyThings.org (<http://www.manythings.org>) has compiled sentence to sentence translations in text files available for download. Here we will use the English-German translation file, but you can change the url to whichever languages you would like to use.

```
import requests
import io
from zipfile import ZipFile
sentence_url = 'http://www.manythings.org/anki/deu-eng.zip'
r = requests.get(sentence_url)
z = ZipFile(io.BytesIO(r.content))
file = z.read('deu.txt')
# Format Data
eng_ger_data = file.decode()
eng_ger_data = eng_ger_data.encode('ascii', errors='ignore')
eng_ger_data = eng_ger_data.decode().split('\n')
eng_ger_data = [x.split('\t') for x in eng_ger_data if len(x)>=1]
[english_sentence, german_sentence] = [list(x) for x in zip(*eng_ger_data)]
print(len(english_sentence))
137673
print(len(german_sentence))
137673
print(eng_ger_data[10])
['I won!', 'Ich habe gewonnen!']
```

How it works

When it comes time to use one of these data sets in a recipe, we will refer you to this section and assume that the data is loaded in such a way as described above. If further data transformation or pre-processing is needed, then such code will be provided in the recipe itself.

See also

Hosmer, D.W., Lemeshow, S., and Sturdivant, R. X. (2013). Applied Logistic Regression: 3rd Edition. [<https://www.umass.edu/statdata/statdata/data/lowbwt.txt>]

Lichman, M. (2013). UCI Machine Learning Repository. [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan, Thumbs up? Sentiment Classification using Machine Learning Techniques, Proceedings of EMNLP 2002. [<http://www.cs.cornell.edu/people/pabo/movie-review-data/>]

Krizhevsky. (2009). Learning Multiple Layers of Features from Tiny Images. [<http://www.cs.toronto.edu/~kriz/cifar.html>]

Project Guetnberg. Accessed April 2016. <http://www.gutenberg.org/>.

Additional Resources

Getting ready

When learning how to use Tensorflow, it helps to know where to turn to for assistance or pointers. This section lists resources to get Tensorflow running and to troubleshoot problems.

How to do it

Here is a list of popular resources.

The official Tensorflow Python API documentation is located at https://www.tensorflow.org/versions/r0.8/api_docs/python/index.html. Here there is documentation and examples on all of the functions, objects, and methods in Tensorflow. Note the version number 'r0.8' in the link and realize that a more current version may be available.

Tensorflow's official tutorials are very thorough and detailed. They are located at <https://www.tensorflow.org/versions/r0.8/tutorials/index.html>. They start at covering image recognition models, and work through Word2Vec, RNN models, and sequence to sequence models. They also have additional tutorials on generating fractals and solving a PDE system. Know that they are continually adding more tutorials and examples to this collection.

Tensorflow's official Github repository is available via <https://github.com/tensorflow/tensorflow>. Here you can view the open sourced code and even fork or clone the most current version of the code if you want. You can also see current filed issues if you navigate to the issues directory.

A downloadable virtual machine that contains Tensorflow 0.8 installed on an Ubuntu 15.04 OS is available as well. This option is great for running Tensorflow on a Windows PC. The VM is available through a Google Document request form here: https://docs.google.com/forms/d/1mUztUIK6_z31BbMW5ihXaYHlhBcbDd94mERe-8XHyoI/viewform. It is about a 2GB download and requires VMWare player to run. VMWare player is a product made by VMWare and is free for personal use here: <https://www.vmware.com/go/downloadplayer/>. This virtual machine is maintained by David Winters (1).

A great source for community help is Stack Overflow. There is a tag for 'Tensorflow'. This tag seems to be growing in interest as Tensorflow is gaining more popularity. To view activity on this tag, visit <http://stackoverflow.com/questions/tagged/Tensorflow>

While Tensorflow is very agile and can be used for many things, the most common usage of Tensorflow is deep learning. To understand the basis for deep learning, how the underlying mathematics works, and to develop more intuition on deep learning, Google has created an online course available on Udacity. To sign up and take the video lecture course visit <https://www.udacity.com/course/deep-learning-ud730>.

Tensorflow has also made a site where you can visually explore training a neural network while changing the parameters and data sets. Visit <http://playground.tensorflow.org/> to explore how different settings affect the training of neural networks

See also

Winters, D. [https://docs.google.com/forms/d/1mUztUIK6_z31BbMW5ihXaYHlhBcbDd94mERe-8XHyoI/viewform]

2

The Tensorflow Way

In this chapter, we will introduce the key components of how Tensor flow operates. Then we will tie it together to create a simple classifier and evaluate the outcomes.

- ▶ Operations in a Computational Graph
- ▶ Layering Nested Operations
- ▶ Working with Multiple Layers
- ▶ Implementing Loss Functions
- ▶ Implementing Back Propagation
- ▶ Working with Batch and Stochastic Training
- ▶ Combining Everything Together
- ▶ Evaluating Models

Introduction

Now that we have introduced how Tensorflow creates tensors, uses variables and placeholders, we will introduce how to act on these objects in a computational graph. From this, we can setup a simple classifier and see how well it performs.

Operations in a Computational Graph

Now that we can put objects into our computational graph, we will introduce operations that act on such objects.

Getting ready

To start a graph, we load Tensorflow and create a session.

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

How to do it

In this example we will combine what we have learned and feed in each number in a list to an operation in a graph and print the output.

First we declare our tensors and placeholders. Here we will create a numpy array to feed into our operation.

```
import numpy as np
x_vals = np.array([1., 3., 5., 7., 9.])
x_data = tf.placeholder(tf.float32)
m_const = tf.constant(3.)
```

Then we declare our operation.

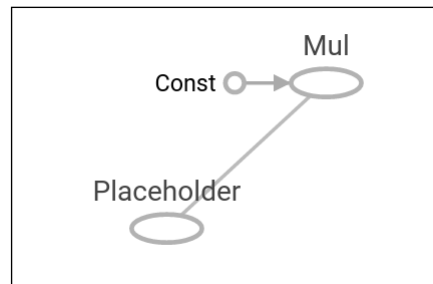
```
my_product = tf.mul(x_data, m_const)
```

Now we tell Tensorflow to loop through the values and feed them through the graph and print the output.

```
for x_val in x_vals:
    print(sess.run(my_product, feed_dict={x_data: x_val}))
3.0
9.0
15.0
21.0
27.0
```

How it works

Steps 1 and 2 above create the data and operations on the computational graph. Then in step 3, we feed the data through the graph and print the output. Here is what the computational graph looks like.



Here we can see in the graph that the placeholder, 'x_data' along with our multiplicative constant, feeds into the multiplication operation.

Layering Nested Operations

Getting ready

It's important to know how to chain operations together. This will setup layered operations in the computational graph. For a demonstration we will multiply a placeholder by two matrices and then perform addition. We will feed in two matrices in the form of a three dimensional numpy array.

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

How to do it

It is also important to note how the data will change shape as it passes through. We will feed in two numpy arrays of size 3x5. We will multiply each matrix by a constant of size 5x1 which will result in a matrix of size 3x1. We will then multiply this by 1x1 matrix resulting in a 3x1 matrix again. Finally, we add a 3x1 matrix at the end.

First we create the data to feed in and the corresponding placeholder.

```
my_array = np.array([[1., 3., 5., 7., 9.],
                    [-2., 0., 2., 4., 6.],
                    [-6., -3., 0., 3., 6.]])
x_vals = np.array([my_array, my_array + 1])
x_data = tf.placeholder(tf.float32, shape=(3, 5))
```

Next we create the constants that we will use for matrix multiplication and addition.

```
m1 = tf.constant([[1.], [0.], [-1.], [2.], [4.]])
m2 = tf.constant([[2.]])
a1 = tf.constant([[10.]])
```

Declare the operations and add them to the graph.

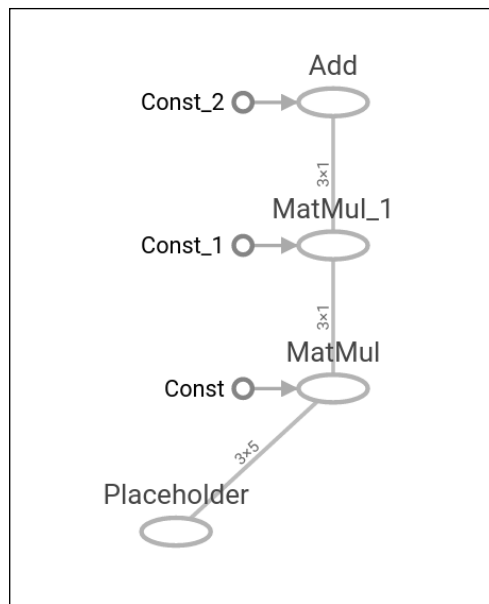
```
prod1 = tf.matmul(x_data, m1)
prod2 = tf.matmul(prod1, m2)
add1 = tf.add(prod2, a1)
```

Finally, we feed the data through our graph as follows.

```
for x_val in x_vals:
    print(sess.run(add1, feed_dict={x_data: x_val}))
[[ 102.]
 [  66.]
 [  58.]]
[[ 114.]
 [  78.]
 [  70.]]
```

How it works

The computational graph we just created can be visualized with Tensorboard. To see how this is done, see *Chapter 11*, section 1. Here is what our layered graph looks like.



In this computational graph you can see that the data size as it propagates upward through the graph.

There's more

Know that we have to declare the data shape and know the outcome shape of the operations before we run data through the graph. Sometimes this is not always the case. There may be a dimension or two that we do not know beforehand or that can vary. To accomplish this, we designate the dimension that can vary or is unknown as value none. For example, to have the prior data placeholder have an unknown amount of columns, we would write the following line.

```
x_data = tf.placeholder(tf.float32, shape=(3,None))
```

Know that this doesn't allow us to break matrix multiplication rules and we must still obey the fact that the multiplying constant must have the same corresponding amount of rows. We can either generate this dynamically or reshape the `x_data` as we feed data in our graph. This will come in handy in later chapters when we are feeding data in multiple batches.

Working With Multiple Layers

Getting ready

In this recipe we will introduce how to best connect various layers, including a custom layer. The data we will generate and use will be representative of small random images. It is best to understand these type of operations on a simple example and how we can use some built in layers to perform calculations. We will do a small moving window average across a 2d image and then flow the resulting output through a custom operation layer.

In this section, we will see that the computational graph can get large and hard to look at. To address this, we will also introduce ways to name operations and create scopes for layers. To start, load numpy and Tensorflow and create a graph.

```
import tensorflow as tf
import numpy as np
sess = tf.Session()
```

How to do it

First we create our sample 2D image with numpy. This image will be a 4x4 pixel image. We will create it in four dimensions, the first and last dimension will have size one. Note that some Tensorflow image functions will operate on 4 dimensional images. Those four dimensions are (image number, height, width, channel), and to make it one image with one channel, we set two of the dimensions to 1.

```
x_shape = [1, 4, 4, 1]
x_val = np.random.uniform(size=x_shape)
```

Now we have to create the placeholder in our graph where we can feed in the sample image.

```
x_data = tf.placeholder(tf.float32, shape=x_shape)
```

To create a moving window average across our 4x4 image, we will use a built in function that will convolute a constant across a window of shape 2x2. This function is quite common to use in image processing and in Tensor flow, the function we will use is 'conv2d()'. This function takes a piecewise product of the window and a filter we specify. We must also specify a stride for the moving window in both directions. Here we will compute 4 moving window averages, the top left, top right, bottom left, and bottom right 4 pixels. We do this by creating a 2x2 window and having strides of length 2 in each direction. To take the average, we will convolute the 2x2 window with a constant of 0.25.

```
my_filter = tf.constant(0.25, shape=[2, 2, 1, 1])
my_strides = [1, 2, 2, 1]
mov_avg_layer= tf.nn.conv2d(x_data, my_filter, my_strides,
                             padding='SAME', name='Moving_Avg_Window')
```

Note that we are also naming this layer "Moving_Avg_Window" by using the name argument of the function.

Now we define a custom layer that will operate on the 2x2 output of the moving window average. The custom function will first multiply the input by another 2x2 matrix tensor, and then add 1 to each entry. After this we take the sigmoid of each element and return the 2x2 matrix. Since matrix multiplication only operates on 2 dimensional matrices, we need to drop the extra dimensions of our image that are of size 1. Tensorflow can do this by the built in function 'squeeze()'. Here we define the new layer.

```
def custom_layer(input_matrix):
    input_matrix_squeezed = tf.squeeze(input_matrix)
    A = tf.constant([[1., 2.], [-1., 3.]])
    b = tf.constant(1., shape=[2, 2])
    temp1 = tf.matmul(A, input_matrix_squeezed)
    temp = tf.add(temp1, b) # Ax + b
    return(tf.sigmoid(temp))
```

Now we have to place the new layer on the graph. We will do this with a named scope so that it is identifiable and collapsible/expandable on the computational graph.

```
with tf.name_scope('Custom_Layer') as scope:
    custom_layer1 = custom_layer(mov_avg_layer)
```

Now we just feed in the 4x4 image in the placeholder and tell Tensorflow to run the graph.

```
print(sess.run(custom_layer1, feed_dict={x_data: x_val}))
[[ 0.91914582  0.96025133]
 [ 0.87262219  0.9469803  ]]
```

How it works

The visualized graph looks better with naming of operations and scoping of layers. We can collapse and expand the custom layer because we created it in a named scope. In the following figure, see the collapsed version on the left and the expanded version on the right.

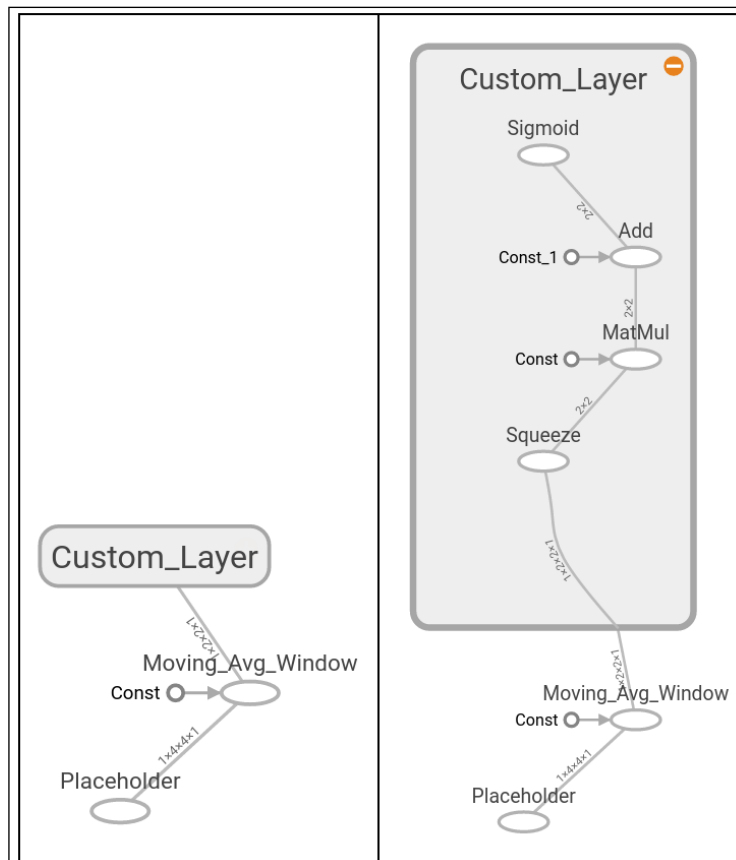


Figure of the computational graph with two layers. The first layer is named as 'Moving_Avg_Window', and the second is a collection of operations called 'Custom_Layer'. It is collapsed on the left and expanded on the right.

Implementing Loss Functions

Getting ready

In order to optimize an algorithm of our specification, we will need to evaluate the outcomes. Evaluating outcomes in Tensorflow depend on specifying a loss function. A loss function tells Tensorflow how good or bad the predictions are compared to the desired result. In most cases, we will have a set of data and a target on which to train our algorithm. The loss function compares the target to the predicted and gives a numerical distance between the two.

For this recipe, we will cover the main loss functions that we can implement in Tensorflow.

To see how the different loss functions operate, we will plot them in this recipe. We will first start a computational graph and load matplotlib, a python plotting library.

```
import matplotlib.pyplot as plt
import tensorflow as tf
```

How to do it

First we will talk about loss functions for regression that is, predicting a continuous dependent variable. To start, we will create a sequence of our predictions and a target as a tensor. We will output the results across 500 x-values between -1 and 1. See the next section for a plot of the outputs.

```
x_vals = tf.linspace(-1., 1., 500)
target = tf.constant(0.)
```

The L2 norm loss is also known as the Euclidean loss function. It is just the square of the distance to the target. Here we will compute the loss function as if the target is zero. The L2 norm is a great loss function because it is very curved near the target and algorithms can use this fact to converge to the target slower the closer it gets.

```
l2_y_vals = tf.square(target - x_vals)
l2_y_out = sess.run(l2_y_vals)
```

Note: Tensorflow has a built in form of the L2 norm, called `nn.l2_loss()`. This function is actually half the l2-norm above. In other words, it is the same as above but divided by 2.

The L1 norm loss is also known as the absolute loss function. Instead of squaring the difference, we take the absolute value. The L1 norm is better for outliers than the L2 norm because it is not as steep for larger values. One issue to be aware of is that the L1 norm is not smooth at the target and this can result in algorithms not converging well.

```
l1_y_vals = tf.abs(target - x_vals)
l1_y_out = sess.run(l1_y_vals)
```

Pseudo-Huber Loss is a continuous and smooth approximation to the Huber loss function. This loss function attempts to take the best of the L1 and L2 norms by being convex near the target and less steep for extreme values. The form depends on an extra parameter, delta, which dictates how steep it will be. We will plot two forms, delta1 = 0.25 and delta2 = 5 to show the difference.

```
delta1 = tf.constant(0.25)
phuber1_y_vals = tf.mul(tf.square(delta1), tf.sqrt(1. +
            tf.square((target - x_vals)/delta1)) - 1.)
phuber1_y_out = sess.run(phuber1_y_vals)

delta2 = tf.constant(5.)
phuber2_y_vals = tf.mul(tf.square(delta2), tf.sqrt(1. +
            tf.square((target - x_vals)/delta2)) - 1.)
phuber2_y_out = sess.run(phuber2_y_vals)
```

Classification loss functions are used to evaluate loss when predicting categorical outcomes.

We will need to redefine our predictions (x_vals) and target. We will save the outputs and plot them in the next section.

```
x_vals = tf.linspace(-3., 5., 500)
target = tf.constant(1.)
targets = tf.fill([500,], 1.)
```

Hinge loss is mostly used for support vector machines, but can be used in neural networks as well. It is meant to compute a loss between two target classes, 1 and -1. In the following code, we are using target value 1, so the closer our predictions are to 1, the lower the loss value.

```
hinge_y_vals = tf.maximum(0., 1. - tf.mul(target, x_vals))
hinge_y_out = sess.run(hinge_y_vals)
```

Cross entropy loss for a binary case is also sometimes referred to as the logistic loss function. It comes about when we are predicting the two classes 0 or 1. We wish to measure a distance from the actual class (0 or 1) to the predicted value, which is usually a real number between 0 and 1. To measure this distance we can use the cross entropy formula from information theory.

```
xentropy_y_vals = - tf.mul(target, tf.log(x_vals)) - tf.mul((1. -
target), tf.log(1. - x_vals))
xentropy_y_out = sess.run(xentropy_y_vals)
```

Sigmoid cross entropy loss is very similar to the above loss function except we transform the x-values by the sigmoid function before we put them in the cross entropy loss.

```
xentropy_sigmoid_y_vals = tf.nn.sigmoid_cross_entropy_with_logits(x_
vals, targets)
xentropy_sigmoid_y_out = sess.run(xentropy_sigmoid_y_vals)
```

Weighted cross entropy loss is a weighted version of the sigmoid cross entropy loss. We provide a weight on the positive target. For an example we will weight the positive target by 0.5.

```
weight = tf.constant(0.5)
xentropy_weighted_y_vals = tf.nn.weighted_cross_entropy_with_logits(x_
vals, targets, weight)
xentropy_weighted_y_out = sess.run(xentropy_weighted_y_vals)
```

Softmax cross entropy loss operates on non-normalized outputs. This function is used to measure a loss when there is only one target category instead of multiple. Because of this, the function transforms the outputs into a probability distribution via the softmax function and then computes the loss function from a true probability distribution.

```
unscaled_logits = tf.constant([[1., -3., 10.]])
target_dist = tf.constant([[0.1, 0.02, 0.88]])
softmax_xentropy = tf.nn.softmax_cross_entropy_with_logits(unscaled_
logits, target_dist)
print(sess.run(softmax_xentropy))
[ 1.16012561]
```

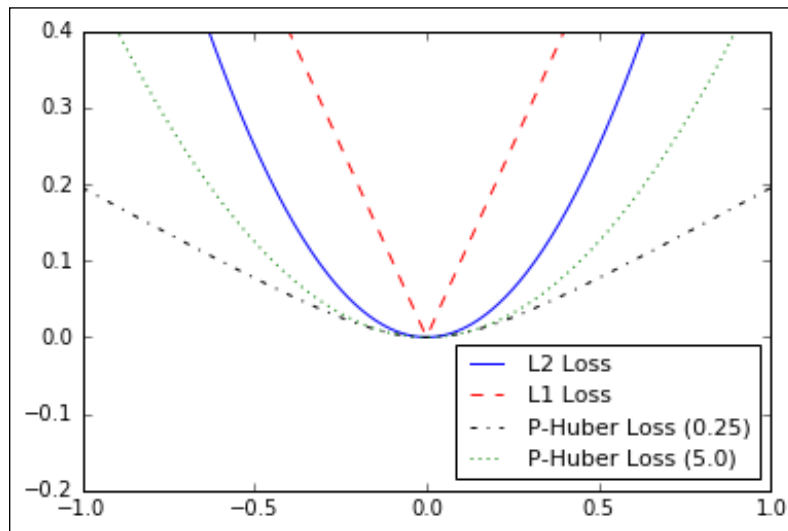
Sparse softmax cross entropy loss is the same as above, except instead of the target being a probability distribution, it is an index of which category is true. Instead of a sparse all-zero target vector with one value of one, we just pass in the index of which category is the true value.

```
unscaled_logits = tf.constant([[1., -3., 10.]])
sparse_target_dist = tf.constant([2])
sparse_xentropy = tf.nn.sparse_softmax_cross_entropy_with_
logits(unscaled_logits, sparse_target_dist)
print(sess.run(sparse_xentropy))
[ 0.00012564]
```

How it works

Here is how to use matplotlib to plot the regression loss functions.

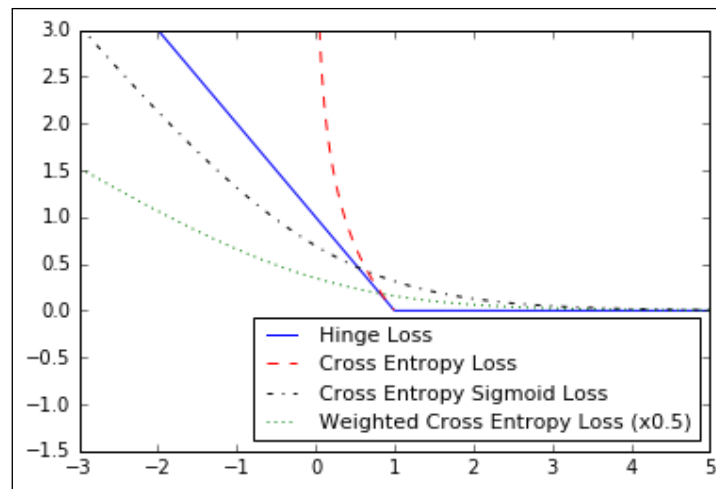
```
x_array = sess.run(x_vals)
plt.plot(x_array, l2_y_out, 'b-', label='L2 Loss')
plt.plot(x_array, l1_y_out, 'r--', label='L1 Loss')
plt.plot(x_array, phuber1_y_out, 'k-.', label='P-Huber Loss (0.25)')
plt.plot(x_array, phuber2_y_out, 'g:', label='P-Huber Loss (5.0)')
plt.ylim(-0.2, 0.4)
plt.legend(loc='lower right', prop={'size': 11})
plt.show()
```



Plotting various regression loss functions.

And here is how to use matplotlib to plot the various classification loss functions.

```
x_array = sess.run(x_vals)
plt.plot(x_array, hinge_y_out, 'b-', label='Hinge Loss')
plt.plot(x_array, xentropy_y_out, 'r--', label='Cross Entropy Loss')
plt.plot(x_array, xentropy_sigmoid_y_out, 'k-.', label='Cross Entropy Sigmoid Loss')
plt.plot(x_array, xentropy_weighted_y_out, 'g:', label='Weighted Cross Entropy Loss (x0.5)')
plt.ylim(-1.5, 3)
plt.legend(loc='lower right', prop={'size': 11})
plt.show()
```



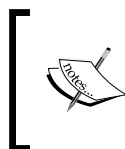
Plots of classification loss functions

There's more

Here is a table summarizing the different loss functions that we have described.

Loss Function	Use	Benefits	Disadvantages
L2	Regression	More stable	Less robust
L1	Regression	More robust	Less stable
Pseudo Huber	Regression	More robust and stable	One more parameter
Hinge	Classification	Creates a max margin for use in SVM	Unbounded loss affected by outliers
Cross Entropy	Classification	More stable	Unbounded loss, less robust

The remaining classification loss functions all have to do with the type of cross entropy loss. The cross entropy sigmoid loss function is for use on unscaled logits and is preferred over computing the sigmoid and then the cross entropy because Tensorflow has built in better ways to handle numerical edge cases. The same goes for softmax cross entropy and sparse softmax cross entropy.



Note: Most of the classification loss functions described here are for two class predictions. This can be extended to multiple classes via summing the cross entropy terms over each prediction/target.

Implementing Back Propagation

Getting ready

Now we will introduce how to change our variables in the model such that a loss function is minimized. We have learned about how to use objects, operations, and create loss functions that will measure the distance between our predictions and targets. Now we just have to tell Tensorflow how to back propagate errors through our computational graph to update the variables and minimize the loss function. This is done via declaring an optimization function. Once we have an optimization function declared, Tensorflow will go through and figure out the back propagation terms for all of our computations in the graph. When we feed data in and minimize the loss function, Tensorflow will modify our variables in the graph accordingly.

There's a lot to do here, so we will do a very simple regression algorithm. We will sample random numbers from a normal, with mean 1 and standard deviation 0.1. Then we will run the numbers through one operation, which will be to multiply them by a variable, A. From this the loss function will be the L2 norm between the output and the target, which will always be the value 10. Theoretically, the best value for A will be the number 10 since our data will have mean 1.

The second example is a very simple binary classification algorithm. Here we will generate 100 numbers from two normal distributions, $N(-1,1)$ and $N(3,1)$. All the numbers from $N(-1, 1)$ will be in the target class 0, and all the numbers from $N(3, 1)$ will be in the target class 1. The model to differentiate these numbers will be a sigmoid function of a translation. In other words the model will be $\text{sigmoid}(x + A)$ where A is a variable we will fit. Theoretically, A will be equal to -1. We arrive at this number because if m1 and m2 are the means of the two normal functions, the value added to them to translate them equidistant to zero will be $-(m1+m2)/2$. We will see how Tensorflow can arrive at that number in the second example.

How to do it

Here is how the regression example works.

We start by loading the numerical python package, numpy and tensorflow.

```
import numpy as np
import tensorflow as tf
```

Now we start a graph session.

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

Next, we create the data, placeholders, and the A variable.

```
x_vals = np.random.normal(1, 0.1, 100)
y_vals = np.repeat(10., 100)
```

```
x_data = tf.placeholder(shape=[1], dtype=tf.float32)
y_target = tf.placeholder(shape=[1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[1]))
```

We add the multiplication operation to our graph.

```
my_output = tf.mul(x_data, A)
```

Next we add our L2 loss function between the multiplication output and the target data.

```
loss = tf.square(my_output - y_target)
```

Before we can run anything we have to initialize the variables.

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

Now we have to declare a way to optimize the variables in our graph. We declare an optimizer algorithm. Most optimization algorithms need to know how far to step in each iteration. This distance is controlled by the learning rate. If our learning rate is too big, our algorithm might overshoot the minimum, but if our learning rate is too small, our algorithm might take too long to converge. The learning rate has a big influence on convergence and we will discuss this at the end of the section.

```
my_opt = tf.train.GradientDescentOptimizer(learning_rate=0.02)
train_step = my_opt.minimize(loss)
```

The final step is to loop through our training algorithm and tell Tensorflow to train many times. We will do this 101 times and print out results every 25th iteration. To train, we will select a random x and y entry and feed it through the graph. Tensorflow will automatically compute the loss, and slightly change the A variables to minimize the loss.

```
for i in range(100):
    rand_index = np.random.choice(100)
    rand_x = [x_vals[rand_index]]
    rand_y = [y_vals[rand_index]]
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%25==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(loss, feed_dict={x_data:
rand_x, y_target: rand_y})))
Here is the output:
Step #25 A = [ 6.23402166]
Loss = 16.3173
Step #50 A = [ 8.50733757]
Loss = 3.56651
```

```
Step #75 A = [ 9.37753201]
Loss = 3.03149
Step #100 A = [ 9.80041122]
Loss = 0.0990248
```

Now we will introduce the code for the simple classification example. We can use the same Tensorflow script if we reset the graph first. Remember we will attempt to find an optimal translation, A , that will translate the two distributions to the origin and the sigmoid function will split the two into two different classes.

First we reset the graph and reinitialize the graph session.

```
from tensorflow.python.framework import ops
ops.reset_default_graph()
sess = tf.Session()
```

Next we will create the data from two different normal distributions, $N(-1, 1)$ and $N(3, 1)$. We will also generate the target labels, placeholders for the data, and the variable, A .

```
x_vals = np.concatenate((np.random.normal(-1, 1, 50), np.random.
normal(3, 1, 50)))
y_vals = np.concatenate((np.repeat(0., 50), np.repeat(1., 50)))
x_data = tf.placeholder(shape=[1], dtype=tf.float32)
y_target = tf.placeholder(shape=[1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(mean=10, shape=[1]))
```



Note that we initialized A to around the value 10, far from the theoretical value of -1. We did this on purpose to show how the algorithm converges from the value 10 to the optimal value, -1.

Next we add the translation operation to the graph. Remember that we do not have to wrap this in a sigmoid function because the loss function will do that for us.

```
my_output = tf.add(x_data, A)
```

Because the specific loss function expects batches of data, which have an extra dimension associated with them (an added dimension which is the batch number), we will add an extra dimension to the output with the function, 'expand_dims()' In the next section we will discuss how to use variable sized batches in training. For now, we will again just use one random data point at a time.

```
my_output_expanded = tf.expand_dims(my_output, 0)
y_target_expanded = tf.expand_dims(y_target, 0)
```

Next we will initialize our one variable, A .

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

Now we declare our loss function. We will use a cross entropy with unscaled logits that transforms them with a sigmoid function. Tensorflow has this all in one function for us in the neural network package called 'nn.sigmoid_cross_entropy_with_logits()'. As stated before, it expects the arguments to have specific dimensions, so we have to use the expanded outputs and targets accordingly.

```
xentropy = tf.nn.sigmoid_cross_entropy_with_logits( my_output_
expanded, y_target_expanded)
```

Just like the regression example, we need to add an optimizer function to the graph so that Tensorflow knows how to update the variables in the graph.

```
my_opt = tf.train.GradientDescentOptimizer(0.05)
```

```
train_step = my_opt.minimize(xentropy)
```

Finally, we loop through a randomly selected data point several hundred times and update the variable A accordingly. Every 200 iterations we will print out the value of A and the loss.

```
for i in range(1400):
    rand_index = np.random.choice(100)
    rand_x = [x_vals[rand_index]]
    rand_y = [y_vals[rand_index]]

    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%200==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(xentropy, feed_dict={x_data:
rand_x, y_target: rand_y})))
Step #200 A = [ 3.59597969]
Loss = [[ 0.00126199]]
Step #400 A = [ 0.50947344]
Loss = [[ 0.01149425]]
Step #600 A = [-0.50994617]
Loss = [[ 0.14271219]]
Step #800 A = [-0.76606178]
Loss = [[ 0.18807337]]
Step #1000 A = [-0.90859312]
Loss = [[ 0.02346182]]
Step #1200 A = [-0.86169094]
Loss = [[ 0.05427232]]
Step #1400 A = [-1.08486211]
Loss = [[ 0.04099189]]
```

How it works

As a recap, for both examples, we

1. Created the data,
2. Initialized placeholders and variables,
3. Created a loss function,
4. Defined an optimization algorithm,
5. And finally, iterated across random data samples to iteratively update our variables.

There's more

We've mentioned before that the optimization algorithm is sensitive to the choice of the learning rate. It is important to summarize the effect of this choice in a concise manner.

	Advantages/Disadvantages	Uses
Smaller Learning Rate	Converges slower but more accurate results.	If solution is unstable, try lowering the learning rate first.
Larger Learning Rate	Less accurate, but converges faster.	For some problems, helps prevent solutions from stagnating.

While specifying a good learning rate helps convergence of algorithms, we must also specify a type of optimization. For the above two examples, we are using standard gradient descent. This is implemented with the Tensorflow function `GradientDescentOptimizer()`.

Sometimes the standard gradient descent algorithm can get stuck or slow down significantly. This can happen when the optimization is stuck in the flat spot of a saddle. To combat this, there is another algorithm that takes into account a momentum term, which adds on a fraction of the prior step's gradient descent value. Tensorflow has this built in with the `MomentumOptimizer()` function.

Another variant is to vary the optimizer step for each variable in our models. Ideally, we would like to take larger steps for smaller moving variables and shorter steps for faster changing variables. We will not go into the mathematics of this approach, but a common implementation of this idea is called the Adagrad algorithm. This algorithm takes into account the whole history of the variable gradients. Again, the function in Tensorflow for this is called `AdagradOptimizer()`.

Sometimes, Adagrad forces the gradients to zero too soon because it takes into account the whole history. A solution to this is to limit how many steps we use. Doing this is called the Adadelta algorithm. We can apply this by using the function `AdadeltaOptimizer()`.

There are a few other implementations of different gradient descent algorithms. For these, we would refer the reader to the Tensorflow documentation at

https://www.tensorflow.org/versions/r0.8/api_docs/python/train.html#optimizers.

Working with Batch and Stochastic Training

Getting ready

In order for Tensorflow to compute the variable gradients for back propagation to work, we have to measure the loss on a sample or multiple samples. Stochastic training is only putting through one randomly sampled data-target pair at a time, just like we did in the prior recipe. Another option is to put a larger portion of the training examples in at a time and average the loss for the gradient calculation. Batch training size can vary up to and including the whole data set at once. Here we will show how to extend the prior regression example, which used stochastic training to batch training.

We will start by loading numpy, matplotlib, and tensorflow and start a graph session.

```
import matplotlib as plt
import numpy as np
import tensorflow as tf
sess = tf.Session()
```

How to do it

We'll start by declaring a batch size. This will be how many random observations we will compute the loss on at once.

```
batch_size = 20
```

Next we declare the data, placeholders, and the variable in the model. The change we make here is we change the shape of the placeholders. They are now two dimensions, where the first dimension is 'None', and will be the number of data points in the batch. We could have explicitly set it to 20, but we can generalize it and use the 'None' value. Again, as mentioned in *chapter 1*, we still have to make sure that the dimensions work out in the model and this does not allow us to perform any illegal matrix operations.

```
x_vals = np.random.normal(1, 0.1, 100)
y_vals = np.repeat(10., 100)
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[1,1]))
```

Now we add our operation to the graph, which will now be matrix multiplication instead of regular multiplication. Remember that matrix multiplication is not commutative so we have to enter the matrices in the correct order in the `matmul()` function.

```
my_output = tf.matmul(x_data, A)
```

Our loss function will change because we have to take the mean of all the L2 losses of each data point in the batch. We do this by wrapping our prior loss output in Tensorflow's `reduce_mean()` function.

```
loss = tf.reduce_mean(tf.square(my_output - y_target))
```

We declare our optimizer just like we did before.

```
my_opt = tf.train.GradientDescentOptimizer(0.02)
train_step = my_opt.minimize(loss)
```

Finally, we will loop through and iterate on the training step to optimize the algorithm. This part is different than before because we want to be able to plot the loss over time and compare the batch vs stochastic training convergence. So we initialize a list to store the loss function every 5 intervals.

```
loss_batch = []
for i in range(100):
    rand_index = np.random.choice(100, size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%5==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
        print('Loss = ' + str(temp_loss))
        loss_batch.append(temp_loss)
```

Here is the final output of the 100 iterations. Notice that the value of A has an extra dimension because it now has to be a 2D matrix.

```
Step #100 A = [[ 9.86720943]]
Loss = 0.
```

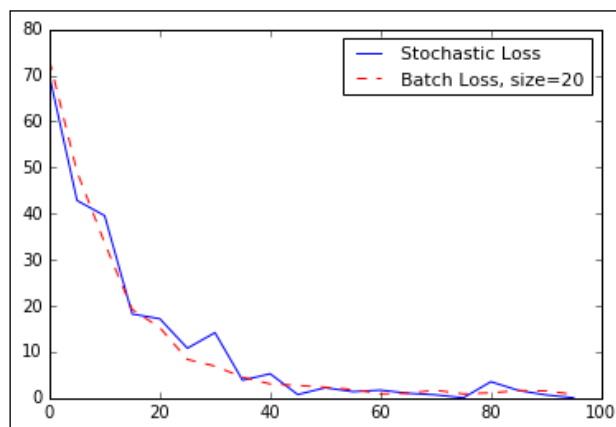
How it works

Batch training and stochastic training differ in their optimization method and their convergence. Finding a good batch size can be difficult. To see how convergence differs between batch and stochastic, here is the code to plot the batch loss from above. There is also a variable here that contains the stochastic loss, but that computation follows from the prior section in this chapter. Here is code to save and record the stochastic loss in the training loop. Just substitute this code in the prior recipe.

```
loss_stochastic = []
for i in range(100):
    rand_index = np.random.choice(100)
    rand_x = [x_vals[rand_index]]
    rand_y = [y_vals[rand_index]]
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%5==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
        print('Loss = ' + str(temp_loss))
        loss_stochastic.append(temp_loss)
```

Here is the code to produce the plot of both the stochastic and batch loss for the same regression problem.

```
plt.plot(range(0, 100, 5), loss_stochastic, 'b-', label='Stochastic
Loss')
plt.plot(range(0, 100, 5), loss_batch, 'r--', label='Batch Loss,
size=20')
plt.legend(loc='upper right', prop={'size': 11})
plt.show()
```



Stochastic loss and batch loss (batch size = 20) plotted over 100 iterations. Note that the batch loss is much smoother and the stochastic loss is much more erratic.

There's more

Type of training	Advantages	Disadvantages
Stochastic	Randomness may help move out of local minimums.	Generally, needs more iterations to converge.
Batch	Finds minimums quicker.	Takes more resources to compute.

Combining Everything Together

Getting ready

In this section we will combine everything we have illustrated so far and create a classifier on the iris dataset. The iris data set is described in more detail in chapter 1, under 'Working with Data Sources'. We will load this data, and do a simple binary classifier to predict if a flower is the species *Iris setosa* or not. To be clear, this dataset has three classes of species, but we will only predict if it is a single species (*I. setosa*) or not, giving us a binary classifier. We will start by loading the libraries and data, then transform the target accordingly.

How to do it

First we load the libraries needed and initialize the computational graph. Note that we also load matplotlib here, because we will want to plot the resulting line after.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
import tensorflow as tf
sess = tf.Session()
```

Next we load the iris data. We will also need to transform the target data to be just 1 or 0 if the target is *setosa* or not. Since the iris data set marks *setosa* as a zero, we will change all targets with value 0 to 1, and the other values all to 0. We will also only use 2 features, petal length and petal width. These two features are the 3rd and 4th entry in each x-value.

```
iris = datasets.load_iris()
binary_target = np.array([1. if x==0 else 0. for x in iris.target])
iris_2d = np.array([[x[2], x[3]] for x in iris.data])
```

Let's declare our batch size, data placeholders, and model variables. Remember that the data placeholders for variable batch sizes have 'None' as the first dimension.

```
batch_size = 20
x1_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
x2_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[1, 1]))
b = tf.Variable(tf.random_normal(shape=[1, 1]))
```

Here we define the linear model. The model will take the form $x_2 = x_1 * A + b$. And if we want to find points above or below that line, we see if they are above or below zero when plugged into the equation $x_2 - x_1 * A - b$. We will do this by taking the sigmoid of that equation and predicting 1 or 0 from that equation. Remember that Tensorflow has loss functions with the sigmoid built in, so we just need to define the output of the model prior to the sigmoid.

```
my_mult = tf.matmul(x2_data, A)
my_add = tf.add(my_mult, b)
my_output = tf.sub(x1_data, my_add)
```

Now we add our sigmoid cross-entropy loss function with Tensorflow's built in function, 'sigmoid_cross_entropy_with_logits()'

```
xentropy = tf.nn.sigmoid_cross_entropy_with_logits(my_output, y_target)
```

We also have to tell Tensorflow how to optimize our computational graph by declaring an optimizing method. We will want to minimize the cross-entropy loss. We will also choose 0.05 as our learning rate.

```
my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(xentropy)
```

Now we create a variable initialization operation and tell Tensorflow to execute it.

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

Now we will train our linear model with 1000 iterations. We feed in the three data points we need, petal length, petal width, and the target variable. Every 200 generations we will print the variable values.

```
for i in range(1000):
    rand_index = np.random.choice(len(iris_2d), size=batch_size)
    rand_x = iris_2d[rand_index]
    rand_x1 = np.array([[x[0]] for x in rand_x])
    rand_x2 = np.array([[x[1]] for x in rand_x])
    rand_y = np.array([[y] for y in binary_target[rand_index]])
```

```
sess.run(train_step, feed_dict={x1_data: rand_x1, x2_data: rand_
x2, y_target: rand_y})
if (i+1)%200==0:
    print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + ', b
= ' + str(sess.run(b)))
Step #200 A = [[ 8.67285347]], b = [[-3.47147632]]
Step #400 A = [[ 10.25393486]], b = [[-4.62928772]]
Step #600 A = [[ 11.152668]], b = [[-5.4077611]]
Step #800 A = [[ 11.81016064]], b = [[-5.96689034]]
Step #1000 A = [[ 12.41202831]], b = [[-6.34769201]]
```

The next set of commands extracts the model variables, and plots the line on a graph. The resulting graph is in the next section.

```
[[slope]] = sess.run(A)
[[intercept]] = sess.run(b)

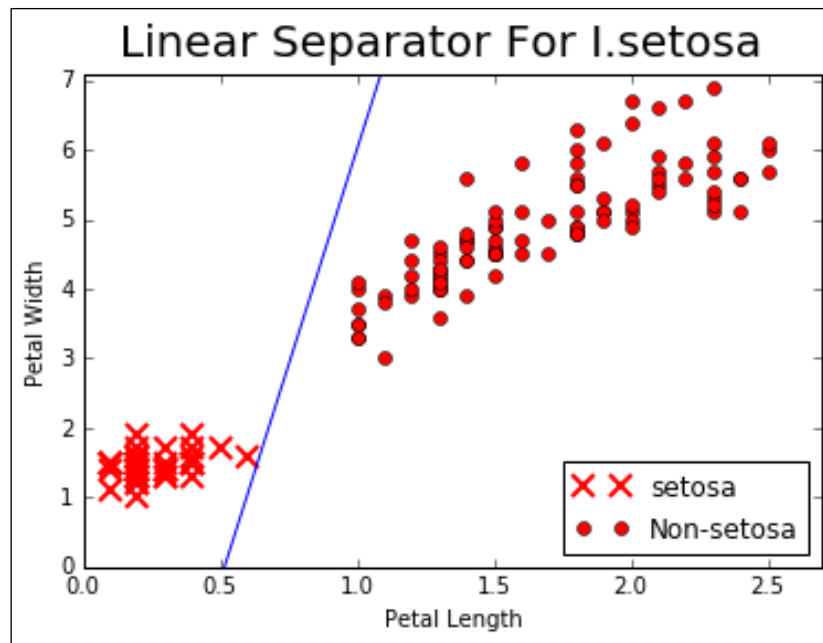
x = np.linspace(0, 3, num=50)
ablineValues = []
for i in x:
    ablineValues.append(slope*i+intercept)

setosa_x = [a[1] for i,a in enumerate(iris_2d) if binary_target[i]==1]
setosa_y = [a[0] for i,a in enumerate(iris_2d) if binary_target[i]==1]
non_setosa_x = [a[1] for i,a in enumerate(iris_2d) if binary_
target[i]==0]
non_setosa_y = [a[0] for i,a in enumerate(iris_2d) if binary_
target[i]==0]

plt.plot(setosa_x, setosa_y, 'rx', ms=10, mew=2, label='setosa')
plt.plot(non_setosa_x, non_setosa_y, 'ro', label='Non-setosa')
plt.plot(x, ablineValues, 'b-')
plt.xlim([0.0, 2.7])
plt.ylim([0.0, 7.1])
plt.suptitle('Linear Separator For I.setosa', fontsize=20)
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.legend(loc='lower right')
plt.show()
```

How it works

Our goal was to fit a line between the *I. setosa* points and the other two species using only petal width and petal length. If we plot the points and the resulting line, we see that we have achieved this.



Plot of *I. setosa* and non-setosa for petal width vs petal length. The blue line is the linear separator that we achieved after 1000 iterations.

There's more

While we achieved our objective of separating the two classes with a line, it may not be the best model for separating two classes. In chapter 4, we will discuss support vector machines, which is a better way of separating two classes in a feature space.

See also

For more information on the iris dataset, see the Wikipedia entry, https://en.wikipedia.org/wiki/Iris_flower_data_set. For information about the Scikit Learn iris dataset implementation, see the documentation at http://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html.

Evaluating Models

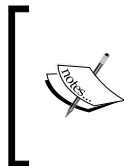
Getting ready

We have learned how to train a regression and classification algorithm in Tensorflow. After this is accomplished, we must be able to evaluate the model's predictions to determine how well it did. This is very important and every subsequent model will have some form of model evaluation. Using Tensorflow, we must build this feature into the computational graph and call it during and/or after our model is training.

Evaluating models during training gives us insight into the algorithm and may give us hints to debug it, improve it, or change models entirely. While evaluation during training isn't always necessary, we will show how to do this with both regression and classification.

After training, we need to quantify how the model performs on the data. Ideally, we have a separate training and test set (and even a validation set) where we can evaluate the model on.

Know that when we want to evaluate a model, we will want to do so on a large batch of data points. If we have implemented batch training, we can reuse our model to make prediction on such a batch. If we have implemented stochastic training, we may have to create a separate evaluator that can process data in batches.



Note: If we included a transformation on our model output in the loss function, e.g. `'sigmoid_cross_entropy_with_logits()'` we must take that into account when computing predictions for accuracy calculations. Don't forget to include this in our evaluation of the model.

How to do it

Regression models attempt to predict a continuous number. The target is not a category, but a desired number. To evaluate these regression predictions against the actual targets, we need an aggregate measure of the distance between the two. Most of the time, a meaningful loss function will satisfy these criteria. Here is how to change the simple regression algorithm from above into printing out the loss in the training loop and evaluating the loss at the end. For an example, we will revisit and rewrite our regression example in the prior 'Implementing Back Propagation' recipe in this chapter.

To start, we load our libraries; declare our graph, variables, and placeholders.

Next we

Classification models are predicting a category based on numerical inputs. The actual targets are a sequence of 1's and 0's and we must have a measure of how close we are to the truth from our predictions. The loss function for classification models usually isn't that helpful in interpreting how good our model is doing. Usually we want some sort of classification accuracy, which is commonly the percentage of correctly predicted categories. For this example, we will use the classification example from the prior *'Implementing Back Propagation'* recipe in this chapter.

How it works

First we will show how to evaluate the simple regression model that simply fits a constant multiplication to the target of 10.

First we start by loading the libraries, creating the graph, data, variables, and placeholders. There is an additional part to this section that is very important. After we create the data, we will split the data into a train and test dataset randomly. This is important because we will always test our models if they are predicting well or not. Evaluating the model both on the training data and test data also lets us see if the model is overfitting or not.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

sess = tf.Session()

x_vals = np.random.normal(1, 0.1, 100)
y_vals = np.repeat(10., 100)
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

batch_size = 25

train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8),
replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]

A = tf.Variable(tf.random_normal(shape=[1,1]))
```

Now we declare our model, loss function, and optimization algorithm. We will also initialize the model variable, A.

```
my_output = tf.matmul(x_data, A)
loss = tf.reduce_mean(tf.square(my_output - y_target))
init = tf.initialize_all_variables()
sess.run(init)
my_opt = tf.train.GradientDescentOptimizer(0.02)
train_step = my_opt.minimize(loss)
```

We run the training loop just as we would before.

```
for i in range(100):
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    rand_x = np.transpose([x_vals_train[rand_index]])
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%25==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(loss, feed_dict={x_data:
rand_x, y_target: rand_y})))

Step #25 A = [[ 6.39879179]]
Loss = 13.7903
Step #50 A = [[ 8.64770794]]
Loss = 2.53685
Step #75 A = [[ 9.40029907]]
Loss = 0.818259
Step #100 A = [[ 9.6809473]]
Loss = 1.10908
```

Now to evaluate the model, we will output the MSE (loss function) on the training and test set.

```
mse_test = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_
test]), y_target: np.transpose([y_vals_test])})
mse_train = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_
train]), y_target: np.transpose([y_vals_train])})
print('MSE on test:' + str(np.round(mse_test, 2)))
print('MSE on train:' + str(np.round(mse_train, 2)))

MSE on test:1.35
MSE on train:0.88
```

For the classification example, we will do something very similar. This time we will need to create our own accuracy function that we can call at the end. One reason for this is because our loss function has the sigmoid built in and we will need to call the sigmoid separately and test if our classes are correct.

6. In the same script, we can just reload the graph and create our data, variables, and placeholders. Remember that we will also need to separate the data and targets into train and test sets.

```
from tensorflow.python.framework import ops
ops.reset_default_graph()

sess = tf.Session()

batch_size = 25

x_vals = np.concatenate((np.random.normal(-1, 1, 50), np.random.
normal(2, 1, 50)))
y_vals = np.concatenate((np.repeat(0., 50), np.repeat(1., 50)))
x_data = tf.placeholder(shape=[1, None], dtype=tf.float32)
y_target = tf.placeholder(shape=[1, None], dtype=tf.float32)

train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]

A = tf.Variable(tf.random_normal(mean=10, shape=[1]))
We will now add the model and the loss function to the graph,
initialize variables, and create the optimization procedure.
my_output = tf.add(x_data, A)

init = tf.initialize_all_variables()
sess.run(init)

xentropy = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_
logits(my_output, y_target))

my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(xentropy)
```

7. Now we run our training loop.

```
for i in range(1800):
    rand_index = np.random.choice(len(x_vals_train), size=batch_
size)
    rand_x = [x_vals_train[rand_index]]
    rand_y = [y_vals_train[rand_index]]
```



```
sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})
    if (i+1)%200==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(xentropy, feed_dict={x_
data: rand_x, y_target: rand_y}))))
```

```
Step #200 A = [ 6.64970636]
Loss = 3.39434
Step #400 A = [ 2.2884655]
Loss = 0.456173
Step #600 A = [ 0.29109824]
Loss = 0.312162
Step #800 A = [-0.20045301]
Loss = 0.241349
Step #1000 A = [-0.33634067]
Loss = 0.376786
Step #1200 A = [-0.36866501]
Loss = 0.271654
Step #1400 A = [-0.3727718]
Loss = 0.294866
Step #1600 A = [-0.39153299]
Loss = 0.202275
Step #1800 A = [-0.36630616]
Loss = 0.358463
```

8. To evaluate the model, we will create our own prediction operation. We wrap the prediction operation in a squeeze function because we want to make the predictions and targets of the same shape. Then we test for equality with the equal function. After that we are left with a tensor of True and False values that we cast to float32 and take the mean of them. This will result in an accuracy value. We will evaluate this function for both the train and test set.

```
y_prediction = tf.squeeze(tf.round(tf.nn.sigmoid(tf.add(x_data,
A))))
correct_prediction = tf.equal(y_prediction, y_target)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

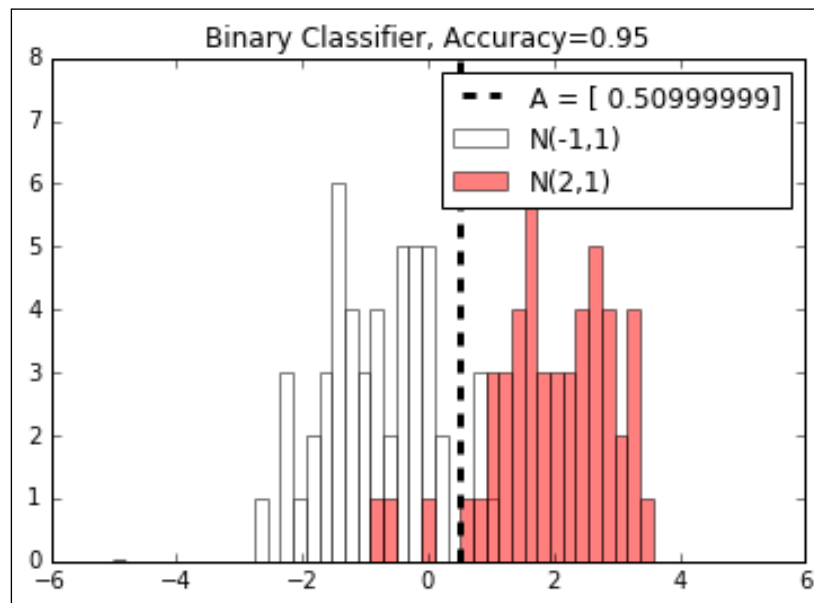
```
acc_value_test = sess.run(accuracy, feed_dict={x_data: [x_vals_
test], y_target: [y_vals_test]})
acc_value_train = sess.run(accuracy, feed_dict={x_data: [x_vals_
train], y_target: [y_vals_train]})
```

```
print('Accuracy on train set: ' + str(acc_value_train))
print('Accuracy on test set: ' + str(acc_value_test))
```

```
Accuracy on train set: 0.925
Accuracy on test set: 0.95
```

9. Sometimes we also want to graph the outcomes. We can easily graph the model and data here because it is one dimensional. Here is how to visualize the model and data with two separate histograms using matplotlib.

```
A_result = -sess.run(A)
bins = np.linspace(-5, 5, 50)
plt.hist(x_vals[0:50], bins, alpha=0.5, label='N(-1,1)',
color='white')
plt.hist(x_vals[50:100], bins[0:50], alpha=0.5, label='N(2,1)',
color='red')
plt.plot((A_result, A_result), (0, 8), 'k--', linewidth=3,
label='A = ' + str(np.round(A_result, 2)))
plt.legend(loc='upper right')
plt.title('Binary Classifier, Accuracy=' + str(np.round(acc_value,
2)))
plt.show()
```



Visualization of data and the end model, A. The two normal are centered at -1 and 2, making the theoretical best split at 0.5. Here the model found the best split very close to that number.

3

Getting Started

For this chapter we will cover basic recipes for understanding how Tensorflow works and how to access data for this book and additional resources.

- ▶ Using the Matrix Inverse Method
- ▶ Implementing a Decomposition Method
- ▶ Learning the Tensorflow Way of Regression
- ▶ Understanding Loss functions in Linear Regression
- ▶ Implementing Deming Regression
- ▶ Implementing Lasso and Ridge Regression
- ▶ Implementing Elastic Net Regression
- ▶ Implementing Logistic Regression

Introduction

Linear regression may be one of the most important algorithms in statistics, machine learning, and science in general. It's one of the most used algorithms and is very important to understand how to implement it and its various flavors. One of the advantages that linear regression has over many other algorithms is that it is very interpretable. We end up with a number for each feature that directly represents how that feature influences the target or dependent variable. In this chapter we will introduce how linear regression can be classically implemented, and then move on to how to best implement them in Tensorflow.

Using the Matrix Inverse Method

Getting ready

Linear regression can be represented as a set of matrix equations, say $y = X\beta$. Here we are interested in solving for the coefficients in matrix β . We have to be careful if our observation matrix (design matrix) X is not square. The solution to solving for β can be expressed as $\beta = (X^T X)^{-1} X^T y$. To show this is indeed the case we will generate 2 dimensional data, solve it in Tensorflow, and plot the result.

How to do it

1. First we load the necessary libraries, initialize the graph, and create the data.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

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

```
x_vals = np.linspace(0, 10, 100)
y_vals = x_vals + np.random.normal(0, 1, 100)
```

2. Next we create the matrices to use in the inverse method. We create the X matrix first, which will be a column of x-data and a column of ones. Then we create the y matrix from the y-data.

```
x_vals_column = np.transpose(np.matrix(x_vals))
ones_column = np.transpose(np.matrix(np.repeat(1, 100)))
A = np.column_stack((x_vals_column, ones_column))
b = np.transpose(np.matrix(y_vals))
```

3. We then turn our A and b matrix into tensors.

```
A_tensor = tf.constant(A)
b_tensor = tf.constant(b)
```

4. Now that we have our matrices setup, we can use Tensorflow to solve this via the matrix inverse method.

```
tA_A = tf.matmul(tf.transpose(A_tensor), A_tensor)
tA_A_inv = tf.matrix_inverse(tA_A)
product = tf.matmul(tA_A_inv, tf.transpose(A_tensor))
solution = tf.matmul(product, b_tensor)
solution_eval = sess.run(solution)
```

5. We now extract the coefficients from the solution, the slope and y-intercept.

```
slope = solution_eval[0][0]
y_intercept = solution_eval[1][0]
```

```

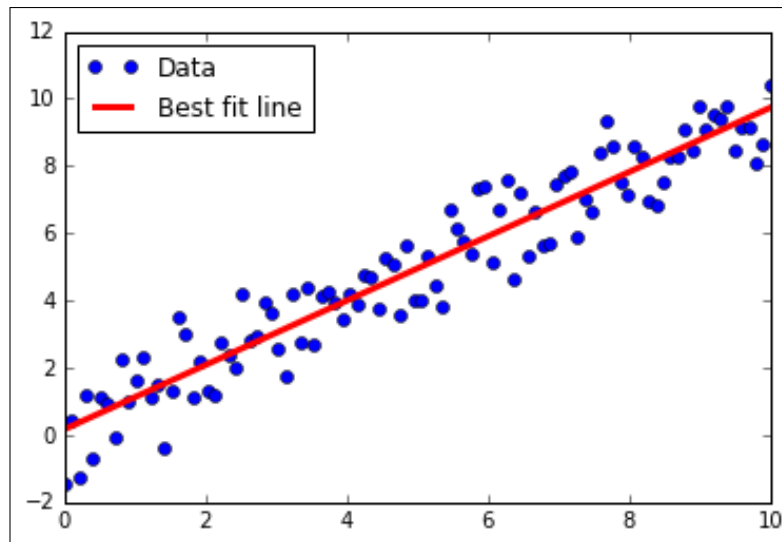
print('slope: ' + str(slope))
print('y_intercept: ' + str(y_intercept))

slope: 0.955707151739
y_intercept: 0.174366829314
6. Finally, we will create the best fit line values and plot the results.

best_fit = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)

plt.plot(x_vals, y_vals, 'o', label='Data')
plt.plot(x_vals, best_fit, 'r-', label='Best fit line', linewidth=3)
plt.legend(loc='upper left')
plt.show()

```



Data points and a best fit line obtained via matrix inverse method.

How it works

Unlike prior recipes, or most recipes in this book, the solution here is found exactly through matrix operations. Most Tensorflow algorithms that we will use are implemented via a training loop and take advantage of automatic back propagation to update model variables. Here, we illustrate the versatility of Tensorflow by implementing a direct solution to fitting a model to data.

Implementing a Decomposition Method

Getting ready

Implementing inverse methods in the prior recipe can be numerically inefficient in most cases, especially when the matrices get very large. Another approach is to decompose the matrix and perform matrix operations on the decompositions instead. One such approach is to use the built in Cholesky decomposition method in Tensorflow. One reason people are so interested in decomposing a matrix into more matrices is because the resulting matrices will have assured properties that allow us to use certain methods efficiently. The Cholesky decomposition decomposes a matrix into a lower and upper triangular matrix, say L and U such that these matrices are transposes of each other. For further information on the properties of this decomposition, there are many resources available that describe it and how to arrive at it. Here we will solve the system by writing it as $AX = b$. We will first solve for L and then solve for U to arrive at our coefficient matrix, A .

How to do it

1. We will set up the system the exact same way as the prior recipe. We will import libraries, initialize the graph, and create the data. Then we obtain our A matrix and b matrix the same as before.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.python.framework import ops
ops.reset_default_graph()

sess = tf.Session()

x_vals = np.linspace(0, 10, 100)
y_vals = x_vals + np.random.normal(0, 1, 100)

x_vals_column = np.transpose(np.matrix(x_vals))
ones_column = np.transpose(np.matrix(np.repeat(1, 100)))
A = np.column_stack((x_vals_column, ones_column))

b = np.transpose(np.matrix(y_vals))

A_tensor = tf.constant(A)
b_tensor = tf.constant(b)
```

Now we find the Cholesky decomposition of our square matrix, .



Note that the Tensorflow function, `cholesky()`, only returns the lower diagonal part of the decomposition. This is fine, as the upper diagonal matrix is just the transpose of the lower one.

```
tA_A = tf.matmul(tf.transpose(A_tensor), A_tensor)
L = tf.cholesky(tA_A)
```

```
tA_b = tf.matmul(tf.transpose(A_tensor), b)
sol1 = tf.matrix_solve(L, tA_b)
```

```
sol2 = tf.matrix_solve(tf.transpose(L), sol1)
```

2. Now that we have the solution, we extract the coefficients.

```
solution_eval = sess.run(sol2)

slope = solution_eval[0][0]
y_intercept = solution_eval[1][0]
```

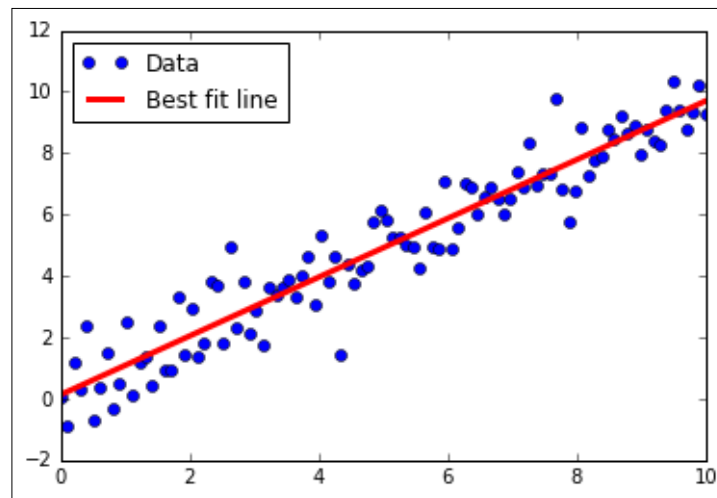
```
print('slope: ' + str(slope))
print('y_intercept: ' + str(y_intercept))
```

```
slope: 0.956117676145
y_intercept: 0.136575513864
```

3. Now we can plot the data and best fit line just as before.

```
best_fit = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)

plt.plot(x_vals, y_vals, 'o', label='Data')
plt.plot(x_vals, best_fit, 'r-', label='Best fit line',
linewidth=3)
plt.legend(loc='upper left')
plt.show()
```

Data points and best fit line obtained via Cholesky decomposition.

How it works

As you can see, we arrive at a very similar answer as the prior recipe. Keep in mind that this way of decomposing a matrix, then performing our operations on the pieces is sometime much more efficient and numerically stable.

The Tensorflow way of Linear Regression

Getting ready

In this recipe, we will loop through batches of data points and let Tensorflow update the slope and y-intercept. Instead of generated data, we will use the iris data set that is built in the library Scikit Learn. Specifically, we will find an optimal line through data points where the x-value is the petal width and the y-value is the sepal length. We choose these two because there appears to be a linear relationship between them, as we will see in the graphs at the end. We will also talk more about the effects of different loss functions in the next section, but for this recipe we will use the L2 loss function.

How to do it

1. We start by loading the necessary libraries, creating a graph, and loading the data.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
```

```
from tensorflow.python.framework import ops
ops.reset_default_graph()
```

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

```
iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])
```

2. We now declare our batch size, placeholders, and model variables.

```
batch_size = 25
```

```
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

```
A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
```

3. Next we will write the formula for the linear model, .

```
model_output = tf.add(tf.matmul(x_data, A), b)
```

4. Now we declare our L2 loss function (which includes the mean over the batch), initialize the variables, and declare our optimizer. Note that we chose 0.05 as our learning rate.

```
loss = tf.reduce_mean(tf.square(y_target - model_output))
```

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

```
my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(loss)
```

5. We can now loop through and train the model on randomly selected batches. We will run for 100 loops and print out the variable values and loss every 25 iterations. Note that here we are also saving the loss every iteration so that we can view it after.

```
loss_vec = []
for i in range(100):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
    rand_y})
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
    target: rand_y})
    loss_vec.append(temp_loss)
```

```
if (i+1)%25==0:
    print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + '
b = ' + str(sess.run(b)))
    print('Loss = ' + str(temp_loss))

Step #25 A = [[ 2.17270374]] b = [[ 2.85338426]]
Loss = 1.08116
Step #50 A = [[ 1.70683455]] b = [[ 3.59916329]]
Loss = 0.796941
Step #75 A = [[ 1.32762754]] b = [[ 4.08189011]]
Loss = 0.466912
Step #100 A = [[ 1.15968263]] b = [[ 4.38497639]]
Loss = 0.281003
```

6. Now we will extract the coefficients we found and create a best fit line to put in the graph.

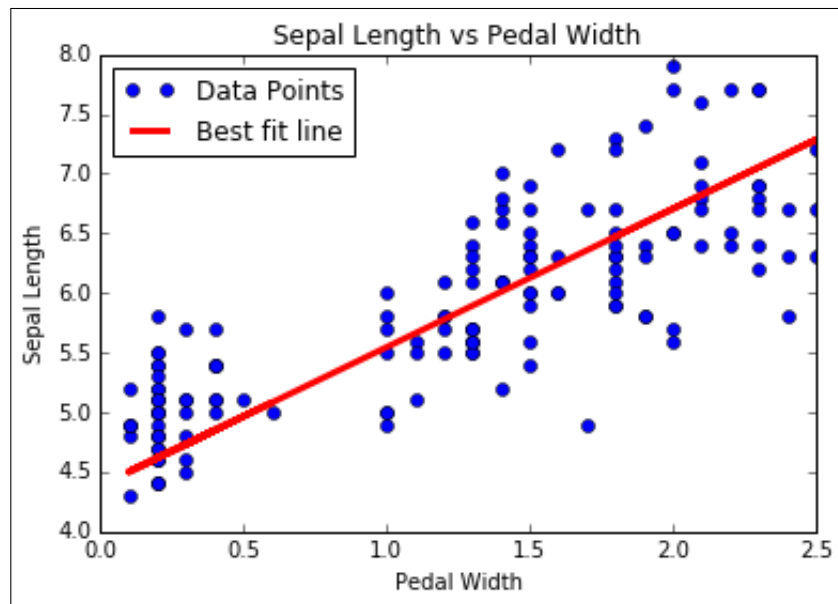
```
[slope] = sess.run(A)
[y_intercept] = sess.run(b)

best_fit = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)
```

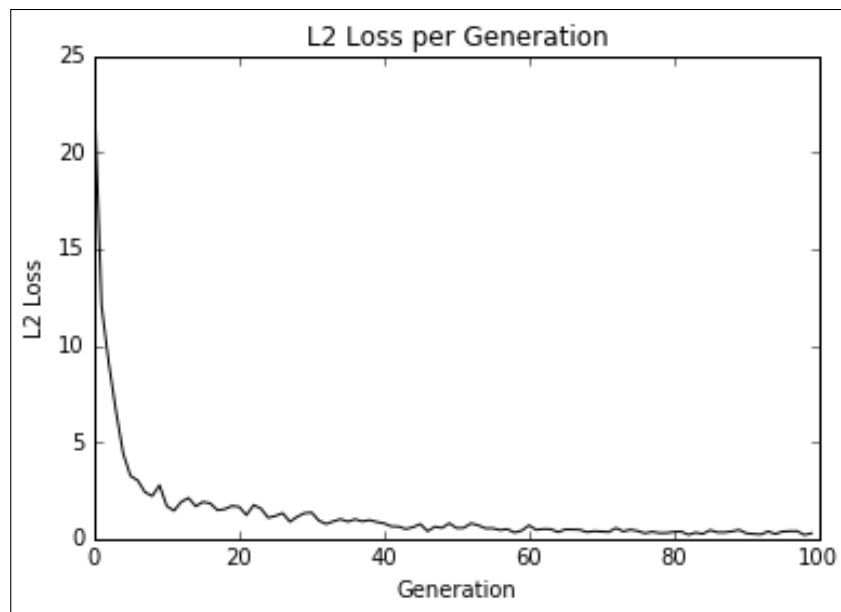
7. Here we will create two plots. The first will be the data with the found line overlayed. The second is the L2 loss function over the 100 iterations.

```
plt.plot(x_vals, y_vals, 'o', label='Data Points')
plt.plot(x_vals, best_fit, 'r-', label='Best fit line',
linewidth=3)
plt.legend(loc='upper left')
plt.title('Sepal Length vs Pedal Width')
plt.xlabel('Pedal Width')
plt.ylabel('Sepal Length')
plt.show()

plt.plot(loss_vec, 'k-')
plt.title('L2 Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('L2 Loss')
plt.show()
```



This is the data points from the iris data set (sepal length vs pedal width) overlaid with the optimal line fit found in Tensorflow with the specified algorithm.



This figure is the L2 loss of fitting the data with our algorithm. Note the jitter in the loss function, this can be decreased with a larger batch size or increased with a smaller batch size.

How it works

The optimal line found is not guaranteed to be the best fit line. Convergence to the best fit line depends on the number of iterations, batch size, learning rate, and the loss function. It is always good practice to observe the loss function over time as it can help us troubleshoot problems or parameter changes.

Understanding Loss Functions in Linear Regression

Getting ready

It is important to know the effect of loss functions in algorithm convergence. Here we will illustrate how the L1 and L2 loss functions affect convergence in linear regression. We will use the same iris data set as in the prior recipe, but we will change our loss functions and learning rates to see how convergence changes.

How to do it

1. The start of the program is unchanged from before until we get to our loss function. We load the necessary libraries, start a session, load the data, create placeholders, and define our variables and model. One thing to note is that we are pulling out our learning rate and model iterations. We are doing this because we want to show the effect of quickly changing these parameters.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets

sess = tf.Session()

iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])

batch_size = 25
learning_rate = 0.1 # Will not converge with learning rate at 0.4
iterations = 50

x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

```
A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))

model_output = tf.add(tf.matmul(x_data, A), b)
```

2. Our loss function will change to the L1 loss.

```
loss_l1 = tf.reduce_mean(tf.abs(y_target - model_output))
```



Note that we can change this back to the L2 loss by substituting in the formula `tf.reduce_mean(tf.square(y_target - model_output))`.

3. Now we resume by initializing the variables declaring our optimizer, and looping through the training part. Note that we are also saving our loss at every generation to measure the convergence.

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

my_opt_l1 = tf.train.GradientDescentOptimizer(learning_rate)
train_step_l1 = my_opt_l1.minimize(loss_l1)

loss_vec_l1 = []
for i in range(iterations):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step_l1, feed_dict={x_data: rand_x, y_target:
    rand_y})
    temp_loss_l1 = sess.run(loss_l1, feed_dict={x_data: rand_x,
    y_target: rand_y})
    loss_vec_l1.append(temp_loss_l1)
    if (i+1)%25==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + '
        b = ' + str(sess.run(b)))
```

4. If we copy the above code and change the loss function, we can plot both loss values with the below code.

```
plt.plot(loss_vec_l1, 'k-', label='L1 Loss')
plt.plot(loss_vec_l2, 'r--', label='L2 Loss')
plt.title('L1 and L2 Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('L1 Loss')
plt.legend(loc='upper right')
plt.show()
```

How it works

When choosing a loss function, we must also choose a corresponding learning rate that will work with our problem. Here, we will illustrate two situations, one in which L2 is preferred and one in which L1 is preferred.

If our learning rate is small, our convergence will take more time. But if our learning rate is too large, we will have issues with our algorithm never converging. Here is a plot of the loss function of the L1 and L2 loss for the iris linear regression problem when the learning rate is 0.05.

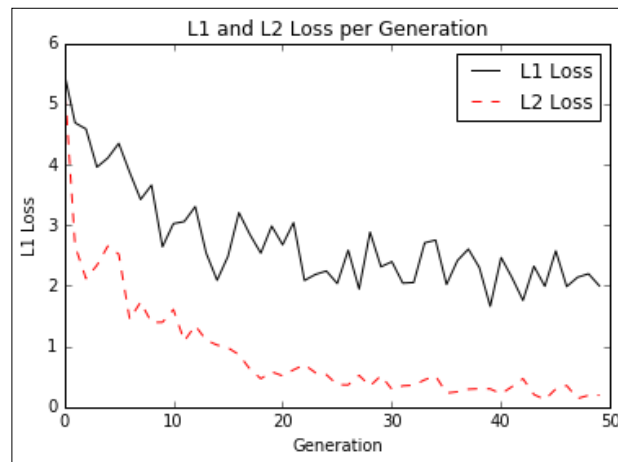
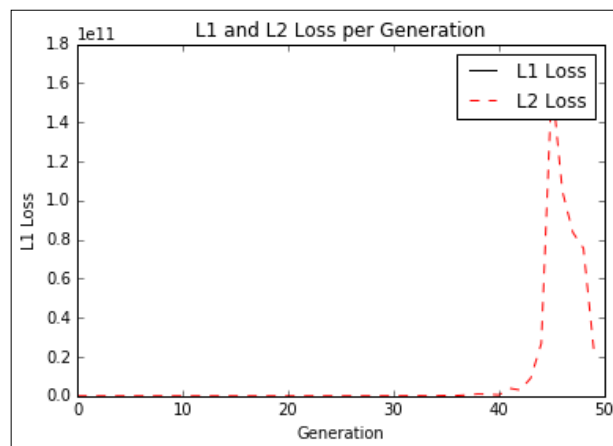


Figure illustrating L1 and L2 loss with a learning rate of 0.05 for the iris linear regression problem.

With a learning rate of 0.05, it would appear that L2 loss is preferred, as it converges to a lower loss on the data. Here is a graph of the loss functions when we increase the learning rate to 0.4.

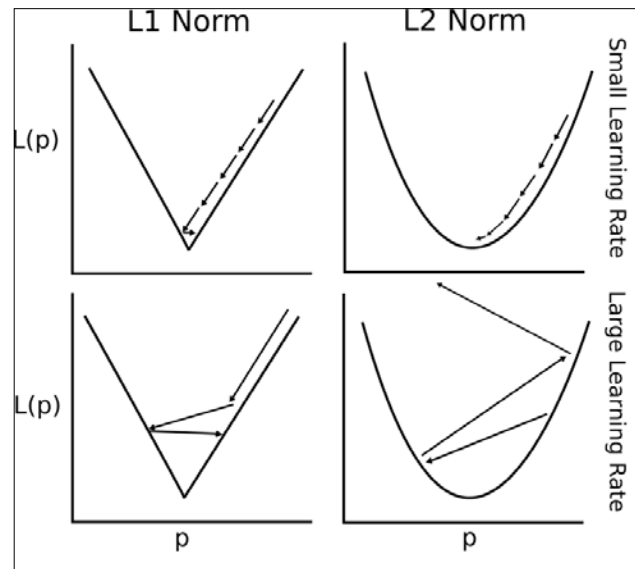


This figure shows the L1 and L2 loss on the iris linear regression problem with a learning rate of 0.4. Note that the L1 loss isn't visible because of the high scale of the y-axis.

Here, we can see that the large learning rate can overshoot in the L2 norm, whereas the L1 norm converges.

There's more

To understand what is happening, we should look at how a large learning rate and small learning rate act on L1 and L2 norms. To visualize this, we look at a one dimensional representation of learning steps on both norms below.

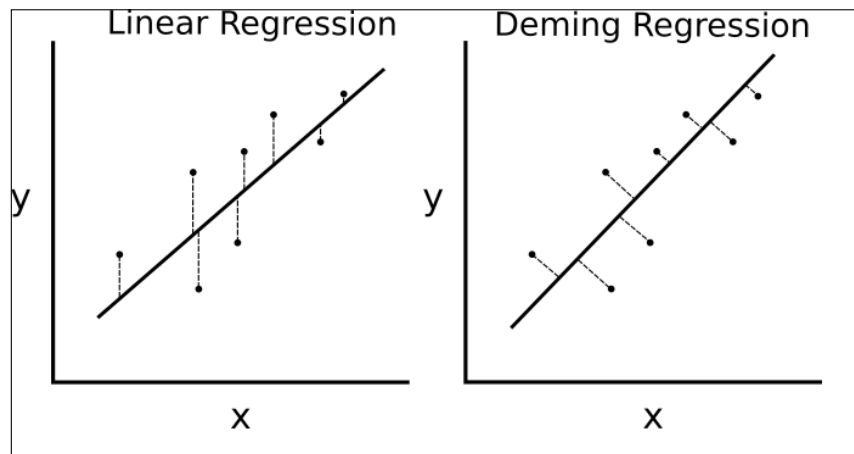


This figure illustrates what can happen with the L1 and L2 norm with larger and smaller learning rates.

Implementing Deming Regression

Getting ready

If least squares linear regression minimizes the vertical distance to the line, Deming regression minimizes the total distance to the line. This type of regression minimizes the error in the y values and the x values. See the below figure for a comparison.



Here we illustrate the difference between regular linear regression and Deming regression. Linear regression on the left minimizes the vertical distance to the line, and Deming regression minimizes the total distance to the line.

To implement Deming regression, we have to modify the loss function. The loss function in regular linear regression minimizes the vertical distance. Here, we want to minimize the total distance. Given a slope and intercept of a line, the perpendicular distance to a point is a known geometric formula. We just have to substitute this formula in and tell Tensorflow to minimize it.

How to do it

1. Everything stays the same except when we get to the loss function. We begin by loading the libraries, starting a session, loading the data, declaring the batch size, creating the placeholders, variables, and model output.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets

sess = tf.Session()
```

```

iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])

batch_size = 50

x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))

model_output = tf.add(tf.matmul(x_data, A), b)

```

2. The loss function is a geometric formula that comprises of a numerator and denominator. For clarity we will write these out separately. Given a line, and a point, the perpendicular distance between the two can be written as the following.

```

demming_numerator = tf.abs(tf.sub(y_target, tf.add(tf.matmul(x_
data, A), b)))
demming_denominator = tf.sqrt(tf.add(tf.square(A),1))
loss = tf.reduce_mean(tf.truediv(demming_numerator, demming_
denominator))

```

3. We now initialize our variables, declare our optimizer, and loop through the training set to arrive at our parameters.

```

init = tf.initialize_all_variables()
sess.run(init)

my_opt = tf.train.GradientDescentOptimizer(0.1)
train_step = my_opt.minimize(loss)

loss_vec = []
for i in range(250):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
    loss_vec.append(temp_loss)
    if (i+1)%50==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + '
b = ' + str(sess.run(b)))
        print('Loss = ' + str(temp_loss))

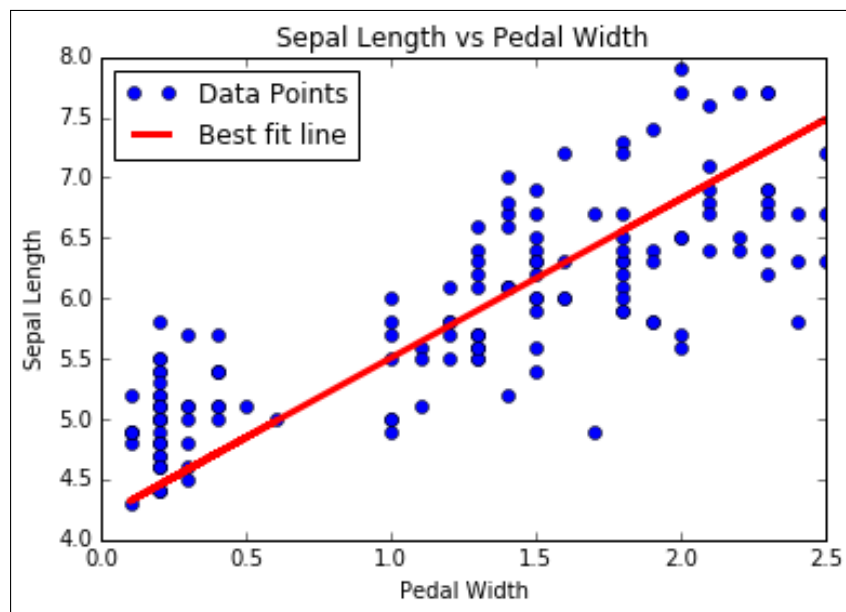
```

4. And we can plot the output with the following code.

```
[slope] = sess.run(A)
[y_intercept] = sess.run(b)

best_fit = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)

plt.plot(x_vals, y_vals, 'o', label='Data Points')
plt.plot(x_vals, best_fit, 'r-', label='Best fit line',
linewidth=3)
plt.legend(loc='upper left')
plt.title('Sepal Length vs Pedal Width')
plt.xlabel('Pedal Width')
plt.ylabel('Sepal Length')
plt.show()
```



The graph depicting the solution to Deming regression on the iris data set.

How it works

The recipe here for Deming regression is almost identical to regular linear regression. The key difference here is how we measure the loss between the predictions and the data points. Instead of a vertical loss, we have a perpendicular loss (or total loss) with the y values and x values.



Note that the type of Deming regression implemented here is called total regression. Total regression is when we assume the error in the x and y values are similar. We can also scale the x and y axes in the distance calculation by the difference in the errors according to our beliefs.

Implementing Lasso and Ridge Regression

Getting ready

Lasso and ridge regression are very similar to regular linear regression, except we are adding regularization terms to limit the slopes (or partial slopes) in the formula. There may be multiple reasons for this, but a common one is that we wish to restrict the features that have an impact on the dependent variable. This can be accomplished by adding a term to the loss function that depends on the value of our slope, A .

For lasso regression, we must add a term that greatly increases our loss function if the slope, A , gets above a certain value. We could use Tensorflow's logical operations, but they do not have a gradient associated with them. Instead, we will use a continuous approximation to a step function, called the continuous heavystep function, that is scaled up and over to the regularization cut off we choose. We will show how to do Lasso regression below.

For ridge regression, we just add a term to the L2 norm which is the scaled L2 norm of the slope coefficient. This modification is simple and is shown below the 'There's More' section at the bottom of this recipe.

How to do it

1. We will use the iris data set again and setup our script the same way as before. We first load the libraries, start a session, load the data, declare the batch size, create the placeholders, variables, and model output.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
from tensorflow.python.framework import ops
```

```
ops.reset_default_graph()

sess = tf.Session()

iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])

batch_size = 50

x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))

model_output = tf.add(tf.matmul(x_data, A), b)
```

2. We add the loss function which is a modified continuous heavyside step function. We also set the cutoff for lasso regression to be at 0.9. This means that we want to restrict the slope coefficient to be less than 0.9.

```
lasso_param = tf.constant(0.9)
heavyside_step = tf.truediv(1., tf.add(1., tf.exp(tf.mul(-100.,
tf.sub(A, lasso_param)))))
regularization_param = tf.mul(heavyside_step, 99.)
loss = tf.add(tf.reduce_mean(tf.square(y_target - model_output)),
regularization_param)
```

We now initialize our variables and declare our optimizer.

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

my_opt = tf.train.GradientDescentOptimizer(0.001)
train_step = my_opt.minimize(loss)
```

3. We will run the training loop a fair bit longer because it can take a while to converge. We can see that the slope coefficient is less than 0.9.

```
loss_vec = []
for i in range(1500):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})
```

```

temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
loss_vec.append(temp_loss[0])
if (i+1)%300==0:
    print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + '
b = ' + str(sess.run(b)))
    print('Loss = ' + str(temp_loss))

Step #300 A = [[ 0.82512331]] b = [[ 2.30319238]]
Loss = [[ 6.84168959]]
Step #600 A = [[ 0.8200165]] b = [[ 3.45292258]]
Loss = [[ 2.02759886]]
Step #900 A = [[ 0.81428504]] b = [[ 4.08901262]]
Loss = [[ 0.49081498]]
Step #1200 A = [[ 0.80919558]] b = [[ 4.43668795]]
Loss = [[ 0.40478843]]
Step #1500 A = [[ 0.80433637]] b = [[ 4.6360755]]
Loss = [[ 0.23839757]]

```

How it works

We implement lasso regression by adding a continuous heavyside step function to the loss function of linear regression. Because of the steepness of the step function, we have to be careful with step size. Too big of a step size and it will not converge. For ridge regression, see the necessary change in the next section.

There's more

For ridge regression, we change the loss function to look like the below code.

```

ridge_param = tf.constant(1.)
ridge_loss = tf.reduce_mean(tf.square(A))
loss = tf.expand_dims(tf.add(tf.reduce_mean(tf.square(y_target -
model_output)), tf.mul(ridge_param, ridge_loss)), 0)

```

Implementing Elastic Net Regression

Getting ready

Elastic net regression is a type of regression that combines lasso regression with ridge regression by adding a L1 and L2 regularization term to the loss function. Since adding that should be straight forward after the prior two recipes, we will implement this in multiple linear regression on the iris data set, instead of sticking to the 2 dimensional data as before. We will use pedal length, pedal width, and sepal width to predict sepal length.

How to do it

1. First we load the necessary libraries and initialize a graph.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
```

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

2. Now we will load the data. This time, each element of x data will be a list of three values instead of one.

```
iris = datasets.load_iris()
x_vals = np.array([[x[1], x[2], x[3]] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])
```

3. Next we declare the batch size, placeholders, variables, and model output. The only difference here is that we change the size specifications of the x data placeholder to take three values instead of one.

```
batch_size = 50
```

```
x_data = tf.placeholder(shape=[None, 3], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

```
A = tf.Variable(tf.random_normal(shape=[3,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
```

```
model_output = tf.add(tf.matmul(x_data, A), b)
```

4. For elastic net, the loss function has the l1 and l2 norms of the partial slopes. We create these terms and then add them into the loss function.

```
elastic_param1 = tf.constant(1.)
elastic_param2 = tf.constant(1.)
l1_a_loss = tf.reduce_mean(tf.abs(A))
l2_a_loss = tf.reduce_mean(tf.square(A))
e1_term = tf.mul(elastic_param1, l1_a_loss)
e2_term = tf.mul(elastic_param2, l2_a_loss)
loss = tf.expand_dims(tf.add(tf.add(tf.reduce_mean(tf.square(y_
target - model_output)), e1_term), e2_term), 0)
```

5. Now we can initialize the variables, declare our optimizer, and run the training loop and fit our coefficients.

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

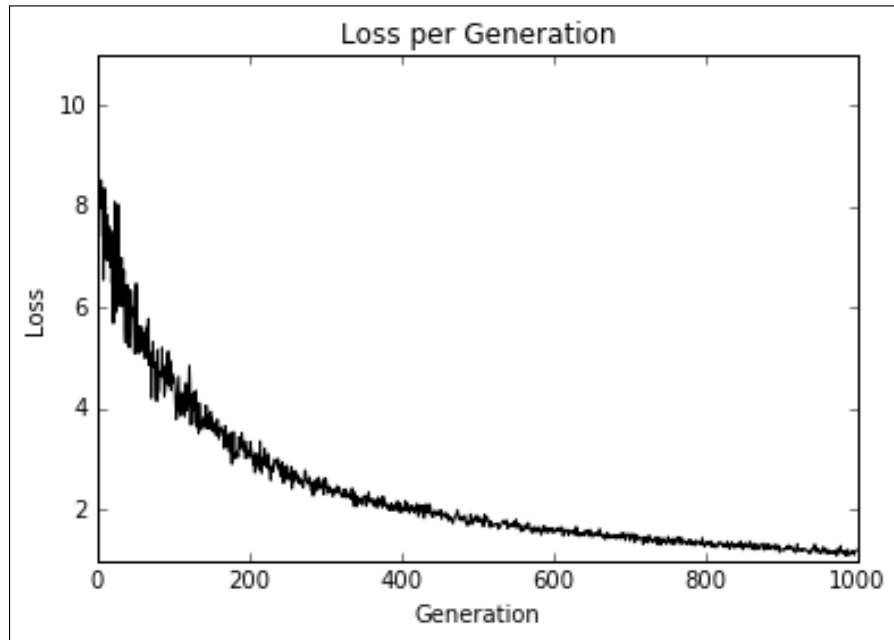
my_opt = tf.train.GradientDescentOptimizer(0.001)
train_step = my_opt.minimize(loss)

loss_vec = []
for i in range(1000):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
    loss_vec.append(temp_loss[0])
    if (i+1)%250==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + '
b = ' + str(sess.run(b)))
        print('Loss = ' + str(temp_loss))

Step #250 A = [[ 0.42095602]
[ 0.1055888 ]
[ 1.77064979]] b = [[ 1.76164341]]
Loss = [ 2.87764359]
Step #500 A = [[ 0.62762028]
[ 0.06065864]
[ 1.36294949]] b = [[ 1.87629771]]
Loss = [ 1.8032167]
Step #750 A = [[ 0.67953539]
[ 0.102514 ]
[ 1.06914485]] b = [[ 1.95604002]]
Loss = [ 1.33256555]
Step #1000 A = [[ 0.6777274 ]
[ 0.16535147]
[ 0.8403284 ]] b = [[ 2.02246833]]
Loss = [ 1.21458709]
```


6. Now we can observe the loss over the training iterations to be sure that it converged.

```
plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```



Elastic net regression loss plotted over the 1000 training iterations

How it works

Elastic net regression is implemented here as well as multiple linear regression. We can see that with these regularization terms in the loss function the convergence is slower than in prior sections. Regularization is as simple as adding in the appropriate terms in the loss functions.

Implementing Logistic Regression

Getting ready

Logistic regression is a way to turn linear regression into a binary classification. This is accomplished by transforming the linear output in a sigmoid function that scales the output between zero and one. The target is a zero or one which indicates whether or not a data point is in one class or another. Since we are predicting a number between zero or one, the prediction is classified into class value '1' if the prediction is above a specified cut off value and class '0' otherwise. For the purpose of this example, we will specify that cut off to be 0.5, which will make the classification as simple as rounding the output.

The data we will use for this example will be the low birth weight data that is obtained through the University of Massachusetts Amherst statistical dataset repository (<https://www.umass.edu/statdata/statdata/>). We will be predicting low birth weight from several other factors.

How to do it

1. We start by loading the libraries, including the request library, because we will access the low birth weight data through a hyperlink. We also will initiate a session.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import requests
from sklearn import datasets
from sklearn.preprocessing import normalize
from tensorflow.python.framework import ops
ops.reset_default_graph()
```

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

2. Next we will load the data through the request module and specify which features we want to use. We have to be specific because one feature is the actual birth weight and we don't want to use this to predict if the birthweight is greater or less than a specific amount. We also don't want to use the ID column as a predictor either.

```
birthdata_url = 'https://www.umass.edu/statdata/statdata/data/lowbwt.dat'
birth_file = requests.get(birthdata_url)
birth_data = birth_file.text.split('\r\n')[5:]
birth_header = [x for x in birth_data[0].split(' ') if len(x)>=1]
birth_data = [[float(x) for x in y.split(' ') if len(x)>=1] for y in birth_data[1:] if len(y)>=1]
```

```
y_vals = np.array([x[1] for x in birth_data])

x_vals = np.array([x[2:9] for x in birth_data])
```

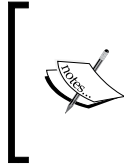
3. First we split the data set into test and train sets.

```
train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

4. Logistic regression convergence works better when the features are scaled between 0 and 1 (min-max scaling). So next we will scale each feature.

```
def normalize_cols(m):
    col_max = m.max(axis=0)
    col_min = m.min(axis=0)
    return (m-col_min) / (col_max - col_min)

x_vals_train = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test = np.nan_to_num(normalize_cols(x_vals_test))
```



Note that we split the data set into train and test before we scaled the data set. This is an important distinction to make. We want to make sure that the training set does not influence the test set at all. If we scaled the whole set before splitting, then we cannot guarantee that they don't influence each other.

5. Next we declare the batch size, placeholders, variables, and the logistic model. We do not wrap the output in a sigmoid because that operation is built into the loss function.

```
batch_size = 25

x_data = tf.placeholder(shape=[None, 7], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

A = tf.Variable(tf.random_normal(shape=[7,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))

model_output = tf.add(tf.matmul(x_data, A), b)
```

6. Now we declare our loss function, which has the sigmoid function, initialize our variables and declare our optimizer function.

```
loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(model_output, y_target))
```

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

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)
```

7. Along with recording the loss function, we will also want to record the classification accuracy on the training and test set. So we will create a prediction function that returns the accuracy for any size batch.

```
prediction = tf.round(tf.sigmoid(model_output))
```

```
predictions_correct = tf.cast(tf.equal(prediction, y_target),
tf.float32)
```

```
accuracy = tf.reduce_mean(predictions_correct)
```

Now we can start our training loop and recording the loss and accuracies.

```
loss_vec = []
```

```
train_acc = []
```

```
test_acc = []
```

```
for i in range(1500):
```

```
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
```

```
    rand_x = x_vals_train[rand_index]
```

```
    rand_y = np.transpose([y_vals_train[rand_index]])
```

```
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})
```

```
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
```

```
    loss_vec.append(temp_loss)
```

```
    temp_acc_train = sess.run(accuracy, feed_dict={x_data: x_vals_
train, y_target: np.transpose([y_vals_train])})
```

```
    train_acc.append(temp_acc_train)
```

```
    temp_acc_test = sess.run(accuracy, feed_dict={x_data: x_vals_
test, y_target: np.transpose([y_vals_test])})
```

```
    test_acc.append(temp_acc_test)
```

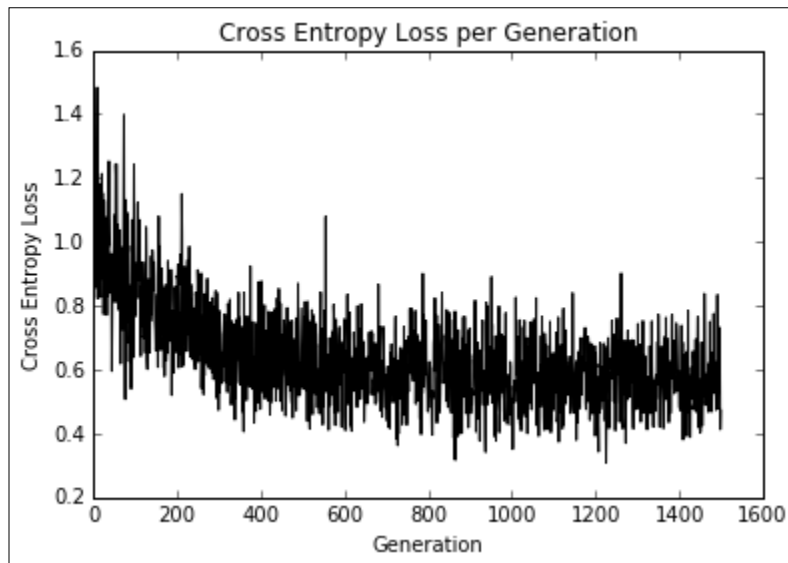
8. Here is the code to look at the plots of loss and accuracies.

```
plt.plot(loss_vec, 'k-')
plt.title('Cross Entropy Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Cross Entropy Loss')
plt.show()

plt.plot(train_acc, 'k-', label='Train Set Accuracy')
plt.plot(test_acc, 'r--', label='Test Set Accuracy')
plt.title('Train and Test Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```

How it works

Here is the loss over the iterations and train and test set accuracies. Since the data set is only 189 observations, the train and test accuracy plots will change due to the random splitting of the data set.



The above figure is about Cross entropy loss plotted over the course of 1500 iterations



Test and Train set accuracy plotted over 1500 generations.

4

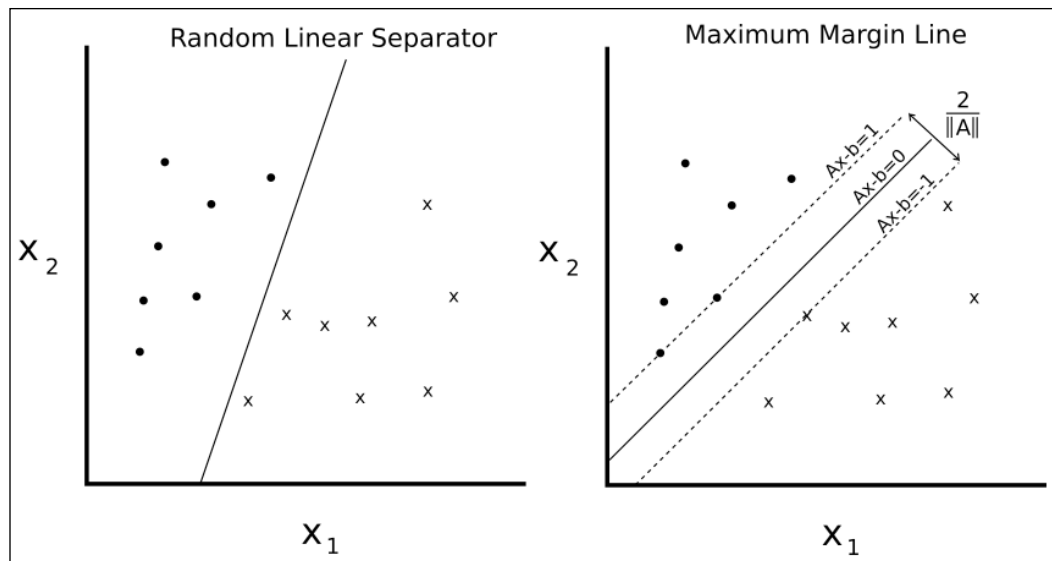
Support Vector Machines

This chapter will cover some important recipes with how to use, implement, and evaluate support vector machines (SVM) in Tensorflow.

- ▶ Introduction
- ▶ Working with a Linear SVM
- ▶ Reduction to Linear Regression
- ▶ Working with Kernels in Tensorflow
- ▶ Implementing a Non-Linear SVM
- ▶ Implementing a Multi-Class SVM

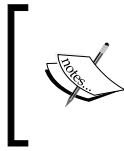
Introduction

- ▶ Support vector machines are a method of binary classification. The basic idea is to find a linear separating line (or hyperplane) between the two classes. We first assume that the binary class targets are -1 or 1, instead of the prior 0 or 1 targets. Since there may be many lines that separate two classes, we define the best linear separator that maximizes the distance between both classes.



Given two separable classes, 'o' and 'x', we wish to find the equation for the linear separator between the two. The left shows that there are many lines that separate the two classes. The right shows the unique maximum margin line. The margin width is given by $2/\|A\|$. This line is found by minimizing the L2 norm of A .

- ▶ Since we can write such a hyperplane as
- ▶ Here, A is a vector of our partial slopes and x is a vector of inputs. The width of the maximum margin can be shown to be 2 divided by the L2 norm of A . There are many proofs out there of this fact, but for a geometric idea, solving for the perpendicular distance from a 2d point to a line may provide motivation for moving forward.
- ▶ For linearly separable binary class data, to maximize the margin, we minimize the L2 norm of A . We must also subject this minimum to the constraint
- ▶ The above constraint assures us that all the points from the corresponding classes are on the same side of the separating line.
- ▶ Since not all data sets are linearly separable, we can introduce a loss function for points that cross the margin lines. For n data points, we introduce what is called the soft margin loss function below.



Note that the product is always greater than 1 if the point is on the correct side of the margin. This makes the left term of the loss function equal to zero, and the only influence on the loss function is the size of the margin.

- ▶ The above loss function will seek a linearly separable line, but will allow for points crossing the margin line. This can be a hard or soft allowance, depending on the value of λ . Larger values of λ result in more emphasis on widening the margin and smaller values of λ result in the model acting more like a hard margin while allowing data points to cross the margin if need be.
- ▶ In this chapter, we will setup a soft margin SVM and show how to extend it to nonlinear cases and multiple classes.

Working with a Linear SVM

For this example, we will create a linear separator from the iris data set. We know from prior chapters that the sepal length and pedal width create a linear separable binary data set for predicting if a flower is *I. setosa* or not.

Getting ready

To implement a soft separable SVM in Tensorflow, we will implement the specific loss function

Here, A is the vector of partial slopes, b is the intercept, x is a vector of inputs, y is the actual class, (-1 or 1) and λ is the soft separability regularization parameter.

How to do it

1. We start by loading the necessary libraries. This will include the scikit learn dataset library for access to the iris data set.
2. Next we start a graph session and load the data as we need it. Remember that we are loading the first and fourth variable in the iris data set as they are the sepal length and sepal width. The target variable we will be loading will take on the value 1 for *I. setosa* and -1 otherwise.

```
sess = tf.Session()

iris = datasets.load_iris()
```

```
x_vals = np.array([[x[0], x[3]] for x in iris.data])
y_vals = np.array([1 if y==0 else -1 for y in iris.target])
```

3. We should now split the data set into train and test sets. We will evaluate the accuracy on both the training and test sets. Since we know this data set are linearly separable, we should expect to get one hundred percent accuracy on both sets.

```
train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

4. Next we set our batch size, placeholders, and model variables. It is important to mention that with this SVM algorithm, we want very large batch sizes to help with convergence. We can imagine that with very small batch sizes, the maximum margin line will jump around slightly. Ideally we would also slowly decrease the learning rate as well, but this will suffice for now. Also, the A variable will take on shape 2x1 because we have two predictor variables, sepal length and pedal width.

```
batch_size = 100
```

```
x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

```
A = tf.Variable(tf.random_normal(shape=[2,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
```

5. We now declare our model output. For correctly classified points, this will return numbers that are greater than or equal to 1 if the target is I. setosa and less than or equal to -1 otherwise.

```
model_output = tf.sub(tf.matmul(x_data, A), b)
```

6. Next we will put together and declare the necessary components to the maximum margin loss. First we will declare a function that will calculate the L2 norm of a vector. Then we add the margin parameter, . We then declare our classification loss and add together the two terms.

```
l2_norm = tf.reduce_sum(tf.square(A))
```

```
alpha = tf.constant([0.1])
```

```
classification_term = tf.reduce_mean(tf.maximum(0., tf.sub(1.,
tf.mul(model_output, y_target))))
```

```
loss = tf.add(classification_term, tf.mul(alpha, l2_norm))
```

7. Now we declare our prediction and accuracy functions so that we can evaluate the accuracy on both the training and test sets.

```
prediction = tf.sign(model_output)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, y_target),
tf.float32))
```

8. Here we will declare our optimizer and initialize our model variables.

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)
```

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

9. We now can start our training loop, keeping in mind that we want to record our loss and training accuracy on both the training and test set.

```
loss_vec = []
train_accuracy = []
test_accuracy = []
for i in range(500):
    rand_index = np.random.choice(len(x_vals_train), size=batch_
size)
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
    loss_vec.append(temp_loss)

    train_acc_temp = sess.run(accuracy, feed_dict={x_data: x_vals_
train, y_target: np.transpose([y_vals_train])})
    train_accuracy.append(train_acc_temp)

    test_acc_temp = sess.run(accuracy, feed_dict={x_data: x_vals_
test, y_target: np.transpose([y_vals_test])})
    test_accuracy.append(test_acc_temp)

    if (i+1)%100==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + '
b = ' + str(sess.run(b)))
        print('Loss = ' + str(temp_loss))

Step #100 A = [[-0.10763293]
[-0.65735245]] b = [[-0.68752676]]
```

```
Loss = [ 0.48756418]
Step #200 A = [[-0.0650763 ]
 [-0.89443302]] b = [[-0.73912662]]
Loss = [ 0.38910741]
Step #300 A = [[-0.02090022]
 [-1.12334013]] b = [[-0.79332656]]
Loss = [ 0.28621092]
Step #400 A = [[ 0.03189624]
 [-1.34912157]] b = [[-0.8507266]]
Loss = [ 0.22397576]
Step #500 A = [[ 0.05958777]
 [-1.55989814]] b = [[-0.9000265]]
Loss = [ 0.20492229]
```

10. In order to plot the outputs, we have to extract the coefficients and separate the x values into I. setosa and non I. setosa.

```
[[a1], [a2]] = sess.run(A)
[[b]] = sess.run(b)
slope = -a2/a1
y_intercept = b/a1

x1_vals = [d[1] for d in x_vals]

best_fit = []
for i in x1_vals:
    best_fit.append(slope*i+y_intercept)

setosa_x = [d[1] for i,d in enumerate(x_vals) if y_vals[i]==1]
setosa_y = [d[0] for i,d in enumerate(x_vals) if y_vals[i]==1]
not_setosa_x = [d[1] for i,d in enumerate(x_vals) if y_
vals[i]==-1]
not_setosa_y = [d[0] for i,d in enumerate(x_vals) if y_
vals[i]==-1]
```

11. The following is the code to plot the data with the linear separator, accuracies, and loss.

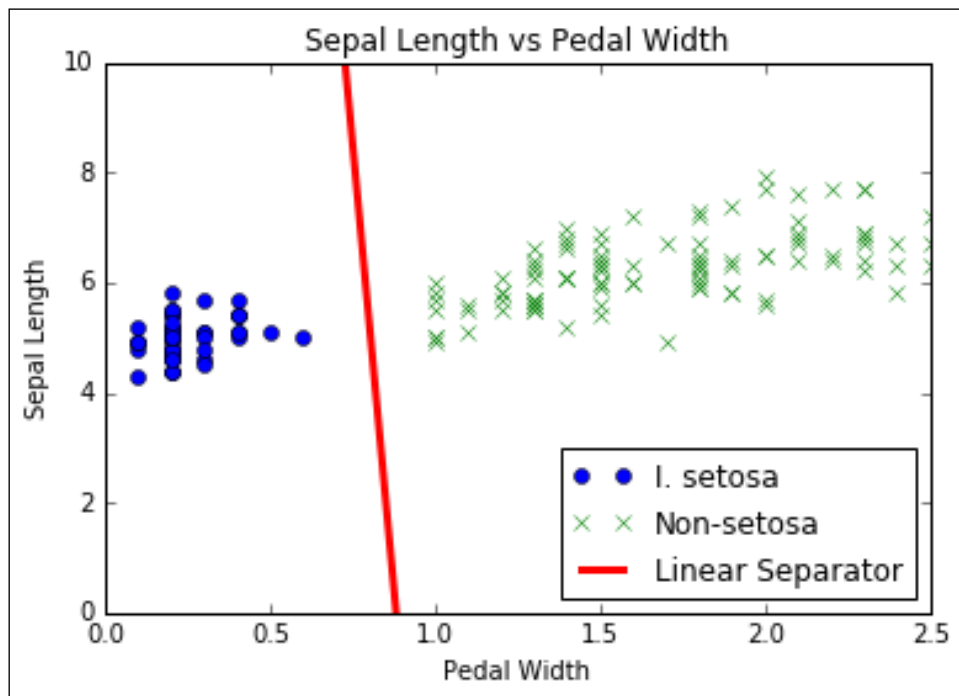
```
plt.plot(setosa_x, setosa_y, 'o', label='I. setosa')
plt.plot(not_setosa_x, not_setosa_y, 'x', label='Non-setosa')
plt.plot(x1_vals, best_fit, 'r-', label='Linear Separator',
linewidth=3)
plt.ylim([0, 10])
plt.legend(loc='lower right')
plt.title('Sepal Length vs Pedal Width')
plt.xlabel('Pedal Width')
plt.ylabel('Sepal Length')
plt.show()
```

```
plt.plot(train_accuracy, 'k-', label='Training Accuracy')
plt.plot(test_accuracy, 'r--', label='Test Accuracy')
plt.title('Train and Test Set Accuracies')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

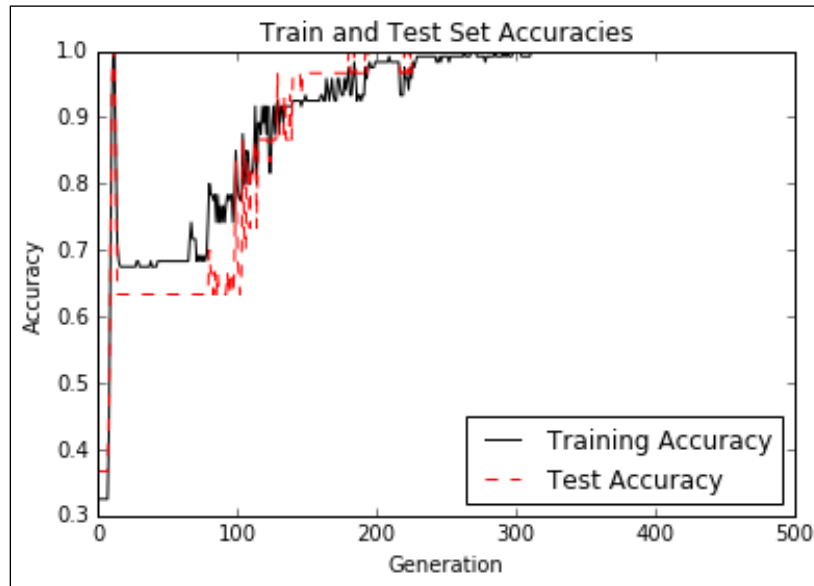
plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```



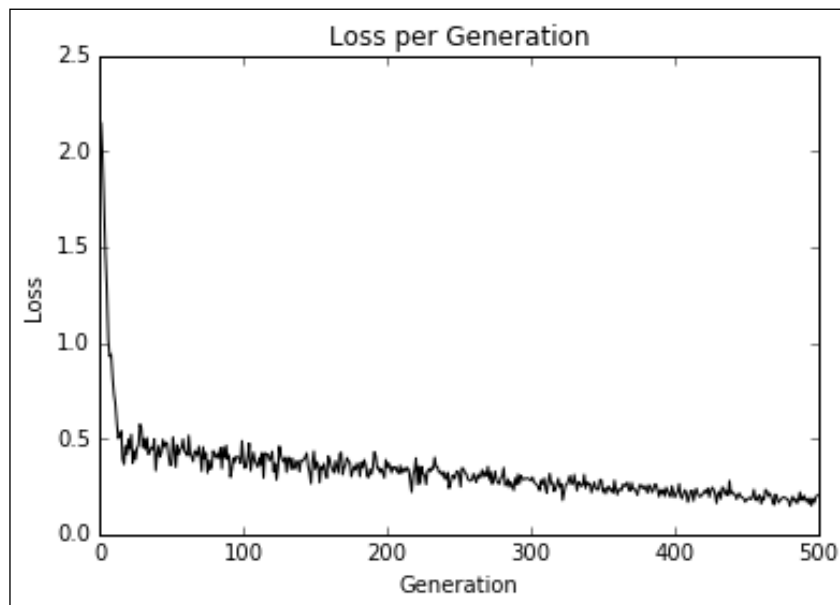
Note: Using Tensorflow in this manner to implement the SVD algorithm may result in slightly different outcomes each run. The reasons for this include the random train/test set splitting, and the selection of different batches of points each training run. Also it would be ideal to also slowly lower the learning rate after each generation.



Final linear SVM fit with the two classes plotted.



Test and Train set accuracy over generations. We do get 100% accuracy because the two classes are linearly separable.



Plot of the maximum margin loss over 500 generations.

How it works

In this recipe, we have shown that implementing a linear SVD model is possible with using the maximum margin loss function.

Reduction to Linear Regression

Getting ready

The same 'maximum margin' concept can be applied toward fitting linear regression. Instead of maximizing the margin that separates the classes, we can think about maximizing the margin that contains the most (x,y) points. To illustrate this, we will use the same iris data set, and show that we can use this concept to fit a line between sepal length and pedal width.

The corresponding loss function will be similar to . Here, γ is half of the width of the margin, which makes the loss equal to zero if a point lies in this region.

How to do it

1. First we load the necessary libraries, start a graph, and load the iris data set. After, we will split the dataset into train and test sets to visualize the loss on both.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets

sess = tf.Session()

iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])

train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```


2. Let's declare our batch size, placeholders, variables, and create our linear model.

```
batch_size = 50

x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))

model_output = tf.add(tf.matmul(x_data, A), b)
```

3. Now we declare our loss function. The loss function as described above, is implemented below with .

```
epsilon = tf.constant([0.5])
loss = tf.reduce_mean(tf.maximum(0., tf.sub(tf.abs(tf.sub(model_
output, y_target)), epsilon)))
```

4. We create an optimizer and initialize our variables next.

```
my_opt = tf.train.GradientDescentOptimizer(0.075)
train_step = my_opt.minimize(loss)
```

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

5. Now we iterate through 200 training generations and save the training and test loss for plotting later.

```
train_loss = []
test_loss = []
for i in range(200):
    rand_index = np.random.choice(len(x_vals_train), size=batch_
size)
    rand_x = np.transpose([x_vals_train[rand_index]])
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})

    temp_train_loss = sess.run(loss, feed_dict={x_data:
np.transpose([x_vals_train]), y_target: np.transpose([y_vals_
train])})
    train_loss.append(temp_train_loss)

    temp_test_loss = sess.run(loss, feed_dict={x_data:
np.transpose([x_vals_test]), y_target: np.transpose([y_vals_
test])})
    test_loss.append(temp_test_loss)
```

```

    if (i+1)%50==0:
        print('-----')
        print('Generation: ' + str(i))
        print('A = ' + str(sess.run(A)) + ' b = ' + str(sess.
run(b)))
        print('Train Loss = ' + str(temp_train_loss))
        print('Test Loss = ' + str(temp_test_loss))

```

```

-----
Generation: 50
A = [[ 2.20651722]] b = [[ 2.71290684]]
Train Loss = 0.609453
Test Loss = 0.460152
-----
Generation: 100
A = [[ 1.6440177]] b = [[ 3.75240564]]
Train Loss = 0.242519
Test Loss = 0.208901
-----
Generation: 150
A = [[ 1.27711761]] b = [[ 4.3149066]]
Train Loss = 0.108192
Test Loss = 0.119284
-----
Generation: 200
A = [[ 1.05271816]] b = [[ 4.53690529]]
Train Loss = 0.079957
Test Loss = 0.107551

```

6. We can now extract the coefficients we found, and get values for the best fit line. For plotting purposes, we will also get values for the margins as well.

```

[[slope]] = sess.run(A)
[[y_intercept]] = sess.run(b)
[width] = sess.run(epsilon)

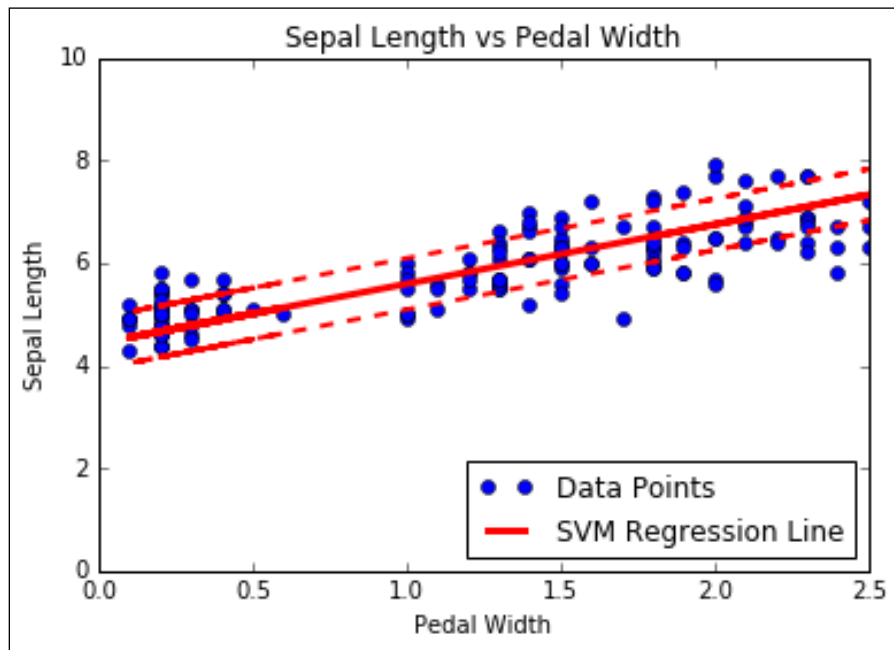
best_fit = []
best_fit_upper = []
best_fit_lower = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)
    best_fit_upper.append(slope*i+y_intercept+width)
    best_fit_lower.append(slope*i+y_intercept-width)

```

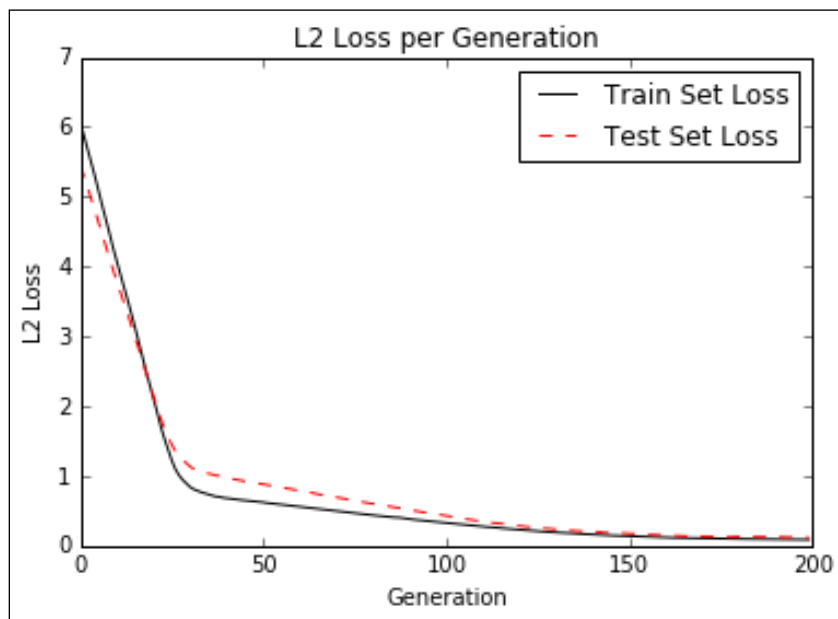
7. Finally, here is the code to plot the data with the fitted line and the train-test loss.

```
plt.plot(x_vals, y_vals, 'o', label='Data Points')
plt.plot(x_vals, best_fit, 'r-', label='SVM Regression Line',
linewidth=3)
plt.plot(x_vals, best_fit_upper, 'r--', linewidth=2)
plt.plot(x_vals, best_fit_lower, 'r--', linewidth=2)
plt.ylim([0, 10])
plt.legend(loc='lower right')
plt.title('Sepal Length vs Pedal Width')
plt.xlabel('Pedal Width')
plt.ylabel('Sepal Length')
plt.show()

plt.plot(train_loss, 'k-', label='Train Set Loss')
plt.plot(test_loss, 'r--', label='Test Set Loss')
plt.title('L2 Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('L2 Loss')
plt.legend(loc='upper right')
plt.show()
```



SVM regression with a 0.5 margin on iris data (sepal length vs pedal width).



SVM Regression loss per generation on both the train and test sets.

How it works

Intuitively, we can think of SVM regression as a function that is trying to fit as many points in the width margin from the line as possible. The fitting of this line is somewhat sensitive to this parameter. If we choose too small epsilon, the algorithm won't be able to fit many points in the margin. If we choose too large of an epsilon, there will be many lines that are able to fit all the data points in the margin. We would want to error on the small size, since nearer points to the margin have contribute less loss than further away points.

Working with Kernels in Tensorflow

Getting ready

In this recipe, we will motivate the usage of kernels in Support Vector Machines. In the linear SVM section, we solved the soft margin with a specific loss function. A different approach to this method is to solve what is called the dual of the optimization problem. It can be shown that the dual for the linear SVM problem is given by the following formula.

Where

Here, the variable in the model will be the vector. Ideally, this vector will be quite sparse, only taking on values near 1 and -1 for the corresponding 'support vectors' of our data set. Our data point vectors are indicated by \mathbf{x} and our targets (1 or -1) are represented by y .

The kernel in the above equations is the dot product, $\mathbf{x} \cdot \mathbf{x}$, which gives us the linear kernel. This kernel is a square matrix filled with the dot products of the data points.

Instead of just doing the dot product between data points, we can expand them with more complicated functions into higher dimensions, in which the classes may be linear separable. This may seem needlessly complicated, but if we select a function, k , that has the property where

then k is called a kernel function. One of the more common kernels is the Gaussian kernel (also known as the radial basis function kernel or the RBF kernel). This kernel is described with the following equation.

In order to make predictions on this kernel, say at a point \mathbf{x} , we just substitute in the prediction point in the appropriate equation in the kernel as follows.

In this section, we will discuss how to implement the Gaussian kernel. We will also note where to make the substitution for implementing the linear kernel where appropriate. The dataset we will use will be manually created to show where the Gaussian kernel would be more appropriate to use over the linear kernel.

How to do it

1. First we load the necessary libraries and start a graph session.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets

sess = tf.Session()
```

2. Now we generate the data. The data we will generate will be two concentric rings of data, each ring will belong to a different class. We have to make sure the classes are -1 or 1 as well. Then we will split the data into x and y values for each class for plotting purposes.

```
(x_vals, y_vals) = datasets.make_circles(n_samples=500, factor=.5,
noise=.1)
y_vals = np.array([1 if y==1 else -1 for y in y_vals])
class1_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==1]
class1_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==1]
class2_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==-1]
class2_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==-1]
```

- Next we declare our batch size, placeholders, and create our model variable, `b`. For SVMs we tend to want larger batch sizes because we want a very stable model that won't fluctuate much with each training generation. Also note that we have an extra placeholder for the prediction points. To visualize the results, we will create a color grid to see which areas belong to which class at the end.

```
batch_size = 250
```

```
x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
prediction_grid = tf.placeholder(shape=[None, 2], dtype=tf.float32)
```

```
b = tf.Variable(tf.random_normal(shape=[1, batch_size]))
```

- We will now create the Gaussian kernel. This kernel can be expressed as matrix operations as follows below.

```
gamma = tf.constant(-50.0)
dist = tf.reduce_sum(tf.square(x_data), 1)
dist = tf.reshape(dist, [-1, 1])
sq_dists = tf.add(tf.sub(dist, tf.mul(2., tf.matmul(x_data,
tf.transpose(x_data))))), tf.transpose(dist))
my_kernel = tf.exp(tf.mul(gamma, tf.abs(sq_dists)))
```



Note the usage of broadcasting in the 'sq_dists' line of the add and subtract operations.

Note that the linear kernel can be expressed as `my_kernel = tf.matmul(x_data, tf.transpose(x_data))`.

- Now we declare the dual problem as stated prior in this recipe. At the end, instead of maximizing, we will be minimizing the negative of the loss function with a `tf.neg()` function.

```
model_output = tf.matmul(b, my_kernel)
first_term = tf.reduce_sum(b)
b_vec_cross = tf.matmul(tf.transpose(b), b)
y_target_cross = tf.matmul(y_target, tf.transpose(y_target))
second_term = tf.reduce_sum(tf.mul(my_kernel, tf.mul(b_vec_cross,
y_target_cross)))
loss = tf.neg(tf.sub(first_term, second_term))
```

6. We now create the prediction and accuracy functions. First, we must create a prediction kernel, similar to step 4 above, but instead of a kernel of the points with itself, we have the kernel of the points with the prediction data. The prediction is then the sign of the output of the model.

```
rA = tf.reshape(tf.reduce_sum(tf.square(x_data), 1), [-1,1])
rB = tf.reshape(tf.reduce_sum(tf.square(prediction_grid), 1), [-1,1])
pred_sq_dist = tf.add(tf.sub(rA, tf.mul(2., tf.matmul(x_data,
tf.transpose(prediction_grid)))), tf.transpose(rB))
pred_kernel = tf.exp(tf.mul(gamma, tf.abs(pred_sq_dist)))

prediction_output = tf.matmul(tf.mul(tf.transpose(y_target), b),
pred_kernel)
prediction = tf.sign(prediction_output - tf.reduce_mean(prediction_
output))
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.squeeze(prediction),
tf.squeeze(y_target)), tf.float32))
```



To implement the linear prediction kernel we can write `pred_kernel = tf.matmul(x_data, tf.transpose(prediction_grid))`.

7. Now we can create an optimizer and initialize all the variables.

```
my_opt = tf.train.GradientDescentOptimizer(0.001)
train_step = my_opt.minimize(loss)

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

8. Next we start the training loop. We will record the loss vector and the batch accuracy for each generation. When we run the accuracy, we have to put in all three placeholders, but we feed in the x data twice to get the prediction on the points.

```
loss_vec = []
batch_accuracy = []
for i in range(500):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
    loss_vec.append(temp_loss)
```

```

        acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
                                                y_target: rand_y,
                                                prediction_
grid:rand_x})
        batch_accuracy.append(acc_temp)

    if (i+1)%100==0:
        print('Step #' + str(i+1))
        print('Loss = ' + str(temp_loss))

Step #100
Loss = -28.0772
Step #200
Loss = -3.3628
Step #300
Loss = -58.862
Step #400
Loss = -75.1121
Step #500
Loss = -84.8905

```

9. In order to see the output class on the whole space, we will create a mesh of prediction points in our system and run the prediction on all of them.

```

x_min, x_max = x_vals[:, 0].min() - 1, x_vals[:, 0].max() + 1
y_min, y_max = x_vals[:, 1].min() - 1, x_vals[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                    np.arange(y_min, y_max, 0.02))
grid_points = np.c_[xx.ravel(), yy.ravel()]
[grid_predictions] = sess.run(prediction, feed_dict={x_data:
rand_x,
                                                y_target:
rand_y,
                                                prediction_
grid: grid_points})
grid_predictions = grid_predictions.reshape(xx.shape)
The following is code to plot the result, batch accuracy, and
loss.
plt.contourf(xx, yy, grid_predictions, cmap=plt.cm.Paired,
alpha=0.8)
plt.plot(class1_x, class1_y, 'ro', label='Class 1')
plt.plot(class2_x, class2_y, 'kx', label='Class -1')
plt.legend(loc='lower right')
plt.ylim([-1.5, 1.5])
plt.xlim([-1.5, 1.5])

```

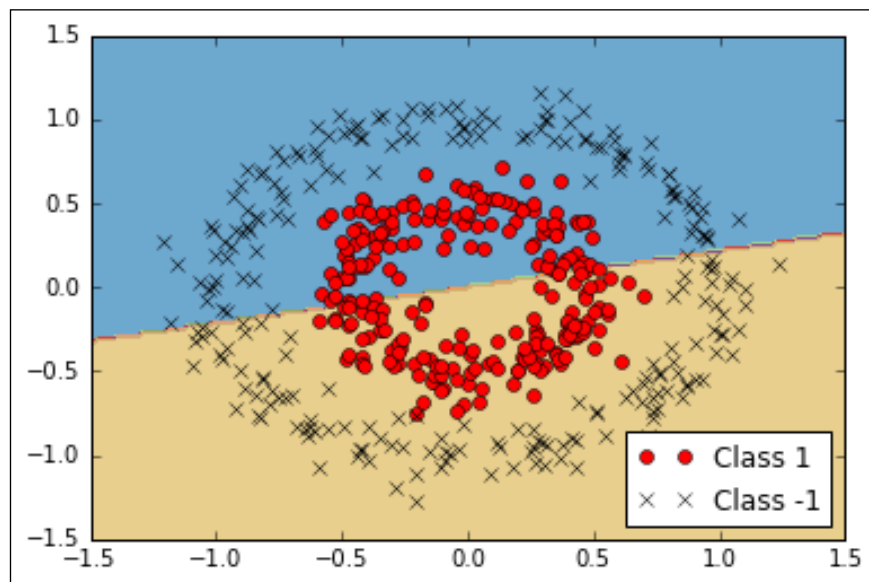


```
plt.show()

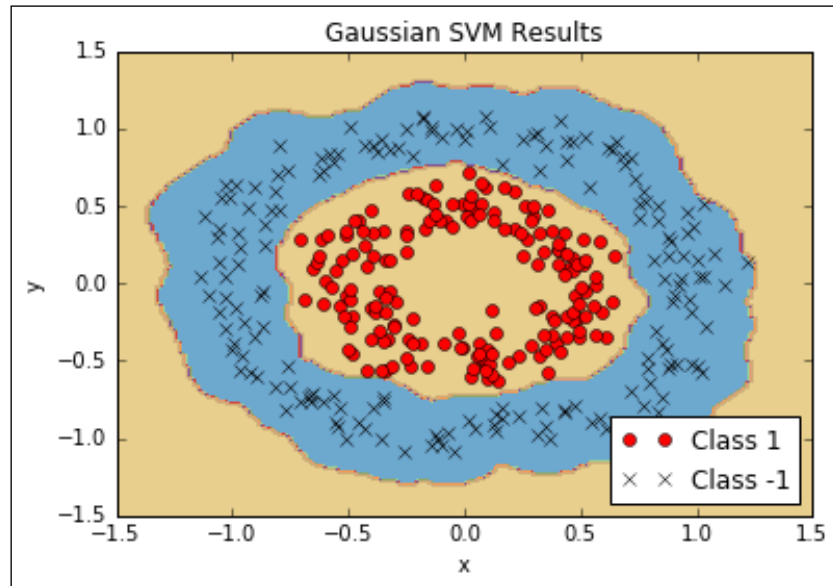
plt.plot(batch_accuracy, 'k-', label='Accuracy')
plt.title('Batch Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```

For succinctness, we will show only the results graph, but we can also separately run the plotting code and see all three if we so choose.



Linear SVM on non-linear separable data.



Nonlinear SVM with Gaussian kernel results on nonlinear ring data.

How it works

There were two important pieces to the code to know about. How we implemented the kernel and how we implemented the loss function for the SVM dual optimization problem. We have shown how to implement the linear and Gaussian kernel and that the Gaussian kernel can separate nonlinear datasets.

We should also mention that there is another parameter, the gamma value in the Gaussian kernel. This parameter controls how much influence points have on the curvature of the separation. Small values are commonly chosen, but it depends heavily on the dataset. Ideally this parameter is chosen with statistical techniques like cross validation.

There's more

Know that there are many more kernels that we could implement if we so choose. Here is a list of a few more common nonlinear kernels.

Polynomial homogeneous kernel.

Polynomial inhomogeneous kernel.

Hyperbolic tangent kernel.

Implementing a Non-Linear SVM

Getting ready

In this section we will implement the above Gaussian kernel SVM on real data. We will load the iris data set and create a classifier for I. setosa (vs non-setosa). We will see the effect of various gamma values on the classification.

How to do it

1. We first load the necessary libraries, which includes the scikit learn datasets so that we can load the iris data. Then we will start a graph session.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
```

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

2. Next we will load the iris data, extract the sepal length and pedal width, and separated the x and y values for each class (for plotting purposes later).

```
iris = datasets.load_iris()
x_vals = np.array([x[0], x[3]] for x in iris.data)
y_vals = np.array([1 if y==0 else -1 for y in iris.target])
class1_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==1]
class1_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==1]
class2_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==-1]
class2_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==-1]
```

3. Now we declare our batch size (larger batches preferred), placeholders, and the model variable, b.

```
batch_size = 100
```

```
x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
prediction_grid = tf.placeholder(shape=[None, 2], dtype=tf.float32)
```

```
b = tf.Variable(tf.random_normal(shape=[1, batch_size]))
```

4. Now we declare our Gaussian kernel. This kernel is dependent on the gamma value, and we will illustrate the effects of various gamma values on the classification later in this recipe.

```
gamma = tf.constant(-10.0)
dist = tf.reduce_sum(tf.square(x_data), 1)
dist = tf.reshape(dist, [-1,1])
sq_dists = tf.add(tf.sub(dist, tf.mul(2., tf.matmul(x_data,
tf.transpose(x_data))))), tf.transpose(dist))
my_kernel = tf.exp(tf.mul(gamma, tf.abs(sq_dists)))
```

5. We now compute the loss for the dual optimization problem.

```
model_output = tf.matmul(b, my_kernel)
first_term = tf.reduce_sum(b)
b_vec_cross = tf.matmul(tf.transpose(b), b)
y_target_cross = tf.matmul(y_target, tf.transpose(y_target))
second_term = tf.reduce_sum(tf.mul(my_kernel, tf.mul(b_vec_cross,
y_target_cross)))
loss = tf.neg(tf.sub(first_term, second_term))
```

6. In order to do predictions, we must create a prediction kernel. After that we also create a calculation of accuracy, which will just be a percentage of points correctly classified.

```
rA = tf.reshape(tf.reduce_sum(tf.square(x_data), 1), [-1,1])
rB = tf.reshape(tf.reduce_sum(tf.square(prediction_grid), 1), [-1,1])
pred_sq_dist = tf.add(tf.sub(rA, tf.mul(2., tf.matmul(x_data,
tf.transpose(prediction_grid))))), tf.transpose(rB))
pred_kernel = tf.exp(tf.mul(gamma, tf.abs(pred_sq_dist)))

prediction_output = tf.matmul(tf.mul(tf.transpose(y_target), b),
pred_kernel)
prediction = tf.sign(prediction_output - tf.reduce_mean(prediction_
output))
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.squeeze(prediction),
tf.squeeze(y_target)), tf.float32))
```

7. Next we declare our optimizer and initialize the variables.

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)

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

8. Now we can start the training loop. We run the loop for 300 generations and will store the loss value and the batch accuracy.

```
loss_vec = []
batch_accuracy = []
for i in range(300):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
    rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
    target: rand_y})
    loss_vec.append(temp_loss)

    acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
                                           y_target: rand_y,
                                           prediction_
    grid:rand_x})
    batch_accuracy.append(acc_temp)
```

9. In order to plot the decision boundary, we will create a mesh of x,y points and evaluate the prediction function we created on all of these points.

```
x_min, x_max = x_vals[:, 0].min() - 1, x_vals[:, 0].max() + 1
y_min, y_max = x_vals[:, 1].min() - 1, x_vals[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))
grid_points = np.c_[xx.ravel(), yy.ravel()]
[grid_predictions] = sess.run(prediction, feed_dict={x_data:
rand_x,
                                                    y_target:
rand_y,
                                                    prediction_
grid: grid_points})
grid_predictions = grid_predictions.reshape(xx.shape)
```

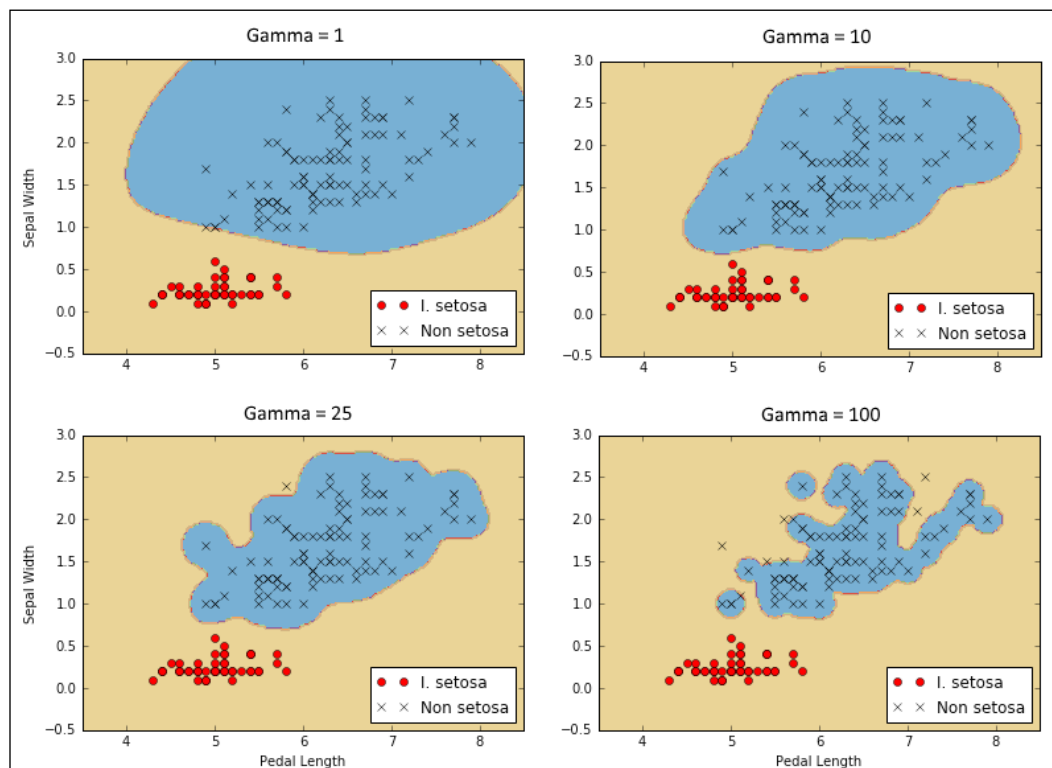
10. For succinctness, we will only show how to plot the points with the decision boundaries. For the plot and effect of gamma, see the next section in this recipe.

```
plt.contourf(xx, yy, grid_predictions, cmap=plt.cm.Paired,
alpha=0.8)
plt.plot(class1_x, class1_y, 'ro', label='I. setosa')
plt.plot(class2_x, class2_y, 'kx', label='Non setosa')
plt.title('Gaussian SVM Results on Iris Data')
plt.xlabel('Pedal Length')
plt.ylabel('Sepal Width')
```

```
plt.legend(loc='lower right')
plt.ylim([-0.5, 3.0])
plt.xlim([3.5, 8.5])
plt.show()
```

How it works

Here is the classification of *I. setosa* results for four different gamma values (1, 10, 25, 100). Notice how the higher the gamma value, the more of an effect each individual point has on the classification boundary.



Classification results of *I. setosa* using a Gaussian kernel SVM with four different values of gamma.

Insert Image kernel results on nonlinear ring data.

Implementing a Multi-Class SVM

Getting ready

By design, SVM algorithms are binary classifiers. However, there are a few strategies employed to get them to work on multiple classes. The two main strategies are called one vs all, and one vs one.

One vs. one is a strategy where a binary classifier is created for each possible pair of classes. Then a prediction is made for a point for a class that has the most votes. This can be computationally hard as we must create classifiers for classes.

Another way to implement multiclass classifiers is to do a one vs all strategy where we create a classifier for each of the classes. The predicted class of a point will be the class that creates the largest SVM margin. This is the strategy we will implement in this section

Here, we will load the iris dataset and perform multiclass nonlinear SVM with a Gaussian kernel. The iris dataset is ideal because there are three classes (I. setosa, I. virginica, I. versicolor). We will create three Gaussian kernel SVMs for each class and make the prediction of points where the highest margin exists.

How to do it

1. First we load the libraries we need and start a graph.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets

sess = tf.Session()
```

2. Now we will load the iris dataset and spit apart the targets for each class. We will only be using sepal length and pedal width to illustrate because we want to be able to plot the outputs. We also separate the x and y values for each class for plotting purposes at the end.

```
iris = datasets.load_iris()
x_vals = np.array([x[0], x[3]] for x in iris.data)
y_vals1 = np.array([1 if y==0 else -1 for y in iris.target])
y_vals2 = np.array([1 if y==1 else -1 for y in iris.target])
y_vals3 = np.array([1 if y==2 else -1 for y in iris.target])
y_vals = np.array([y_vals1, y_vals2, y_vals3])
class1_x = [x[0] for i,x in enumerate(x_vals) if iris.
target[i]==0]
```

```

class1_y = [x[1] for i,x in enumerate(x_vals) if iris.
target[i]==0]
class2_x = [x[0] for i,x in enumerate(x_vals) if iris.
target[i]==1]
class2_y = [x[1] for i,x in enumerate(x_vals) if iris.
target[i]==1]
class3_x = [x[0] for i,x in enumerate(x_vals) if iris.
target[i]==2]
class3_y = [x[1] for i,x in enumerate(x_vals) if iris.
target[i]==2]

```

3. The biggest changes we have in this example as compared to the prior nonlinear SVM recipes is that a lot of the dimensions will change (we have 3 classifiers now instead of one). We will also make use of matrix broadcasting and reshaping techniques to calculate all three SVMs at once. Since we are doing this all at once, our `y_target` placeholder now has dimensions `[3, None]` and our model variable, `b`, will be initialized to be size `[3, batch_size]`.

```
batch_size = 50
```

```

x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[3, None], dtype=tf.float32)
prediction_grid = tf.placeholder(shape=[None, 2], dtype=tf.
float32)

```

```
b = tf.Variable(tf.random_normal(shape=[3, batch_size]))
```

4. Next we calculate the Gaussian kernel. Since this is only dependent on the `x` data, this code doesn't change from the prior recipe.

```

gamma = tf.constant(-10.0)
dist = tf.reduce_sum(tf.square(x_data), 1)
dist = tf.reshape(dist, [-1,1])
sq_dists = tf.add(tf.sub(dist, tf.mul(2., tf.matmul(x_data,
tf.transpose(x_data)))), tf.transpose(dist))
my_kernel = tf.exp(tf.mul(gamma, tf.abs(sq_dists)))

```

5. One big change is that we will want to do batch matrix multiplication. We will end up with 3-dimensional matrices and we will want to broadcast matrix multiplication across the third index. Our data and target matrices are not setup for this. In order for an operation like to work across an extra dimension, we create a function to expand such matrices, reshape the matrix into a transpose, and then call Tensorflow's `batch_matmul` across the extra dimension.

```

def reshape_matmul(mat):
    v1 = tf.expand_dims(mat, 1)
    v2 = tf.reshape(v1, [3, batch_size, 1])
    return(tf.batch_matmul(v2, v1))

```


6. With this function created, we can now compute the dual loss function.

```
model_output = tf.matmul(b, my_kernel)
first_term = tf.reduce_sum(b)
b_vec_cross = tf.matmul(tf.transpose(b), b)
y_target_cross = reshape_matmul(y_target)

second_term = tf.reduce_sum(tf.mul(my_kernel, tf.mul(b_vec_cross,
y_target_cross)), [1,2])
loss = tf.reduce_sum(tf.neg(tf.sub(first_term, second_term)))
```

7. Now we can create the prediction kernel. Notice that we have to be careful with the `reduce_sum` function and not reduce across all three SVM predictions, so we have to tell Tensorflow not to sum everything up with a second index argument.

```
rA = tf.reshape(tf.reduce_sum(tf.square(x_data), 1), [-1,1])
rB = tf.reshape(tf.reduce_sum(tf.square(prediction_grid), 1), [-1,1])
pred_sq_dist = tf.add(tf.sub(rA, tf.mul(2., tf.matmul(x_data,
tf.transpose(prediction_grid)))), tf.transpose(rB))
pred_kernel = tf.exp(tf.mul(gamma, tf.abs(pred_sq_dist)))
```

8. When we are done with the prediction kernel, we can create predictions. A big change here is that the predictions are not the `sign()` of the output. Since we are implementing a one vs. all strategy, the prediction is the classifier that has the largest output. To accomplish this we use Tensorflow's built in `argmax()` function.

```
prediction_output = tf.matmul(tf.mul(y_target,b), pred_kernel)
prediction = tf.argmax(prediction_output-tf.expand_dims(tf.
reduce_mean(prediction_output,1), 1), 0)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction,
tf.argmax(y_target,0)), tf.float32))
```

9. Now that we have the kernel, loss, and prediction capabilities up, we just have to declare our optimizer and initialize our variables.

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)

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

10. This algorithm converges relatively quickly, so we won't have run the training loop for more than 100 iterations. We do so with the following code.

```
loss_vec = []
batch_accuracy = []
for i in range(100):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
```

```

    rand_y = y_vals[:,rand_index]
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
    loss_vec.append(temp_loss)

    acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
                                           y_target: rand_y,
                                           prediction_
grid:rand_x})
    batch_accuracy.append(acc_temp)

    if (i+1)%25==0:
        print('Step #' + str(i+1))
        print('Loss = ' + str(temp_loss))
Step #25
Loss = -2.8951
Step #50
Loss = -27.9612
Step #75
Loss = -26.896
Step #100
Loss = -30.2325

```

11. We can now create the prediction grid of points and run the prediction function on all of them.

```

x_min, x_max = x_vals[:, 0].min() - 1, x_vals[:, 0].max() + 1
y_min, y_max = x_vals[:, 1].min() - 1, x_vals[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))
grid_points = np.c_[xx.ravel(), yy.ravel()]
grid_predictions = sess.run(prediction, feed_dict={x_data: rand_x,
                                                  y_target:
rand_y,
                                                  prediction_
grid: grid_points})
grid_predictions = grid_predictions.reshape(xx.shape)

```

12. The following is code to plot the results, batch accuracy, and loss function. For succinctness we will only display the end result.

```

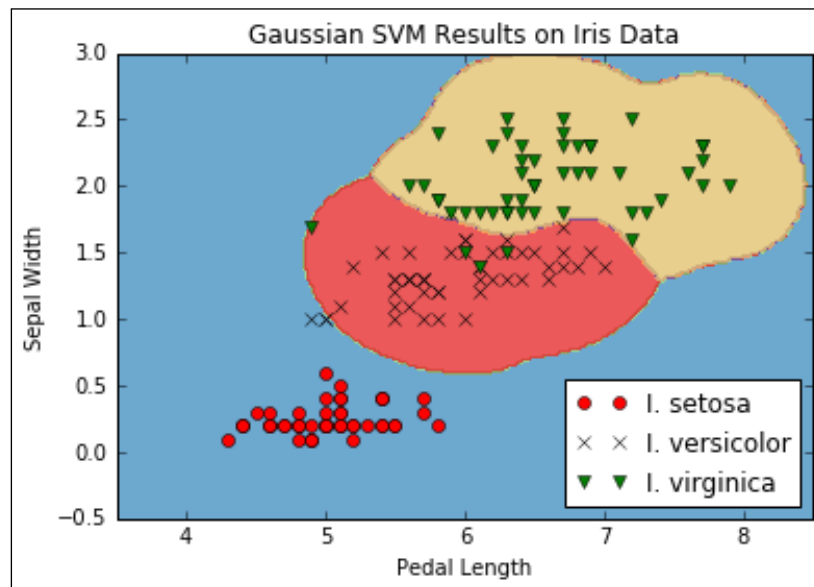
plt.contourf(xx, yy, grid_predictions, cmap=plt.cm.Paired,
alpha=0.8)
plt.plot(class1_x, class1_y, 'ro', label='I. setosa')

```

```
plt.plot(class2_x, class2_y, 'kx', label='I. versicolor')
plt.plot(class3_x, class3_y, 'gv', label='I. virginica')
plt.title('Gaussian SVM Results on Iris Data')
plt.xlabel('Pedal Length')
plt.ylabel('Sepal Width')
plt.legend(loc='lower right')
plt.ylim([-0.5, 3.0])
plt.xlim([3.5, 8.5])
plt.show()

plt.plot(batch_accuracy, 'k-', label='Accuracy')
plt.title('Batch Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```



Multiclass (3 classes) nonlinear Gaussian SVM results on the iris dataset with gamma = 10.

How it works

The important points to notice in this recipe is how we changed our algorithm to optimize over three SVM models at once. Our model parameter, , has an extra dimension to take into account all three models. Here we can see that the extension of an algorithm to multiple similar algorithms was made relatively easy due to Tensorflow's built in capabilities to deal with extra dimensions.

5

Nearest Neighbor Methods

This chapter will focus on nearest neighbor methods and how to implement them in Tensorflow. We will start with an introduction to the method and show how to implement various forms while ending with examples on address matching and image recognition.

- ▶ Introduction
- ▶ Working with Nearest Neighbors
- ▶ Working with Text Based Distances
- ▶ Computing Mixed Distance Functions
- ▶ Using an Address Matching Example
- ▶ Using Nearest Neighbors for Image Recognition


Introduction

Nearest neighbor methods are based on a simple idea. We consider our training set as the model and make predictions on new points based on how close they are to points in the training set. The most naïve way is to make the prediction as the closest training data point class. But since most datasets contain a degree of noise, a more common method would be to take a weighted average of a set of k nearest neighbors. This method is called k -nearest neighbors (knn).

Given a training dataset, X , with corresponding targets, y , we can make a prediction on a point, x , by looking at a set of nearest neighbors. The actual method of prediction depends on whether or not we are doing regression (continuous) or classification (discrete).

For discrete classification targets, the prediction may be given by a maximum voting scheme weighted by the distance to the prediction point.

- ▶ Here, our prediction, \hat{y} , is the maximum weighted value over all classes, j , where the weighted distance from the prediction point to the training point i , is given by $\frac{1}{d(i, \hat{y})}$. And I_{ij} is just an indicator function if point i is in class j .
- ▶ For continuous regression targets, the prediction is given by a weighted average of all k points nearest to the prediction.
- ▶ It is obvious that the prediction is heavily dependent on the choice of the distance metric, d .
- ▶ Common specifications of the distance metric are L1 and L2 distances.
- ▶ There are many different specifications of distance metrics that we can choose. In this chapter we will explore the L1 and L2 metrics as well as edit and textual distances.
- ▶ We also have to choose how to weight the distances. A straightforward way to weight the distances is by the distance itself. Further away points from our prediction should have less impact than nearer points. The most common way to weight is by the normalized inverse of the distance. We will implement this method in the next recipe.

 Note that k-nearest neighbors is an aggregating method. For regression, we are performing a weighted average of neighbors. Because of this, predictions will be less extreme and less varied than the actual targets. The magnitude of this effect will be determined by k , the number of neighbors in the algorithm

Working with Nearest Neighbors

Getting ready

To illustrate how making predictions with nearest neighbors works in Tensorflow, we will use the Boston housing dataset. Here we will be predicting median neighbourhood housing value as a function of several features.

Since we consider the training set the trained model, we will find the k -nearest neighbors to the prediction points and do a weighted average of the target value.

How to do it

1. First we will start by loading the required libraries and starting a graph session. We will use the requests module to load the necessary Boston housing data from the UCI machine learning repository.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import requests
```

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

2. Next we will load the data using the requests module.

```
housing_url = 'https://archive.ics.uci.edu/ml/machine-learning-
databases/housing/housing.data'
housing_header = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM',
'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
cols_used = ['CRIM', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'TAX',
'PTRATIO', 'B', 'LSTAT']
num_features = len(cols_used)
housing_file = requests.get(housing_url)
housing_data = [[float(x) for x in y.split(' ') if len(x)>=1] for
y in housing_file.text.split('\n') if len(y)>=1]
```

3. Next we separate the data into our dependent and independent features. We will be predicting the last variable, 'MEDV', which is the median value for the group of houses. We will also not use the features 'ZN', 'CHAS', and 'RAD' because of their uninformative or binary nature.

```
y_vals = np.transpose([np.array([y[13] for y in housing_data])])
x_vals = np.array([x for i,x in enumerate(y) if housing_header[i]
in cols_used] for y in housing_data])
x_vals = (x_vals - x_vals.min(0)) / x_vals.ptp(0)
```

4. Now we split the x and y values into the train and test sets. We will create the training set by selecting about 80% of the rows at random and leaving the remaining 20% for the test set.

```
train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```


5. Next we declare our k value and batch size.

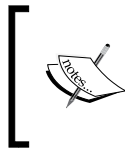
```
k = 4
batch_size=len(x_vals_test)
```

6. We will declare our placeholders next. Remember that there are no model variables to train, as the model is determined exactly by our training set.

```
x_data_train = tf.placeholder(shape=[None, num_features],
dtype=tf.float32)
x_data_test = tf.placeholder(shape=[None, num_features], dtype=tf.
float32)
y_target_train = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target_test = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

7. Next we create our distance function for a batch of test points. Here we illustrate the use of the L1 distance.

```
distance = tf.reduce_sum(tf.abs(tf.sub(x_data_train, tf.expand_
dims(x_data_test,1))), reduction_indices=2)
```



Note that the L2 distance function can be used as well. We would change the distance formula to the following: `distance = tf.sqrt(tf.reduce_sum(tf.square(tf.sub(x_data_train, tf.expand_dims(x_data_test,1))), reduction_indices=1))`

8. Now we create our prediction function. To do this, we will use the function 'top_k()', which returns the values and indices of the largest values in a tensor. Since we want the indices of the smallest distances, we will instead find the k-biggest negative distances. We also declare the predictions and the mean squared error (MSE) of the target values.

```
top_k_xvals, top_k_indices = tf.nn.top_k(tf.neg(distance), k=k)
x_sums = tf.expand_dims(tf.reduce_sum(top_k_xvals, 1),1)
x_sums_repeated = tf.matmul(x_sums,tf.ones([1, k], tf.float32))
x_val_weights = tf.expand_dims(tf.div(top_k_xvals,x_sums_
repeated), 1)

top_k_yvals = tf.gather(y_target_train, top_k_indices)
prediction = tf.squeeze(tf.batch_matmul(x_val_weights,top_k_
yvals), squeeze_dims=[1])
mse = tf.div(tf.reduce_sum(tf.square(tf.sub(prediction, y_target_
test))), batch_size)
```

9. Test

```
num_loops = int(np.ceil(len(x_vals_test)/batch_size))

for i in range(num_loops):
    min_index = i*batch_size
```

```

max_index = min((i+1)*batch_size,len(x_vals_train))
x_batch = x_vals_test[min_index:max_index]
y_batch = y_vals_test[min_index:max_index]
predictions = sess.run(prediction, feed_dict={x_data_train:
x_vals_train, x_data_test: x_batch, y_target_train: y_vals_train,
y_target_test: y_batch})
batch_mse = sess.run(mse, feed_dict={x_data_train: x_vals_
train, x_data_test: x_batch, y_target_train: y_vals_train, y_
target_test: y_batch})

print('Batch #' + str(i+1) + ' MSE: ' + str(np.round(batch_
mse,3)))

```

Batch #1 MSE: 23.153

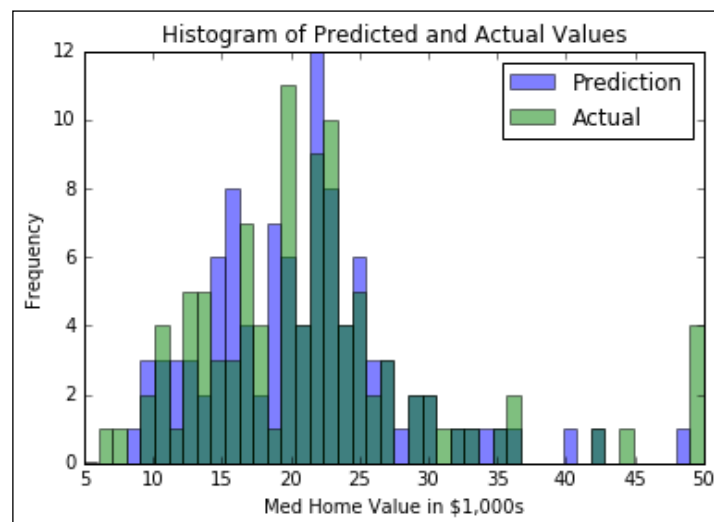
10. Additionally, we can also look at a histogram of the actual target values compared with the predicted values. One reason to look at this is to notice the fact that with an averaging method, we have trouble predicting the extreme ends of the targets.

```
bins = np.linspace(5, 50, 45)
```

```

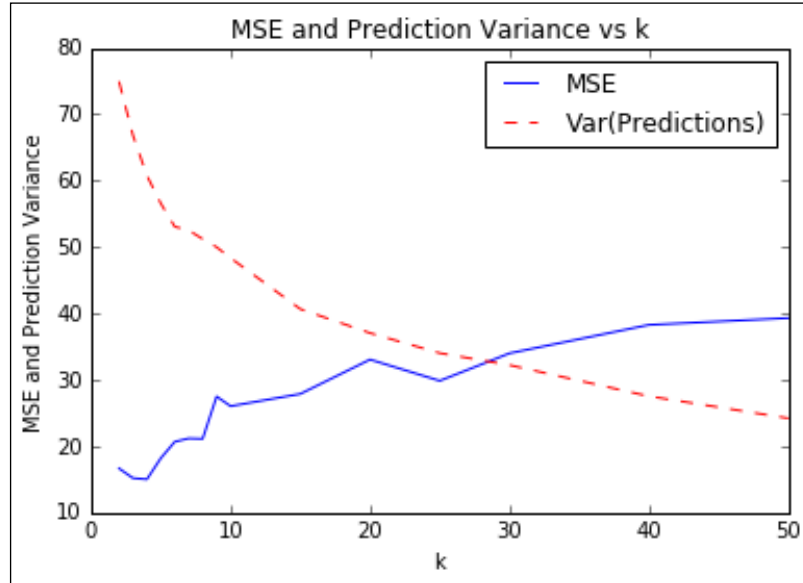
plt.hist(predictions, bins, alpha=0.5, label='Prediction')
plt.hist(y_batch, bins, alpha=0.5, label='Actual')
plt.title('Histogram of Predicted and Actual Values')
plt.xlabel('Med Home Value in $1,000s')
plt.ylabel('Frequency')
plt.legend(loc='upper right')
plt.show()

```



A histogram of the predicted values and actual target values for k-nearest neighbors (k=4).

11. One hard thing to determine is the best value of k . For the above figure and predictions, we used $k=4$ for our model. We chose this specifically because it gives us the lowest MSE. This is verified by cross validation. If we use cross validation across multiple values of k , we will see that $k=4$ gives us a minimum MSE. We show this in the below figure. It is also worthwhile to plot the variance in the predicted values to show that it will decrease the more neighbors we average over.



The MSE for k -NN predictions for various values of k . We also plot the variance of the predicted values on the test set. Note that the variance decreases with the higher k .

How it works

With the nearest neighbors algorithm, the model is the training set. Because of this, we do not have to train any variables in our model. The only parameter, k , we determined via cross validation to minimize our mean squared error.

There's more

For the weighting of the k -nearest neighbors, we chose to weight directly by the distance. There are other options that we could consider as well. Another common method is to weight by the inverse squared distance.

Working with Text Based Distances

Getting ready

In this recipe, we will illustrate how to use Tensorflow's text distance metric, the Levenshtein distance (the edit distance), between strings. This will be important later in this chapter as we expand the nearest neighbor methods to include features with text.

The Levenshtein distance is the minimal number of edits to get from one string to another string. The allowed edits are inserting a character, deleting a character, or substituting a character with a different one. For this recipe, we will use Tensorflow's Levenshtein distance function, `'edit_distance()'`. It is worthwhile to illustrate the use of this function because the usage of this function will be applicable to later chapters.



Note that Tensorflow's `edit_distance()` function only accepts sparse tensors. We will have to create our strings as sparse tensors of individual characters.

How to do it

1. First we load Tensorflow and initialize a graph.

```
import tensorflow as tf

sess = tf.Session()
```

2. First, we will show how to calculate the edit distance between two words, 'bear' and 'beer'. First we will create a list of characters from our strings with python's `'list()'` function. Next we create a sparse 3d matrix from that list. We have to tell Tensorflow the indices of the characters, the characters, and the shape of the matrix. After this we can decide if we want the total edit distance (`normalize=False`) or the normalized edit distance (`normalize=True`), in which we divide the edit distance by the length of the second word.



Tensorflow's documentation treats the two strings as a proposed (hypothesis) string and a ground truth string. We will continue this notation here with 'h' and 't' tensors.

```
hypothesis = list('bear')
truth = list('beers')
h1 = tf.SparseTensor([[0,0,0], [0,0,1], [0,0,2], [0,0,3]],
                    hypothesis,
                    [1,1,1])
```

```
t1 = tf.SparseTensor([[0,0,0], [0,0,1], [0,0,2], [0,0,3], [0,0,4]],
                    truth,
                    [1,1,1])

print(sess.run(tf.edit_distance(h1, t1, normalize=False)))

[[ 2.]]
```

3. Next we will illustrate how to compare two words, 'bear' and 'beer' both with another word, 'beers'. In order to achieve this, we must replicate the 'beers' in order to have the same amount of comparable words.

```
hypothesis2 = list('bearbeer')
truth2 = list('beersbeers')
h2 = tf.SparseTensor([[0,0,0], [0,0,1], [0,0,2], [0,0,3], [0,1,0],
                    [0,1,1], [0,1,2], [0,1,3]],
                    hypothesis2,
                    [1,2,4])

t2 = tf.SparseTensor([[0,0,0], [0,0,1], [0,0,2], [0,0,3], [0,0,4],
                    [0,1,0], [0,1,1], [0,1,2], [0,1,3], [0,1,4]],
                    truth2,
                    [1,2,5])

print(sess.run(tf.edit_distance(h2, t2, normalize=True)))

[[ 0.40000001  0.2      ]]
```

4. A more efficient way to compare a set of words against another word is shown in this example. We create the indices and list of characters beforehand for both the hypothesis and ground truth string.

```
hypothesis_words = ['bear', 'bar', 'tensor', 'flow']
truth_word = ['beers']

num_h_words = len(hypothesis_words)
h_indices = [[xi, 0, yi] for xi,x in enumerate(hypothesis_words)
              for yi,y in enumerate(x)]
h_chars = list(''.join(hypothesis_words))

h3 = tf.SparseTensor(h_indices, h_chars, [num_h_words,1,1])

truth_word_vec = truth_word*num_h_words
t_indices = [[xi, 0, yi] for xi,x in enumerate(truth_word_vec) for
              yi,y in enumerate(x)]
t_chars = list(''.join(truth_word_vec))
```

```

t3 = tf.SparseTensor(t_indices, t_chars, [num_h_words,1,1])

print(sess.run(tf.edit_distance(h3, t3, normalize=True)))

[[ 0.40000001]
 [ 0.60000002]
 [ 0.80000001]
 [ 1.         ]]

```

5. Now we will illustrate how to calculate the edit distance between two word lists using placeholders. The concept is the same, except we will be feeding in 'SparseTensorValue()' instead of sparse tensors. First we will create a function that creates the sparse tensors from a word list.

```

def create_sparse_vec(word_list):
    num_words = len(word_list)
    indices = [[xi, 0, yi] for xi,x in enumerate(word_list) for
yi,y in enumerate(x)]
    chars = list(''.join(word_list))
    return(tf.SparseTensorValue(indices, chars, [num_words,1,1]))

hyp_string_sparse = create_sparse_vec(hypothesis_words)
truth_string_sparse = create_sparse_vec(truth_word*len(hypothesis_
words))

hyp_input = tf.sparse_placeholder(dtype=tf.string)
truth_input = tf.sparse_placeholder(dtype=tf.string)

edit_distances = tf.edit_distance(hyp_input, truth_input,
normalize=True)

feed_dict = {hyp_input: hyp_string_sparse,
              truth_input: truth_string_sparse}

print(sess.run(edit_distances, feed_dict=feed_dict))

[[ 0.40000001]
 [ 0.60000002]
 [ 0.80000001]
 [ 1.         ]]

```

How it works

There's more

Other text distance metrics exist that we should discuss. Below is a definition table describing other various text distances between two strings, s and t .

Name	Description	Formula
Hamming distance	Number of equal character positions. Only valid if both strings are equal length.	$\sum I(s_i = t_i)$, where I is an indicator function of equal characters.
Cosine distance	The dot product of the k-gram differences divided by the L2 norm of the k-gram differences.	
Jaccard distance	Number of characters in common divided by total union of characters in both strings.	

Computing with Mixed Distance Functions

Getting ready

It is important to extend the nearest neighbor algorithm to take into account variables that are scaled differently. In this example we will show how to scale the distance function for different variables. Specifically, we will scale the distance function as a function of the feature variance.

The key to weighting the distance function is to use a weight matrix. The distance function written with matrix operations becomes the following formula.

Here, W , is a diagonal weight matrix that we use to scale the distance metric for each feature.

For this recipe we will try to improve our MSE on the Boston housing value data set. This data set is a great example of features that are on different scales and the nearest neighbor algorithm would benefit from scaling the distance function.

How to do it

1. First, we will load the necessary libraries and start a graph session.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

```
import requests
```

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

2. Next we load the data and store it in a numpy array. Again, note that we will only use certain columns for prediction. We do not use id variables nor variables that have very low variance.

```
housing_url = 'https://archive.ics.uci.edu/ml/machine-learning-
databases/housing/housing.data'
housing_header = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM',
'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
cols_used = ['CRIM', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'TAX',
'PTRATIO', 'B', 'LSTAT']
num_features = len(cols_used)
housing_file = requests.get(housing_url)
housing_data = [[float(x) for x in y.split(' ') if len(x)>=1] for
y in housing_file.text.split('\n') if len(y)>=1]

y_vals = np.transpose([np.array([y[13] for y in housing_data])])
x_vals = np.array([x for i,x in enumerate(y) if housing_header[i]
in cols_used] for y in housing_data])
```

3. Now we scale the x values to be between zero and one with min-max scaling.
4. We now create the diagonal weight matrix that will provide the scaling of the distance metric by the standard deviation of the features.

```
weight_diagonal = x_vals.std(0)
weight_matrix = tf.cast(tf.diag(weight_diagonal), dtype=tf.
float32)
```

5. Now we split the data into a train and test set. We also declare k, the amount of nearest neighbors and make the batch size equal to the test set size.

```
train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]

k = 4
batch_size=len(x_vals_test)
```


6. We declare our placeholders that we need next. We have four place holders, the x-inputs and y-targets for both the train and test set.

```
x_data_train = tf.placeholder(shape=[None, num_features],
dtype=tf.float32)
x_data_test = tf.placeholder(shape=[None, num_features], dtype=tf.
float32)
y_target_train = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target_test = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

7. Now we can declare our distance function. For readability, we break up the distance function into its components. Know that we will have to tile the weight matrix by the batch size and use the batch_matmul() function to perform batch matrix multiplication across the batch size.

```
subtraction_term = tf.sub(x_data_train, tf.expand_dims(x_data_
test,1))
first_product = tf.batch_matmul(subtraction_term, tf.tile(tf.
expand_dims(weight_matrix,0), [batch_size,1,1]))
second_product = tf.batch_matmul(first_product,
tf.transpose(subtraction_term, perm=[0,2,1]))
distance = tf.sqrt(tf.batch_matrix_diag_part(second_product))
```

8. After we calculate all the training distances for each test point, we need to return the top k-nearest neighbors. We do this with the top_k() function. Since this function returns the largest values, and we want the smallest distances we return the largest of the negative distance values. We then want to make predictions as the weighted average of the distances of the top k neighbors.

```
top_k_xvals, top_k_indices = tf.nn.top_k(tf.neg(distance), k=k)
x_sums = tf.expand_dims(tf.reduce_sum(top_k_xvals, 1),1)
x_sums_repeated = tf.matmul(x_sums,tf.ones([1, k], tf.float32))
x_val_weights = tf.expand_dims(tf.div(top_k_xvals,x_sums_
repeated), 1)
top_k_yvals = tf.gather(y_target_train, top_k_indices)
prediction = tf.squeeze(tf.batch_matmul(x_val_weights,top_k_
yvals), squeeze_dims=[1])
```

9. To evaluate our model, we calculate the MSE of our predictions.

```
mse = tf.div(tf.reduce_sum(tf.square(tf.sub(prediction, y_target_
test))), batch_size)
```

10. Now we can loop through our test batches and calculate the MSE for each.

```
num_loops = int(np.ceil(len(x_vals_test)/batch_size))

for i in range(num_loops):
    min_index = i*batch_size
    max_index = min((i+1)*batch_size,len(x_vals_train))
```

```

x_batch = x_vals_test[min_index:max_index]
y_batch = y_vals_test[min_index:max_index]
predictions = sess.run(prediction, feed_dict={x_data_train: x_
vals_train, x_data_test: x_batch,
                                     y_target_train: y_vals_
train, y_target_test: y_batch})
batch_mse = sess.run(mse, feed_dict={x_data_train: x_vals_
train, x_data_test: x_batch,
                                     y_target_train: y_vals_
train, y_target_test: y_batch})

print('Batch #' + str(i+1) + ' MSE: ' + str(np.round(batch_
mse,3)))

```

Batch #1 MSE: 21.322

11. As a final comparison, we can plot the distribution of housing values for the actual test set and the predictions on the test set with the below code.

```

bins = np.linspace(5, 50, 45)

plt.hist(predictions, bins, alpha=0.5, label='Prediction')
plt.hist(y_batch, bins, alpha=0.5, label='Actual')
plt.title('Histogram of Predicted and Actual Values')
plt.xlabel('Med Home Value in $1,000s')
plt.ylabel('Frequency')
plt.legend(loc='upper right')
plt.show()

```

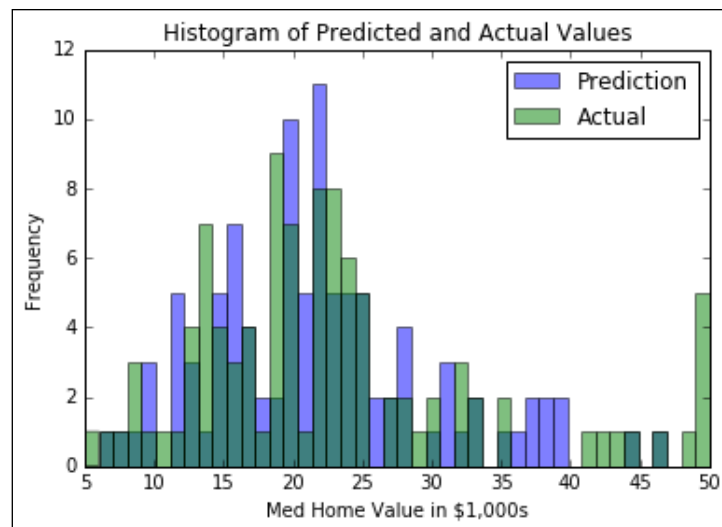


Figure depicting the two histograms of the predicted and actual housing values on the Boston data set, this time we have scaled the distance function differently for each feature.

How it works

We decreased our MSE on the test set here by introducing a method of scaling the distance functions for each feature. Here, we scaled the distance functions by a factor of the feature standard deviation. This makes for a more accurate view of measuring which points are the closest neighbors or not. From this we also took the weighted average of the top k neighbors as a function of distance to get the housing value prediction.

There's more

This scaling factor can also be used to downweight or upweight features in the nearest neighbor distance calculation. This can be useful for situations where we trust features more or less than others.

Using an Address Matching Example

Getting ready

Nearest neighbor is a great algorithm to use for address matching. Address matching is a type of record matching in which we have addresses in multiple data sets and we would like to match them up. In address matching, we may have typos in the address, different cities, or different zip codes, but they may all refer to the same address. Nearest neighbor across the numerical and character components of an address may help us identify addresses that are actually the same.

In this example, we will generate two data sets. Each data set will be comprised of a street address and a zip code. But one data set has a high number of typos in the street address. We will take the non-typo data set as our gold standard and return one address from it for each typo address that is the closest as a function of string-distance (for the street) and numerical distance (for the zip code).

The first part of the code will focus on generating the two data sets. Then the second part of the code will run through the test set and return the closest address from the training set.

How to do it

1. We first start by loading the necessary libraries.

```
import random
import string
import numpy as np
import tensorflow as tf
```

2. We will now create the reference dataset. To show succinct output, we will only make each dataset comprise of 10 addresses (but it can be run with many more).

```
n = 10
street_names = ['abbey', 'baker', 'canal', 'donner', 'elm']
street_types = ['rd', 'st', 'ln', 'pass', 'ave']
rand_zips = [random.randint(65000,65999) for i in range(5)]

numbers = [random.randint(1, 9999) for i in range(n)]
streets = [random.choice(street_names) for i in range(n)]
street_suffs = [random.choice(street_types) for i in range(n)]
zips = [random.choice(rand_zips) for i in range(n)]
full_streets = [str(x) + ' ' + y + ' ' + z for x,y,z in
zip(numbers, streets, street_suffs)]
reference_data = [list(x) for x in zip(full_streets,zips)]
```

3. To create the test set, we need a function that will randomly create a typo in a string and return the resulting string.

```
def create_typo(s, prob=0.75):
    if random.uniform(0,1) < prob:
        rand_ind = random.choice(range(len(s)))
        s_list = list(s)
        s_list[rand_ind]=random.choice(string.ascii_lowercase)
        s = ''.join(s_list)
    return(s)

typo_streets = [create_typo(x) for x in streets]
typo_full_streets = [str(x) + ' ' + y + ' ' + z for x,y,z in
zip(numbers, typo_streets, street_suffs)]
test_data = [list(x) for x in zip(typo_full_streets,zips)]
```

4. Now we can initialize a graph session and declare the placeholders we need. We will need four placeholders, in each test and reference set, we will need an address and zip code placeholder.

```
sess = tf.Session()

test_address = tf.sparse_placeholder( dtype=tf.string)
test_zip = tf.placeholder(shape=[None, 1], dtype=tf.float32)
ref_address = tf.sparse_placeholder(dtype=tf.string)
ref_zip = tf.placeholder(shape=[None, n], dtype=tf.float32)
Now we declare the numerical zip distance and the edit distance
for the address string.
zip_dist = tf.square(tf.sub(ref_zip, test_zip))

address_dist = tf.edit_distance(test_address, ref_address,
normalize=True)
```

5. We now convert the zip distance and the address distance into similarities. For the similarities we want a similarity of '1' when the two inputs are exactly the same and near '0' when they are very different. For the zip distance, we can do this by taking the distances, subtracting from the max, and then dividing by the range of the distances. For the address similarity, since the distance is already scaled between 0 and 1, we just subtract it from one to get the similarity.

```
zip_max = tf.gather(tf.squeeze(zip_dist), tf.argmax(zip_dist, 1))
zip_min = tf.gather(tf.squeeze(zip_dist), tf.argmin(zip_dist, 1))
zip_sim = tf.div(tf.sub(zip_max, zip_dist), tf.sub(zip_max, zip_min))
address_sim = tf.sub(1., address_dist)
```

6. To combine the two similarity functions, we take a weighted average of the two. For this recipe, we put equal weight on the address and the zip code. We can also change this depending on how much we trust each feature. We then return the index of the highest similarity of the reference set.

```
address_weight = 0.5
zip_weight = 1. - address_weight
weighted_sim = tf.add(tf.transpose(tf.mul(address_weight, address_sim)),
                      tf.mul(zip_weight, zip_sim))

top_match_index = tf.argmax(weighted_sim, 1)
```

7. In order to use the edit distance in Tensorflow, we have to convert the address strings to a sparse vector. In a prior recipe in this chapter, 'Working with Text Based Distances' we created the following function and will use it in this recipe as well.

```
def sparse_from_word_vec(word_vec):
    num_words = len(word_vec)
    indices = [[xi, 0, yi] for xi,x in enumerate(word_vec) for
               yi,y in enumerate(x)]
    chars = list('').join(word_vec)
    return(tf.SparseTensorValue(indices, chars, [num_words,1,1]))
```

8. We need to separate the addresses and zip codes in the reference data set, so we can feed them into the placeholders when we loop through the test set.

```
reference_addresses = [x[0] for x in reference_data]
reference_zips = np.array([[x[1] for x in reference_data]])
```

9. We need to create the sparse tensor set of reference addresses using our prior defined function.

```
sparse_ref_set = sparse_from_word_vec(reference_addresses)
```

10. Now we can loop through each entry of the test set and return the index of the reference set that it is the closest to. We print off both test and reference for each entry. As you can see, we have great results on this generated data set.

```

for i in range(n):
    test_address_entry = test_data[i][0]
    test_zip_entry = [[test_data[i][1]]]

    # Create sparse address vectors
    test_address_repeated = [test_address_entry] * n
    sparse_test_set = sparse_from_word_vec(test_address_repeated)

    feeedict={test_address: sparse_test_set,
              test_zip: test_zip_entry,
              ref_address: sparse_ref_set,
              ref_zip: reference_zips}
    best_match = sess.run(top_match_index, feed_dict=feeedict)
    best_street = reference_addresses[best_match]
    [best_zip] = reference_zips[0][best_match]
    [[test_zip_]] = test_zip_entry
    print('Address: ' + str(test_address_entry) + ', ' + str(test_
zip_))
    print('Match  : ' + str(best_street) + ', ' + str(best_zip))

Address: 8659 baker ln, 65463
Match   : 8659 baker ln, 65463
Address: 1048 eanal ln, 65681
Match   : 1048 canal ln, 65681
Address: 1756 vaker st, 65983
Match   : 1756 baker st, 65983
Address: 900 abbjy pass, 65983
Match   : 900 abbey pass, 65983
Address: 5025 canal rd, 65463
Match   : 5025 canal rd, 65463
Address: 6814 elh st, 65154
Match   : 6814 elm st, 65154
Address: 3057 cagal ave, 65463
Match   : 3057 canal ave, 65463
Address: 7776 iaker ln, 65681
Match   : 7776 baker ln, 65681
Address: 5167 caker rd, 65154
Match   : 5167 baker rd, 65154
Address: 8765 donnor st, 65154
Match   : 8765 donner st, 65154

```

How it works

One of the hard things to figure out in address matching problems like this is the value of the weights and how to scale the distances. This may take some exploration and insight into the data itself. Also when dealing with addresses we may consider different components than we did here. We may consider the street number a separate component from the street address, or even have other components like city and state. When dealing with numerical address components, know that they can be treated as numbers (with a numerical distance) or like characters (with an edit distance). It is up to you to choose how. Know that we might consider using an edit distance with the zip code if we think that typos in the zip code come from human entry and not, say, computer mapping errors.

To get a feel for how typos affect the results, we encourage the reader to change the typo function to make more typos or more frequent typos and increase the data set size to see how well this algorithm works.

Using Nearest Neighbors for Image Recognition

Getting ready

Nearest neighbors can also be used for image recognition. The 'Hello World' of image recognition data sets is the MNIST handwritten digit data set. Since we will be using this data set for various neural network image recognition algorithms in later chapters, it will be great to compare the results to a non-neural network algorithm.

The MNIST digit data set is composed of thousands of labelled images that are 28x28 pixels large. Although this is considered to be a small image, it has a total of 784 pixels (or features) for the nearest neighbor algorithm. We will compute the nearest neighbor prediction for this categorical problem by considering the mode prediction of the nearest k neighbors (k=4 in this example).

How to do it

1. We start by loading the necessary libraries. Know that we will also import PIL (Python Image Library) to be able to plot a sample of the predicted outputs. And Tensorflow has a built in method to load the MNIST data set that we will use.

```
import random
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from PIL import Image
from tensorflow.examples.tutorials.mnist import input_data
```

- Now we start a graph session and load the MNIST data in a one hot encoded form.

```
sess = tf.Session()

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

- Because the MNIST data set is large and computing the distances between 784 features on tens of thousands of inputs would be computationally hard, we will sample a smaller set of images to train on. Also, we choose a test set number that is divisible by 6 only for plotting purposes, as we will plot the last batch of six images to see a sample of the results.

```
train_size = 1000
test_size = 102
rand_train_indices = np.random.choice(len(mnist.train.images),
train_size, replace=False)
rand_test_indices = np.random.choice(len(mnist.test.images), test_
size, replace=False)
x_vals_train = mnist.train.images[rand_train_indices]
x_vals_test = mnist.test.images[rand_test_indices]
y_vals_train = mnist.train.labels[rand_train_indices]
y_vals_test = mnist.test.labels[rand_test_indices]
```

- We declare our k value and batch size.

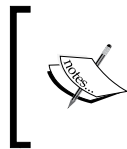
```
k = 4
batch_size=6
```

- Now we initialize our placeholders that we will feed in the graph.

```
x_data_train = tf.placeholder(shape=[None, 784], dtype=tf.float32)
x_data_test = tf.placeholder(shape=[None, 784], dtype=tf.float32)
y_target_train = tf.placeholder(shape=[None, 10], dtype=tf.
float32)
y_target_test = tf.placeholder(shape=[None, 10], dtype=tf.float32)
```

- We declare our distance metric. Here we will use the L1 metric (absolute value).

```
distance = tf.reduce_sum(tf.abs(tf.sub(x_data_train, tf.expand_
dims(x_data_test,1))), reduction_indices=2)
```



Note that we can also make our distance function use the L2 distance by using the following code instead: `distance = tf.sqrt(tf.reduce_sum(tf.square(tf.sub(x_data_train, tf.expand_dims(x_data_test,1))), reduction_indices=1))`

7. Now we find the top k images that are the closest and predict the mode. The mode will be performed on one hot encoded indices and counting which occurs the most.

```
top_k_xvals, top_k_indices = tf.nn.top_k(tf.neg(distance), k=k)
prediction_indices = tf.gather(y_target_train, top_k_indices)

count_of_predictions = tf.reduce_sum(prediction_indices,
reduction_indices=1)
prediction = tf.argmax(count_of_predictions, dimension=1)
```

8. We can now loop through our test set, compute the predictions, and store them.

```
num_loops = int(np.ceil(len(x_vals_test)/batch_size))

test_output = []
actual_vals = []
for i in range(num_loops):
    min_index = i*batch_size
    max_index = min((i+1)*batch_size, len(x_vals_train))
    x_batch = x_vals_test[min_index:max_index]
    y_batch = y_vals_test[min_index:max_index]
    predictions = sess.run(prediction, feed_dict={x_data_train: x_
vals_train, x_data_test: x_batch,
                                                    y_target_train: y_vals_
train, y_target_test: y_batch})
    test_output.extend(predictions)
    actual_vals.extend(np.argmax(y_batch, axis=1))
```

9. Now that we have saved the actual and predicted output, we can calculate the accuracy. This will change due to our random sampling of the test/train data sets, but we should end up with accuracies around 80% to 90%.

```
accuracy = sum([1./test_size for i in range(test_size) if test_
output[i]==actual_vals[i]])
print('Accuracy on test set: ' + str(accuracy))
```

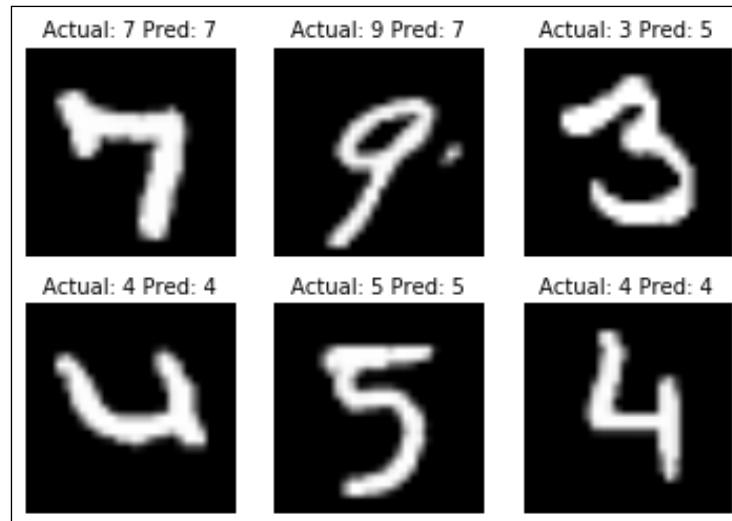
Accuracy on test set: 0.8333333333333325

Below is code to plot the last batch results.

```
actuals = np.argmax(y_batch, axis=1)

Nrows = 2
Ncols = 3
for i in range(len(actuals)):
    plt.subplot(Nrows, Ncols, i+1)
    plt.imshow(np.reshape(x_batch[i], [28,28]), cmap='Greys_r')
    plt.title('Actual: ' + str(actuals[i]) + ' Pred: ' +
str(predictions[i]),
```

```
                                fontsize=10)
frame = plt.gca()
frame.axes.get_xaxis().set_visible(False)
frame.axes.get_yaxis().set_visible(False)
```



The last batch of six images we ran our nearest neighbor prediction on.
We can see that we do not get all of the images exactly correct.

How it works

Given enough computation time and computational resources, we could have made the test and training sets bigger. This probably would have increased our accuracy. Also, this algorithm warrants further exploration on the ideal k value to choose. The k value would be chosen after a set of cross validation experiments on the data set.

There's more

In this chapter, we've explored how to use k -nearest neighbor algorithms for regression and classification. We've talked about the different usage of distance functions and how to mix them together. We encourage the reader to explore different distance metrics, weights and k -values to optimize the accuracy of these methods.

6

Neural Networks

In this chapter we will introduce neural networks and how to implement them in Tensorflow. Most of the subsequent chapters will be based on neural networks, so learning how to use them in Tensorflow is very important. We will start by introducing basic concepts of neural network and work up to multilayer networks. And the last section we will create a neural network that learns to play tic tac toe.

In this chapter, we'll cover the following recipes:

- ▶ Implementing Operational Gates
- ▶ Working with Gates and Activation Functions
- ▶ Implementing a One Layer Neural Network
- ▶ Implementing Different Layers
- ▶ Using Multi-Layer Networks
- ▶ Improving Predictions of Linear Models
- ▶ Learning to Play Tic Tac Toe

Introduction

- ▶ Neural networks are currently breaking records in tasks such as image and speech recognition, reading handwriting, understanding text, and much more. While some of these aforementioned tasks will be covered in later chapters, it is important to introduce neural networks as an easy-to-implement machine learning algorithm, so that we can expand on it later.
- ▶ The concept of a neural network has been around for decades. The only recently gained traction computationally in the past few years because we now have the computational power to train large networks.

- ▶ A neural network is basically a sequence of operations applied to a matrix of input data. These operations are usually collections of additions and multiplications followed by applications of non-linear functions. One example that we have already seen is logistic regression, the last section in the Linear Regression chapter. Logistic regression is the sum of the partial slope-feature products followed by the application of the sigmoid function, which is non-linear. Neural networks generalize this a bit more by allowing any combination of operations and non-linear functions, which includes the applications of absolute value, maximum, minimum, etc...
- ▶ The big important trick with neural networks is called, back propagation. Back propagation is a procedure that allows us to update the model variables based on the learning rate and the output of the loss function. We have been using back propagation to update our model variables in the linear regression chapter and the support vector machine chapter.
- ▶ Another important feature to take note of in neural networks is the non-linear activation function. Since most neural networks are just combinations of addition and multiplication operations, they will not be able to model non-linear data sets. To combat this, we use non-linear activation functions in the neural networks. This will allow the neural network to adapt to most non-linear situations.
- ▶ It is important to remember that like most of the algorithms we have seen so far, neural networks are sensitive to the hyper-parameters that we choose. In this chapter, we will see the impact of different learning rates, loss functions, and optimization procedures.

There are more resources for learning about neural networks that are more in depth and detailed.

The seminal paper describing back propagation is "Efficient Back Prop" by Yann LeCun et. al. The PDF is located here: <http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>

There is an online book called "Neural Networks and Deep Learning" by Michael Nielsen, located here: <http://neuralnetworksanddeeplearning.com/>

For a more pragmatic approach and introduction to neural networks, Andrej Karpathy has written a great summary and javascript examples called "A Hacker's Guide to Neural Networks". The write up is located here: <http://karpathy.github.io/neuralnets/>

Another site that summarizes some good notes on deep learning is called "Deep Learning for Beginners" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville. This web page can be found here: <http://randomekek.github.io/deep/deeplearning.html>



Implementing Operational Gates

One of the most fundamental concepts of neural networks is implementing an operation as an operational gate. In this section we will first implement a multiplication operation as a gate and then two nested operations as a gate.

Getting ready

The first operational gate we will implement looks like $f(x) = a \cdot x$. To optimize this gate, we declare the 'a' input as a variable and the 'x' input as a placeholder. This means that Tensorflow will try to change the 'a' value and not the 'x' value. We will create the loss function as the difference between the output and the target value, 50.

The second, nested operational gate will be $f(x) = a \cdot x + b$. Again, we will declare 'a' and 'b' as variables and 'x' as a placeholder. We optimize the output toward the target value of 50 again. The interesting thing to note is that the solution for this second example is not unique. There are many combinations of model variables that will allow the output to be 50. With neural networks, we do not care as much for the values of the intermediate model variables, but place more emphasis on the desired output.

If we think of the operations as operational gates on our computational graph, here is a figure depicting the two examples.

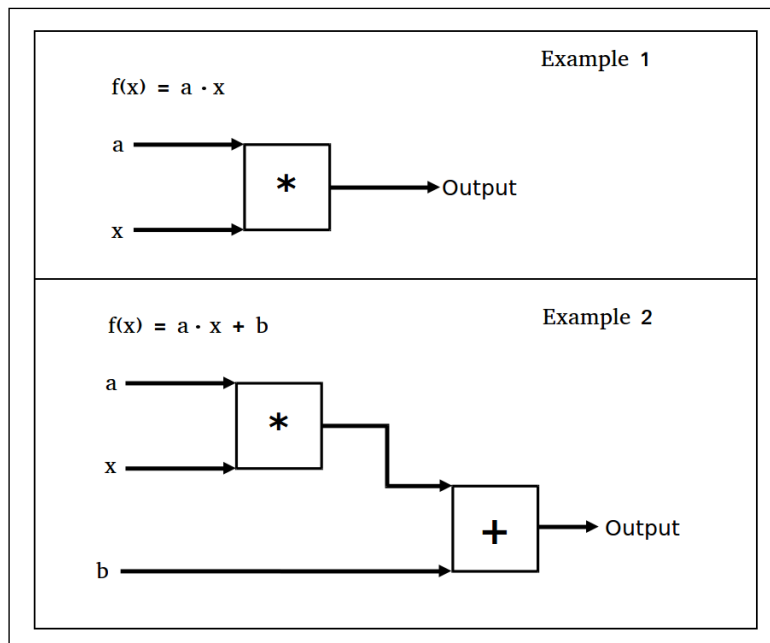


Figure of the two operational gate examples in this section.

How to do it

Here is how we implement the first operational gate, $f(x) = a \cdot x$ in Tensorflow and train the output towards the value of 50.

1. We start off by loading Tensorflow and creating a graph session.

```
import tensorflow as tf
sess = tf.Session()
```
2. Now, we declare our model variable, input data, and placeholder. We make our input data equal to the value 5, so that the multiplication factor to get 50 will be 10. (i.e. $5 \cdot 10 = 50$)

```
a = tf.Variable(tf.constant(4.))
x_val = 5.
x_data = tf.placeholder(dtype=tf.float32)
```
3. Next we add the operation to our computational graph.

```
multiplication = tf.mul(a, x_data)
```
4. We will declare the loss function as the L2 distance between the output and the desired target value of 50.

```
loss = tf.square(tf.sub(multiplication, 50.))
```
5. Now we initialize our model variable and declare our optimizing algorithm as the standard gradient descent.

```
init = tf.initialize_all_variables()
sess.run(init)
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)
```
6. We can now optimize our model output towards the desired value of 50. We do this by continually feeding in the input value of 5 and back propagating the loss to update the model variable towards the value of 10.

```
print('Optimizing a Multiplication Gate Output to 50.')
for i in range(10):
    sess.run(train_step, feed_dict={x_data: x_val})
    a_val = sess.run(a)
    mult_output = sess.run(multiplication, feed_dict={x_data: x_val})
print(str(a_val) + ' * ' + str(x_val) + ' = ' + str(mult_output))

Optimizing a Multiplication Gate Output to 50.
7.0 * 5.0 = 35.0
8.5 * 5.0 = 42.5
9.25 * 5.0 = 46.25
```

```

9.625 * 5.0 = 48.125
9.8125 * 5.0 = 49.0625
9.90625 * 5.0 = 49.5312
9.95312 * 5.0 = 49.7656
9.97656 * 5.0 = 49.8828
9.98828 * 5.0 = 49.9414
9.99414 * 5.0 = 49.9707

```

7. Next, we will do the same with a two nested operations, $f(x) = a \cdot x + b$.
8. We will start the exact same way as the above example, except we initialize two model variables, 'a' and 'b'.

```

from tensorflow.python.framework import ops
ops.reset_default_graph()
sess = tf.Session()

a = tf.Variable(tf.constant(1.))
b = tf.Variable(tf.constant(1.))
x_val = 5.
x_data = tf.placeholder(dtype=tf.float32)

two_gate = tf.add(tf.mul(a, x_data), b)

loss = tf.square(tf.sub(two_gate, 50.))

my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)

init = tf.initialize_all_variables()
sess.run(init)

```

9. We now optimize the model variables to train the output towards the target value of 50.

```

print('\nOptimizing Two Gate Output to 50.')
for i in range(10):
    sess.run(train_step, feed_dict={x_data: x_val})
    a_val, b_val = (sess.run(a), sess.run(b))
    two_gate_output = sess.run(two_gate, feed_dict={x_data: x_val})
    print(str(a_val) + ' * ' + str(x_val) + ' + ' + str(b_val) + ' = ' + str(two_gate_output))

```

```

Optimizing Two Gate Output to 50.
5.4 * 5.0 + 1.88 = 28.88
7.512 * 5.0 + 2.3024 = 39.8624

```



```
8.52576 * 5.0 + 2.50515 = 45.134
9.01236 * 5.0 + 2.60247 = 47.6643
9.24593 * 5.0 + 2.64919 = 48.8789
9.35805 * 5.0 + 2.67161 = 49.4619
9.41186 * 5.0 + 2.68237 = 49.7417
9.43769 * 5.0 + 2.68754 = 49.876
9.45009 * 5.0 + 2.69002 = 49.9405
9.45605 * 5.0 + 2.69121 = 49.9714
```



It is important to note here again that the solution to the second example is not unique. This does not matter as much in neural networks, as all parameters are adjusted towards reducing the loss. The final solution here will depend on the initial values of 'a' and 'b'. If these were randomly initialized, instead of to the value of 1, we would see different ending values for the model variables for each run.

How it works

We achieved optimizing a computational graph via Tensorflow's implicit back propagation. Tensorflow keeps track of our model's operations and variable values and makes adjustments in respect to our optimization algorithm specification and the output of the loss function.

We can keep expanding the operational gates, while keeping track of which inputs are variables and which inputs are data. This is important to keep track of, because Tensorflow will change all variables to minimize the loss, but not the data, which is declared as placeholders.

The implicit ability to keep track of the computational graph and update the model variables automatically with every training step is one of the great features of Tensorflow and what makes it so powerful.

Working with Gates and Activation Functions

Getting ready

In this section we will compare and contrast two different activation functions, the sigmoid and the ReLU (rectified linear unit). Recall that the two functions are given by the below equations.

$$\text{sigmoid}(x) = \frac{1}{1 + e^x}$$

$$\text{ReLU}(x) = \max(0, x)$$

In this example we will create two 1-layer neural networks with the same structure except one will feed through the sigmoid activation and one will feed through the ReLU activation. The loss function will be governed by the L2 distance from the value 0.75. We will randomly pull batch data from a normal distribution (Normal(mean=2, sd=0.1)), and optimize the output towards 0.75.

How to do it

1. We start by loading the necessary libraries and initializing a graph. This is also a good point to bring up how to set a random seed with Tensorflow. Since we will be using a random number generator from numpy and Tensorflow, we need to set a random seed for both. With the same random seeds set, we should be able to replicate

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
sess = tf.Session()
tf.set_random_seed(5)
np.random.seed(42)
```

2. Now we declare our batch size, model variables, data and a placeholder for feeding the data in. Our computational graph will consist of feeding in our normally distributed data into two similar neural networks that differ only by the activation function at the end.

```
batch_size = 50

a1 = tf.Variable(tf.random_normal(shape=[1,1]))
b1 = tf.Variable(tf.random_uniform(shape=[1,1]))
a2 = tf.Variable(tf.random_normal(shape=[1,1]))
b2 = tf.Variable(tf.random_uniform(shape=[1,1]))
x = np.random.normal(2, 0.1, 500)
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

3. Next we declare our two models, the sigmoid activation model and the ReLU activation model.

```
sigmoid_activation = tf.sigmoid(tf.add(tf.matmul(x_data, a1), b1))
relu_activation = tf.nn.relu(tf.add(tf.matmul(x_data, a2), b2))
The loss functions will be the average L2 norm between the model
output and the value of 0.75.
loss1 = tf.reduce_mean(tf.square(tf.sub(sigmoid_activation,
0.75)))
loss2 = tf.reduce_mean(tf.square(tf.sub(relu_activation, 0.75)))
Now we declare our optimization algorithm and initialize our
variables.
```

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step_sigmoid = my_opt.minimize(loss1)
train_step_relu = my_opt.minimize(loss2)

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

4. Now we loop through our training for 750 iterations for both models. We will also save the loss output and the activation output values for plotting after.

```
loss_vec_sigmoid = []
loss_vec_relu = []
activation_sigmoid = []
activation_relu = []
for i in range(750):
    rand_indices = np.random.choice(len(x), size=batch_size)
    x_vals = np.transpose([x[rand_indices]])
    sess.run(train_step_sigmoid, feed_dict={x_data: x_vals})
    sess.run(train_step_relu, feed_dict={x_data: x_vals})

    loss_vec_sigmoid.append(sess.run(loss1, feed_dict={x_data: x_
vals}))
    loss_vec_relu.append(sess.run(loss2, feed_dict={x_data: x_
vals}))

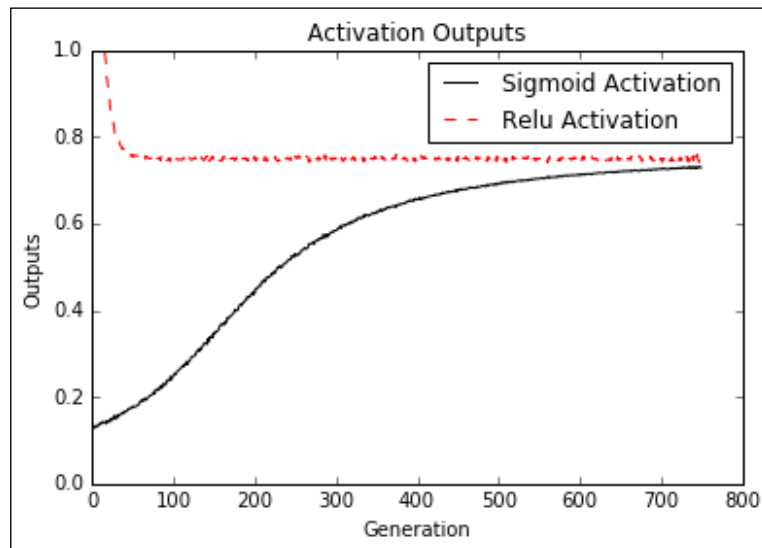
    activation_sigmoid.append(np.mean(sess.run(sigmoid_activation,
feed_dict={x_data: x_vals})))
    activation_relu.append(np.mean(sess.run(relu_activation, feed_
dict={x_data: x_vals})))
```

5. The following is code to plot the loss and the activation outputs.

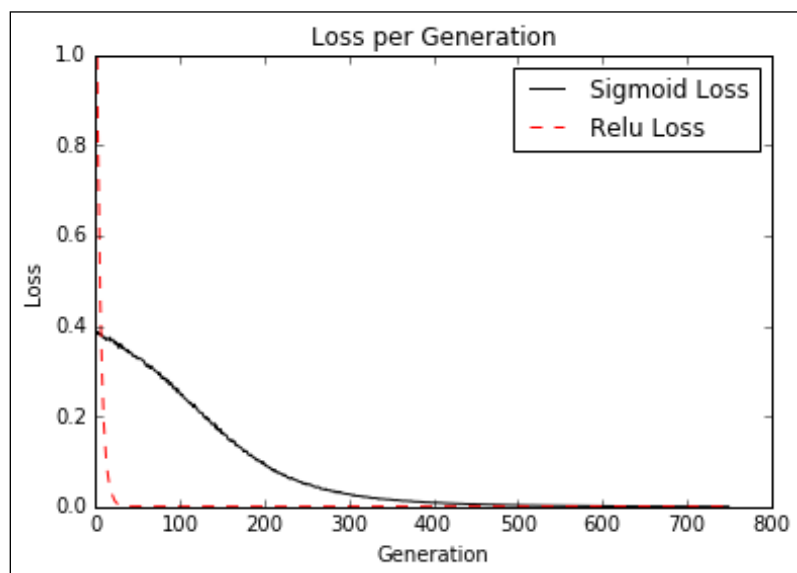
```
plt.plot(activation_sigmoid, 'k-', label='Sigmoid Activation')
plt.plot(activation_relu, 'r--', label='Relu Activation')
plt.ylim([0, 1.0])
plt.title('Activation Outputs')
plt.xlabel('Generation')
plt.ylabel('Outputs')
plt.legend(loc='upper right')
plt.show()

plt.plot(loss_vec_sigmoid, 'k-', label='Sigmoid Loss')
plt.plot(loss_vec_relu, 'r--', label='Relu Loss')
plt.ylim([0, 1.0])
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
```

```
plt.legend(loc='upper right')  
plt.show()
```



The two neural networks with similar architecture and target (0.75) with two different activation functions, sigmoid and ReLU. It is important to notice how much quicker the ReLU activation network converges to the desired target of 0.75 than the Sigmoid.



This figure depicts the loss value of the sigmoid and the ReLU activation networks. Notice how extreme the ReLU loss is at the beginning of the iterations.

How it works

Because of the form of the ReLU activation function, it returns the value of zero much more often than the sigmoid function. We consider this behaviour as a type of sparsity. This sparsity results in a speed up of convergence, but a loss of controlled gradients. On the other hand, the sigmoid function has very well controlled gradients and does not risk the extreme values that the ReLU activation does.

Activation function	Advantages	Disadvantages
Sigmoid	Less Extreme Outputs	Slower Convergence
ReLU	Converges Quicker	Extreme Output Values Possible

There's more

In this section we compared the ReLU activation function and the sigmoid activation for neural networks. There are many other activation functions that are commonly used for neural networks. But most fall into either one of two categories. The first category are functions that are shaped like the sigmoid function (arctan, hypertangent, heavyside step, etc...), and the second category are functions that are shaped like the ReLU function (softplus, leaky ReLU, etc...). Most of what was discussed in this section about comparing the two functions will hold true for activations in either category. But it is important to note that the choice of the activation function has a big impact on the convergence and outputs of the neural networks.

Implementing a One Layer Neural Network

Getting ready

In this section we will implement a neural network with one hidden layer. It will be important to understand that a fully connected neural network is based mostly on matrix multiplication. As such, the dimensions of the data and matrix are very important to get lined up correctly.

Since this is a regression problem, we will use the mean squared error as the loss function.

How to do it

1. To create the computational graph, we start by loading the necessary libraries.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
```

- Now we load the iris data and store the pedal length as the target value. Then we start a graph session.

```
iris = datasets.load_iris()
x_vals = np.array([x[0:3] for x in iris.data])
y_vals = np.array([x[3] for x in iris.data])
sess = tf.Session()
```

- Since the dataset is of a smaller size, we want to set a seed to make the results reproducible.

```
seed = 2
tf.set_random_seed(seed)
np.random.seed(seed)
```

- To prepare the data, we create a 80-20 train-test split and normalize the x features to be between 0 and 1 via min-max scaling.

```
train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

```
def normalize_cols(m):
    col_max = m.max(axis=0)
    col_min = m.min(axis=0)
    return (m-col_min) / (col_max - col_min)
```

```
x_vals_train = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test = np.nan_to_num(normalize_cols(x_vals_test))
```

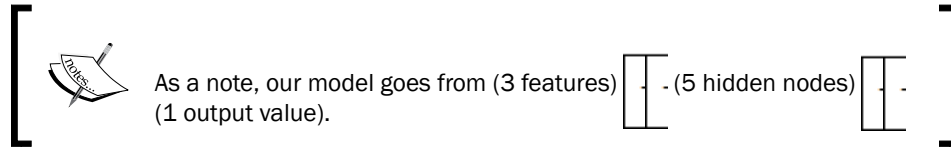
- Now we declare the batch size and placeholders for the data and target.

```
batch_size = 50
x_data = tf.placeholder(shape=[None, 3], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

- The important part is to declare our model variables with the appropriate shape. We can declare the size of our hidden layer to be any size we wish, here we set it to have 5 hidden nodes.

```
hidden_layer_nodes = 5
A1 = tf.Variable(tf.random_normal(shape=[3,hidden_layer_nodes]))
b1 = tf.Variable(tf.random_normal(shape=[hidden_layer_nodes]))
A2 = tf.Variable(tf.random_normal(shape=[hidden_layer_nodes,1]))
b2 = tf.Variable(tf.random_normal(shape=[1]))
```

7. We now declare our model in two steps. The first step will be creating the hidden layer output and the second will be creating the final output of the model.



```
hidden_output = tf.nn.relu(tf.add(tf.matmul(x_data, A1), b1))
final_output = tf.nn.relu(tf.add(tf.matmul(hidden_output, A2),
b2))
```

8. Here is our mean squared error as a loss function.

```
loss = tf.reduce_mean(tf.square(y_target - final_output))
```

9. Now we declare our optimizing algorithm and initialize our variables.

```
my_opt = tf.train.GradientDescentOptimizer(0.005)
train_step = my_opt.minimize(loss)
```

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

10. Now we loop through our training iterations. We also initialize two lists that we can store our train and test loss. In every loop we also want to randomly select a batch from the training data for fitting the model.

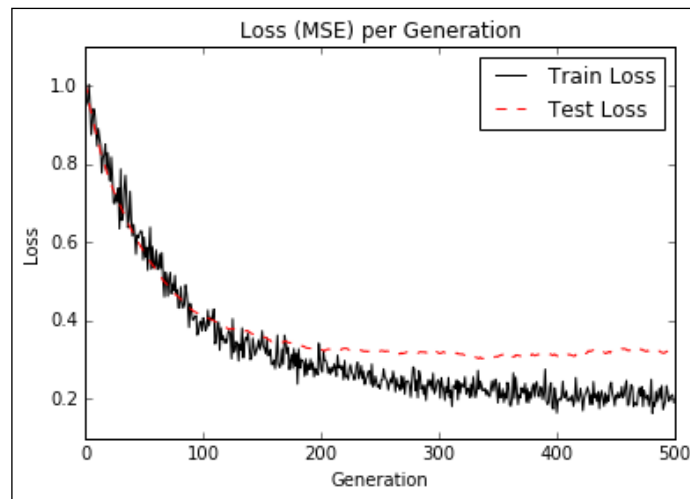
```
loss_vec = []
test_loss = []
for i in range(500):
    rand_index = np.random.choice(len(x_vals_train), size=batch_
size)
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
    loss_vec.append(np.sqrt(temp_loss))

    test_temp_loss = sess.run(loss, feed_dict={x_data: x_vals_
test, y_target: np.transpose([y_vals_test])})
    test_loss.append(np.sqrt(test_temp_loss))
    if (i+1)%50==0:
        print('Generation: ' + str(i+1) + '. Loss = ' + str(temp_
loss))
```

11. And here is how we can plot the losses with matplotlib.

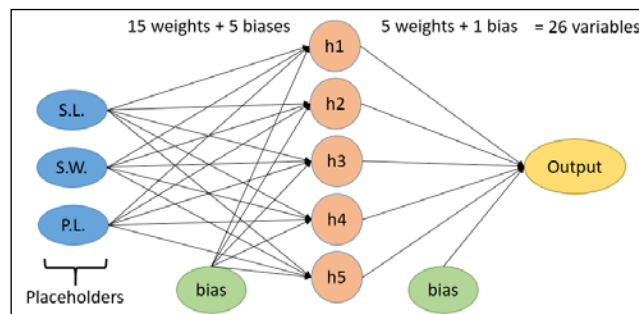
```
plt.plot(loss_vec, 'k-', label='Train Loss')
plt.plot(test_loss, 'r--', label='Test Loss')
plt.title('Loss (MSE) per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```



We plot the loss (MSE) of the train and test set. Notice that we are slightly overfitting the model after 200 generations, as the test MSE does not drop any further, but the training MSE does continue to drop.

How it works

If we want to view our model as a neural network diagram, we can visualize it as the following figure.



Here is a visualization of our neural network that has five nodes in the hidden layer. We are feeding in three values, the sepal length (S.L), the sepal width (S.W.), and the pedal length (P.L.). The target will be the petal width. In total, there will be 26 total variables in the model.

There's more

Note that we can identify when the model starts overfitting on the training data from viewing the loss function on the test and train sets. We can also see that the train set loss is much less smooth than the test set. This is because of two things, the first is that we are using a smaller batch size than the test set, although not by much. The other cause of this is due to the fact that we are training on the train set and the test set does not impact the variables of the model.

Implementing Different Layers

Getting ready

We have explored how to connect between data inputs and a fully connected hidden layer. There are more types of layers that are built-in function inside Tensorflow. The most popular layers that are used are convolutional layers and maxpool layers. We will show how to create and use such layers with input data and with fully connected data. First we will look at how to use such layers on one-dimensional data and then on two-dimensional data.

How to do it

We will first look at one-dimensional data. We generate a random array of data for this task.

1. We start by loading the libraries we need and starting a graph session.

```
import tensorflow as tf
import numpy as np
sess = tf.Session()
```
2. Now we initialize our data (numpy array of length 25) and create the placeholder that we will feed it through.

```
data_size = 25
data_1d = np.random.normal(size=data_size)
x_input_1d = tf.placeholder(dtype=tf.float32, shape=[data_size])
```
3. We now define a function that will make a convolutional layer. Then we declare a random filter and create the convolutional layer.



Note that many of Tensorflow's layer functions are designed to deal with 4D data (4D = [batch size, width, height, channels]). We will need to modify our input data and the output data to extend or collapse the extra dimensions needed. For our example data, we have batch size of 1, width of 1, height of 25, and a channel size of 1. To expand dimensions, we use the 'expand_dims()' function, and to collapse dimensions, we use the 'squeeze()' function.

```
def conv_layer_1d(input_1d, my_filter):
    input_2d = tf.expand_dims(input_1d, 0)
    input_3d = tf.expand_dims(input_2d, 0)
    input_4d = tf.expand_dims(input_3d, 3)

    convolution_output = tf.nn.conv2d(input_4d, filter=my_filter,
                                       strides=[1,1,1,1], padding="VALID")

    conv_output_1d = tf.squeeze(convolution_output)
    return(conv_output_1d)

my_filter = tf.Variable(tf.random_normal(shape=[1,5,1,1]))

my_convolution_output = conv_layer_1d(x_input_1d, my_filter)
```

4. Tensorflow's activation functions will act elementwise by default. This means we just have to call our activation function on the layer of interest. We do this by creating an activation function and then initializing it on the graph.

```
def activation(input_1d):
    return(tf.nn.relu(input_1d))

my_activation_output = activation(my_convolution_output)
```

5. Now we declare a maxpool layer function. This function will create a maxpool on a moving window across our one-dimensional vector. For this example we will initialize it to have a width of 5.



Tensorflow's maxpool arguments are very similarly to the convolutional layer. While it does not have a filter, it does have a size, stride, and padding option. Since we have a window of 5 with valid padding

(no zero padding), then our output array will have 4 or $2 \cdot \text{floor}(5/2)$ less entries.

```
def max_pool(input_1d, width):
    input_2d = tf.expand_dims(input_1d, 0)
    input_3d = tf.expand_dims(input_2d, 0)
    input_4d = tf.expand_dims(input_3d, 3)

    pool_output = tf.nn.max_pool(input_4d, ksize=[1, 1, width, 1],
                                  strides=[1, 1, 1, 1],
                                  padding='VALID')
```

```
pool_output_1d = tf.squeeze(pool_output)
return(pool_output_1d)
```

```
my_maxpool_output = max_pool(my_activation_output, width=5)
```

6. The final layer that we will connect is the fully connected layer. We want to create a versatile function that inputs a 1D array and outputs the number of values indicated. Also remember that to do matrix multiplication with a 1D array, we must expand the dimensions into 2D.

```
def fully_connected(input_layer, num_outputs):
    weight_shape = tf.squeeze(tf.pack([tf.shape(input_layer),
    [num_outputs]]))

    weight = tf.random_normal(weight_shape, stddev=0.1)
    bias = tf.random_normal(shape=[num_outputs])

    input_layer_2d = tf.expand_dims(input_layer, 0)

    full_output = tf.add(tf.matmul(input_layer_2d, weight), bias)

    full_output_1d = tf.squeeze(full_output)
    return(full_output_1d)
```

```
my_full_output = fully_connected(my_maxpool_output, 5)
```

7. Now we initialize all the variables and run the graph and print the outputs of each layer.

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

feed_dict = {x_input_1d: data_1d}

# Convolution Output
print('Input = array of length 25')
print('Convolution w/filter, length = 5, stride size = 1, results
in an array of length 21:')
print(sess.run(my_convolution_output, feed_dict=feed_dict))

# Activation Output
print('\nInput = the above array of length 21')
print('ReLU element wise returns the array of length 21:')
print(sess.run(my_activation_output, feed_dict=feed_dict))

# Maxpool Output
print('\nInput = the above array of length 21')
```

```
print('MaxPool, window length = 5, stride size = 1, results in the
array of length 17:')
print(sess.run(my_maxpool_output, feed_dict=feed_dict))
```

```
# Fully Connected Output
print('\nInput = the above array of length 17')
print('Fully connected layer on all four rows with five outputs:')
print(sess.run(my_full_output, feed_dict=feed_dict))
```

```
Input = array of length 25
Convolution w/filter, length = 5, stride size = 1, results in an
array of length 21:
```

```
[-0.91608119  1.53731811 -0.7954089   0.5041104   1.88933098
 -1.81099761  0.56695032  1.17945457 -0.66252393 -1.90287709
  0.87184119  0.84611893 -5.25024986 -0.05473572  2.19293165
 -4.47577858 -1.71364677  3.96857905 -2.0452652  -1.86647367
 -0.12697852]
```


```
Input = the above array of length 21
ReLU element wise returns the array of length 21:
```

```
[ 0.          1.53731811  0.          0.5041104   1.88933098
  0.          0.          1.17945457  0.          0.
  0.87184119  0.84611893  0.          0.          2.19293165
  0.          0.          3.96857905  0.          0.
  0.          ]
```

```
Input = the above array of length 21
MaxPool, window length = 5, stride size = 1, results in the array
of length 17:
```

```
[ 1.88933098  1.88933098  1.88933098  1.88933098  1.88933098
  1.17945457  1.17945457  1.17945457  0.87184119  0.87184119
  2.19293165  2.19293165  2.19293165  3.96857905  3.96857905
  3.96857905  3.96857905]
```

```
Input = the above array of length 17
Fully connected layer on all four rows with five outputs:
[ 1.23588216 -0.42116445  1.44521213  1.40348077 -0.79607368]
```


 One-dimensional data is very important to consider for neural networks. Time series, signal processing, and some text embeddings are considered to be one-dimensional and frequently used in neural networks.

We will now consider the same types of layers in an equivalent order but for two-dimensional data.

1. We start by clearing and resetting the computational graph.

```
ops.reset_default_graph()
sess = tf.Session()
```

2. We initialize our input array to be a 10x10 matrix. Then we initialize a placeholder for the graph with the same shape.

```
data_size = [10,10]
data_2d = np.random.normal(size=data_size)

x_input_2d = tf.placeholder(dtype=tf.float32, shape=data_size)
```

3. Just like in the one-dimensional example, we declare a convolutional layer function. Since our data has a height and width already, we just need to expand it in two dimensions (a batch size of one, and a channel size of one) so that we can operate on it with the `conv2d()` function. For the filter, we will use a random 2x2 filter, stride two in both directions, and use valid padding (no zero padding). Because our input matrix is 10x10, our convolutional output will be 5x5.

```
def conv_layer_2d(input_2d, my_filter):
    input_3d = tf.expand_dims(input_2d, 0)
    input_4d = tf.expand_dims(input_3d, 3)

    convolution_output = tf.nn.conv2d(input_4d, filter=my_filter,
    strides=[1,2,2,1], padding="VALID")

    conv_output_2d = tf.squeeze(convolution_output)
    return(conv_output_2d)
```

```
my_filter = tf.Variable(tf.random_normal(shape=[2,2,1,1]))
```

```
my_convolution_output = conv_layer_2d(x_input_2d, my_filter)
```

4. The activation function works on an element-wise basis, so now we can create an activation operation and initialize it on the graph.

```
def activation(input_2d):
    return(tf.nn.relu(input_2d))

my_activation_output = activation(my_convolution_output)
```

5. Our maxpool layer is very similar to the one-dimensional case except we have to declare a width and height for the maxpool window. And just like our convolutional 2d layer, we only have to expand our into in two dimensions this time.

```
def max_pool(input_2d, width, height):
    input_3d = tf.expand_dims(input_2d, 0)
    input_4d = tf.expand_dims(input_3d, 3)

    pool_output = tf.nn.max_pool(input_4d, ksize=[1, height,
width, 1], strides=[1, 1, 1, 1], padding='VALID')

    pool_output_2d = tf.squeeze(pool_output)
    return (pool_output_2d)

my_maxpool_output = max_pool(my_activation_output, width=2,
height=2)
```

6. Our fully connected layer is very similar to the one dimensional output. We should also note here that the 2D input to this layer is considered as one object, so we want each of the entries connected to each of the outputs. In order to accomplish this, we fully flatten out the two-dimensional matrix and then expand it for matrix multiplication.

```
def fully_connected(input_layer, num_outputs):
    flat_input = tf.reshape(input_layer, [-1])
    weight_shape = tf.squeeze(tf.pack([tf.shape(flat_input), [num_
outputs]]))
    weight = tf.random_normal(weight_shape, stddev=0.1)
    bias = tf.random_normal(shape=[num_outputs])

    input_2d = tf.expand_dims(flat_input, 0)

    full_output = tf.add(tf.matmul(input_2d, weight), bias)

    full_output_2d = tf.squeeze(full_output)
    return (full_output_2d)

my_full_output = fully_connected(my_maxpool_output, 5)
```

7. We now initialize our variables and create a feed dictionary for our operations.

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

feed_dict = {x_input_2d: data_2d}
```

8. And here is how we can see the outputs for each of the layers.

```
# Convolution Output
print('Input = [10 X 10] array')
print('2x2 Convolution, stride size = [2x2], results in the [5x5] array:')
print(sess.run(my_convolution_output, feed_dict=feed_dict))

# Activation Output
print('\nInput = the above [5x5] array')
print('ReLU element wise returns the [5x5] array:')
print(sess.run(my_activation_output, feed_dict=feed_dict))

# Max Pool Output
print('\nInput = the above [5x5] array')
print('MaxPool, stride size = [1x1], results in the [4x4] array:')
print(sess.run(my_maxpool_output, feed_dict=feed_dict))

# Fully Connected Output
print('\nInput = the above [4x4] array')
print('Fully connected layer on all four rows with five outputs:')
print(sess.run(my_full_output, feed_dict=feed_dict))

Input = [10 X 10] array
2x2 Convolution, stride size = [2x2], results in the [5x5] array:
[[ 0.37630892 -1.41018617 -2.58821273 -0.32302785  1.18970704]
 [-4.33685207  1.97415686  1.0844903  -1.18965471  0.84643292]
 [ 5.23706436  2.46556497 -0.95119286  1.17715418  4.1117816 ]
 [ 5.86972761  1.2213701  1.59536231  2.66231227  2.28650784]
 [-0.88964868 -2.75502229  4.3449688  2.67776585 -2.23714781]]

Input = the above [5x5] array
ReLU element wise returns the [5x5] array:
[[ 0.37630892  0.          0.          0.          1.18970704]
 [ 0.          1.97415686  1.0844903  0.          0.84643292]
 [ 5.23706436  2.46556497  0.          1.17715418  4.1117816 ]
 [ 5.86972761  1.2213701  1.59536231  2.66231227  2.28650784]
 [ 0.          0.          4.3449688  2.67776585  0.          ]]

Input = the above [5x5] array
MaxPool, stride size = [1x1], results in the [4x4] array:
[[ 1.97415686  1.97415686  1.0844903  1.18970704]
 [ 5.23706436  2.46556497  1.17715418  4.1117816 ]
 [ 5.86972761  2.46556497  2.66231227  4.1117816 ]
```

```
[ 5.86972761  4.3449688  4.3449688  2.67776585]]
```

Input = the above [4x4] array

Fully connected layer on all four rows with five outputs:

```
[-0.6154139 -1.96987963 -1.88811922  0.20010889  0.32519674]
```

How it works

We can now see how to use the convolutional and maxpool layers in Tensorflow with one-dimensional and two-dimensional data. Regardless of the shape of the input, we ended up with the same size output. This is important to illustrate the flexibility of neural network layers. This section should also impress upon us again the importance of shapes and sizes in the neural network operations.

Using a Multi-Layer Neural Network

Getting ready

Now that we know how to create neural networks and work with layers, we will apply this methodology towards predicting the birthweight in the low-birthweight data set. We will create a neural network with three hidden layers. The low-birthweight data set includes the actual birthweight and an indicator variable if the birthweight is above or below 2500 grams. In this example, we will make the target the actual birthweight (regression) and then see what the accuracy is on the classification at the end (see if our model can identify <2500 grams).

How to do it

1. First we start by loading the libraries and initializing our computational graph.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import requests
import numpy as np
```

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

2. Now we load the data from the website using the requests module. After this we split the data into the features of interest and the target value.

```
birthdata_url = 'https://www.umass.edu/statdata/statdata/data/lowbwt.dat'
birth_file = requests.get(birthdata_url)
birth_data = birth_file.text.split('\r\n')[5:]
birth_header = [x for x in birth_data[0].split(' ') if len(x) >= 1]
```



```
birth_data = [[float(x) for x in y.split(' ') if len(x)>=1] for y
in birth_data[1:] if len(y)>=1]

y_vals = np.array([x[10] for x in birth_data])

cols_of_interest = ['AGE', 'LWT', 'RACE', 'SMOKE', 'PTL', 'HT',
'UI', 'FTV']
x_vals = np.array([x[ix] for ix, feature in enumerate(birth_
header) if feature in cols_of_interest] for x in birth_data])
```

3. To help with repeatability, we set a seed for both numpy and tensorflow. Then we declare our batch size.

```
seed = 3
tf.set_random_seed(seed)
np.random.seed(seed)
```

```
batch_size = 100
```

4. Next we split the data into an 80-20 train-test split. After this we will normalize our input features to be between zero and one with a min-max scaling.

```
train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

```
def normalize_cols(m):
    col_max = m.max(axis=0)
    col_min = m.min(axis=0)
    return (m-col_min) / (col_max - col_min)
```

```
x_vals_train = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test = np.nan_to_num(normalize_cols(x_vals_test))
```

Since we will have multiple layers that have similar initialized variables, we will create a function to initialize both the weights and the bias.

```
def init_weight(shape, st_dev):
    weight = tf.Variable(tf.random_normal(shape, stddev=st_dev))
    return(weight)
```

```
def init_bias(shape, st_dev):
    bias = tf.Variable(tf.random_normal(shape, stddev=st_dev))
    return(bias)
```

5. We initialize our placeholders next. There will be eight input features and one output, the birthweight in grams.

```
x_data = tf.placeholder(shape=[None, 8], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

6. The fully connected layer will be used three times for all three hidden layers. To prevent repeated code, we create a layer function to use when we initialize our model.

```
def fully_connected(input_layer, weights, biases):
    layer = tf.add(tf.matmul(input_layer, weights), biases)
    return(tf.nn.relu(layer))
```

7. We now create our model. For each layer (and output layer), we will initialize a weight matrix, bias matrix, and the fully connected layer. For this example, we will use hidden layers of sizes 25, 10, and 3.



Our model that we are using will have 522 variables to fit. To arrive at this number, we see that between the data and first hidden layer we have $8 \times 25 + 25 = 225$ variables. If we continue in this way and add them up we will have $225 + 260 + 33 + 4 = 522$ variables. This is significantly larger than the nine variables that we used in the logistic regression model on this data.

```
#-----Create second layer (25 hidden nodes)-----
weight_1 = init_weight(shape=[8, 25], st_dev=10.0)
bias_1 = init_bias(shape=[25], st_dev=10.0)
layer_1 = fully_connected(x_data, weight_1, bias_1)

#-----Create second layer (10 hidden nodes)-----
weight_2 = init_weight(shape=[25, 10], st_dev=10.0)
bias_2 = init_bias(shape=[10], st_dev=10.0)
layer_2 = fully_connected(layer_1, weight_2, bias_2)

#-----Create third layer (3 hidden nodes)-----
weight_3 = init_weight(shape=[10, 3], st_dev=10.0)
bias_3 = init_bias(shape=[3], st_dev=10.0)
layer_3 = fully_connected(layer_2, weight_3, bias_3)

#-----Create output layer (1 output value)-----
```

```
weight_4 = init_weight(shape=[3, 1], st_dev=10.0)
bias_4 = init_bias(shape=[1], st_dev=10.0)
final_output = fully_connected(layer_3, weight_4, bias_4)
```

8. We now use the L1 loss function (absolute value), declare our optimizer (Adam optimization), and initialize our variables.

```
loss = tf.reduce_mean(tf.abs(y_target - final_output))
```

```
my_opt = tf.train.AdamOptimizer(0.05)
train_step = my_opt.minimize(loss)
```

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

9. Next we will train our model for 200 generations. We also include code that will store the train and test loss, select a random batch size, and print the status every 25 generations.

```
loss_vec = []
test_loss = []
for i in range(200):
    rand_index = np.random.choice(len(x_vals_train), size=batch_
size)
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})

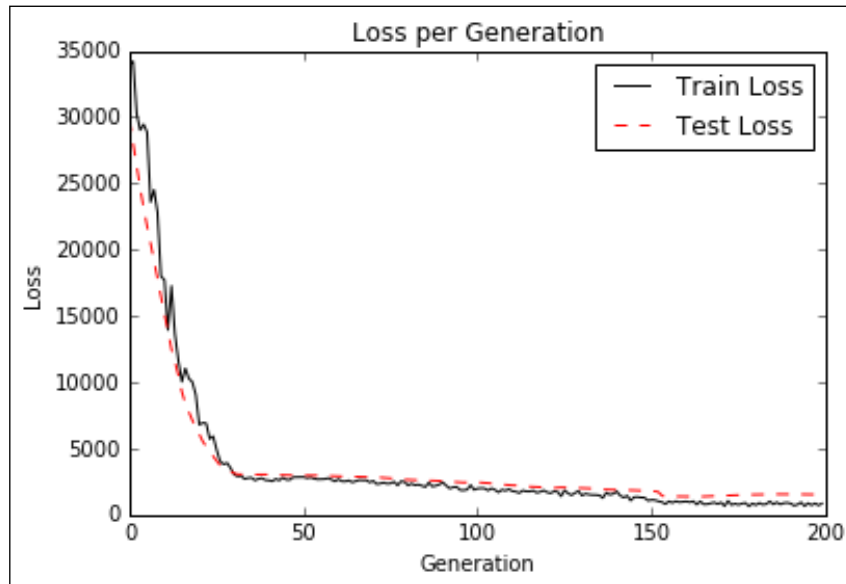
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
    loss_vec.append(temp_loss)

    test_temp_loss = sess.run(loss, feed_dict={x_data: x_vals_
test, y_target: np.transpose([y_vals_test])})
    test_loss.append(test_temp_loss)
    if (i+1)%25==0:
        print('Generation: ' + str(i+1) + '. Loss = ' + str(temp_
loss))
```

```
Generation: 25. Loss = 5922.52
Generation: 50. Loss = 2861.66
Generation: 75. Loss = 2342.01
Generation: 100. Loss = 1880.59
Generation: 125. Loss = 1394.39
Generation: 150. Loss = 1062.43
Generation: 175. Loss = 834.641
Generation: 200. Loss = 848.54
```

10. Here is a snippet of code that plots the train and test loss with matplotlib.

```
plt.plot(loss_vec, 'k-', label='Train Loss')
plt.plot(test_loss, 'r--', label='Test Loss')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.legend(loc="upper right")
plt.show()
```



Here we plot the train and test loss for our neural network that we trained to predict the birthweight in grams. Notice that only after about 30 generations we have arrived at a good model.

11. Now we want to compare our birthweight results to our prior logistic results. In the logistic linear regression (chapter 3, recipe 8), we achieved around 60% accuracy after thousands of iterations. To compare with what we have done here, we will output the train/test regression results and turn them into classification results by creating an indicator if they are above or below 2500g. Here is the code to arrive at this model's accuracy.

```
actuals = np.array([x[1] for x in birth_data])
test_actuals = actuals[test_indices]
train_actuals = actuals[train_indices]

test_preds = [x[0] for x in sess.run(final_output, feed_dict={x_
data: x_vals_test})]
train_preds = [x[0] for x in sess.run(final_output, feed_dict={x_
data: x_vals_train})]
```

```
test_preds = np.array([1.0 if x<2500.0 else 0.0 for x in test_
preds])
train_preds = np.array([1.0 if x<2500.0 else 0.0 for x in train_
preds])

# Print out accuracies
test_acc = np.mean([x==y for x,y in zip(test_preds, test_
actuals)])
train_acc = np.mean([x==y for x,y in zip(train_preds, train_
actuals)])
print('On predicting the category of low birthweight from
regression output (<2500g):')
print('Test Accuracy: {}'.format(test_acc))
print('Train Accuracy: {}'.format(train_acc))

Test Accuracy: 0.5526315789473685
Train Accuracy: 0.6688741721854304
```

How it works

In this recipe, we created a regression neural network with three fully connected hidden layers to predict the birthweight of the low-birthweight data set. When comparing this to a logistic output to predict above or below 2500 grams, we achieved similar results and achieved them in fewer generations. In the next recipe we will try to improve our logistic regression by making it a multiple layer logistic-type neural network.

Improving the Predictions of Linear Models

How to do it

1. We start with loading the libraries and initializing our computational graph.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import requests
```

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

2. Now we load, extract, and normalize our data just like in the prior recipe, except that we are using the low birthweight indicator variable as our target instead of the actual birthweight.

```
birthdata_url = 'https://www.umass.edu/statdata/statdata/data/
lowbwt.dat'
```

```

birth_file = requests.get(birthdata_url)
birth_data = birth_file.text.split('\r\n')[5:]
birth_header = [x for x in birth_data[0].split(' ') if len(x)>=1]
birth_data = [[float(x) for x in y.split(' ') if len(x)>=1] for y
in birth_data[1:] if len(y)>=1]

y_vals = np.array([x[1] for x in birth_data])

x_vals = np.array([x[2:9] for x in birth_data])

train_indices = np.random.choice(len(x_vals), round(len(x_
vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]

def normalize_cols(m):
    col_max = m.max(axis=0)
    col_min = m.min(axis=0)
    return (m-col_min) / (col_max - col_min)

x_vals_train = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test = np.nan_to_num(normalize_cols(x_vals_test))

```

3. Next we declare our batch size and our placeholders for the data.

```

batch_size = 90

x_data = tf.placeholder(shape=[None, 7], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

```

Just like before, we will declare functions that initialize a variable and a layer in our model. To create a better logistic function, we create a function that returns a logistic layer on an input layer. In other words, we will just use a fully connected layer and return a sigmoid element wise for each layer. It is important to remember that our loss function will have the final sigmoid included, so we want to specify on our last layer that we will not return the sigmoid of the output.

```

def init_variable(shape):
    return(tf.Variable(tf.random_normal(shape=shape)))

# Create a logistic layer definition

```

```
def logistic(input_layer, multiplication_weight, bias_weight,
activation = True):
    linear_layer = tf.add(tf.matmul(input_layer, multiplication_
weight), bias_weight)

    if activation:
        return(tf.nn.sigmoid(linear_layer))
    else:
        return(linear_layer)
```

4. Now we declare three layers (Two hidden layers and an output layer). We start by initializing a weight and bias matrix for each layer and defining the layer operations.

```
# First logistic layer (7 inputs to 14 hidden nodes)
A1 = init_variable(shape=[7,14])
b1 = init_variable(shape=[14])
logistic_layer1 = logistic(x_data, A1, b1)

# Second logistic layer (14 hidden inputs to 5 hidden nodes)
A2 = init_variable(shape=[14,5])
b2 = init_variable(shape=[5])
logistic_layer2 = logistic(logistic_layer1, A2, b2)

# Final output layer (5 hidden nodes to 1 output)
A3 = init_variable(shape=[5,1])
b3 = init_variable(shape=[1])
final_output = logistic(logistic_layer2, A3, b3, activation=False)
Next we declare our loss function (cross entropy), optimization
algorithm, and initialize the variables.
# Create loss function
loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
final_output, y_target))

# Declare optimizer
my_opt = tf.train.AdamOptimizer(learning_rate = 0.002)
train_step = my_opt.minimize(loss)

# Initialize variables
init = tf.initialize_all_variables()
sess.run(init)
```

5. In order to evaluate and compare our model to prior models, we want to create a prediction and accuracy operation on the graph. This will allow us to feed in the whole test set and determine the accuracy.

```
prediction = tf.round(tf.nn.sigmoid(final_output))
predictions_correct = tf.cast(tf.equal(prediction, y_target),
tf.float32)
accuracy = tf.reduce_mean(predictions_correct)
```

6. Now we start our training loop. We will train for 1500 generations and save the model loss and train/test accuracies for plotting later.

```
loss_vec = []
train_acc = []
test_acc = []
for i in range(1500):
    rand_index = np.random.choice(len(x_vals_train), size=batch_
size)
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target:
rand_y})

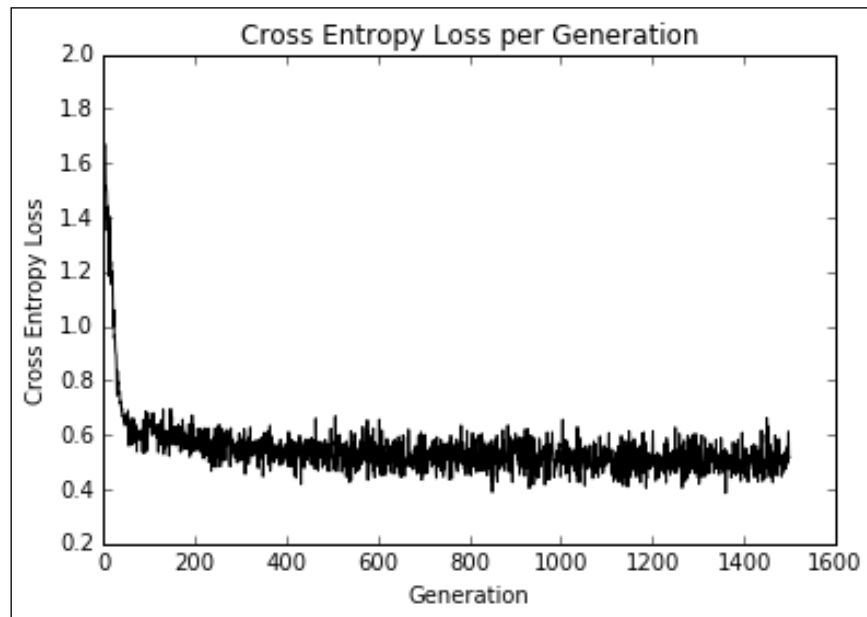
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_
target: rand_y})
    loss_vec.append(temp_loss)
    temp_acc_train = sess.run(accuracy, feed_dict={x_data: x_vals_
train, y_target: np.transpose([y_vals_train])})
    train_acc.append(temp_acc_train)
    temp_acc_test = sess.run(accuracy, feed_dict={x_data: x_vals_
test, y_target: np.transpose([y_vals_test])})
    test_acc.append(temp_acc_test)
    if (i+1)%150==0:
        print('Loss = ' + str(temp_loss))

Loss = 0.696393
Loss = 0.591708
Loss = 0.59214
Loss = 0.505553
Loss = 0.541974
Loss = 0.512707
Loss = 0.590149
Loss = 0.502641
Loss = 0.518047
Loss = 0.502616
```

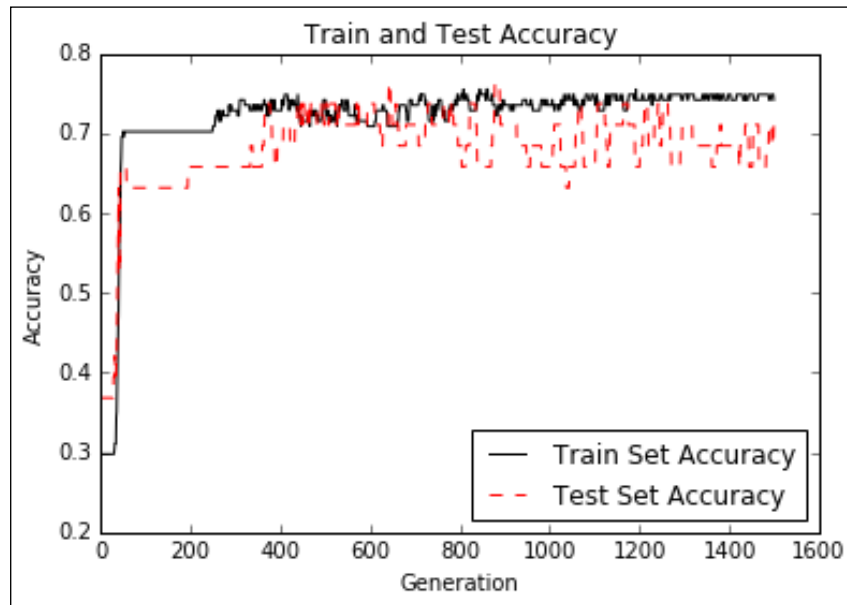

7. The following code blocks illustrate how to plot the cross entropy loss and the train and test set accuracies with matplotlib.

```
# Plot loss over time
plt.plot(loss_vec, 'k-')
plt.title('Cross Entropy Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Cross Entropy Loss')
plt.show()

# Plot train and test accuracy
plt.plot(train_acc, 'k-', label='Train Set Accuracy')
plt.plot(test_acc, 'r--', label='Test Set Accuracy')
plt.title('Train and Test Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```



Within approximately 50 generations, we have reached a good model. We continue to train and can see that very little is gained over the remaining generations.



Here we can see that we arrived at a good model very quickly. Also notice the slight overfitting occurring near the end.

How it works

When considering to use neural networks to model data, we must weigh the advantages and disadvantages. While our model has converged faster than prior models and may be a bit more accurate in some cases, this comes with a price. We are training many more model variables and have a greater chance of overfitting. To see the overfitting occurring, we look at the accuracy of the test and train sets and see the accuracy of the training set continue to increase slightly, while the accuracy on the test set stays the same or even decreases slightly.

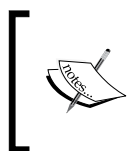
It is also important to note that our model variables are not as interpretable as a linear model. Neural network models are much more like a black box and harder to interpret.

Learning to play Tic-Tac-Toe

Getting ready

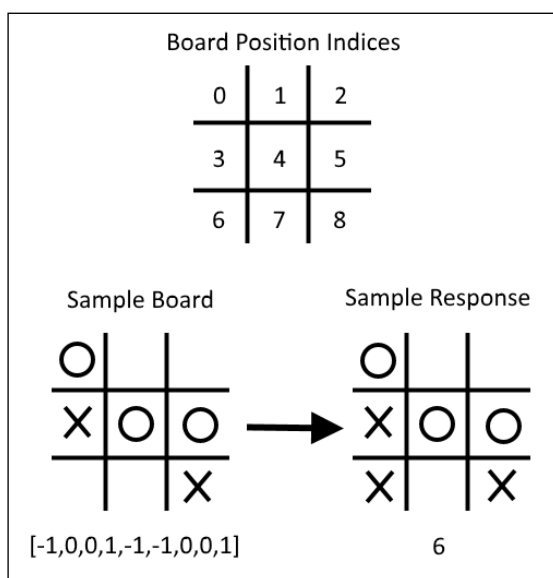
To end this chapter, we will attempt to use a neural network to learn the optimal moves of tic-tac-toe. We will approach this knowing that tic-tac-toe is a deterministic game and that the optimal moves are already known.

To train our model, we will have a list of board positions followed by the best optimal response for a number of different boards. We can reduce the amount of boards to train on by considering only board positions that are different with respect to symmetries. The non-identity transformations of a tic-tac-toe board are a rotation (either direction) by 90 degrees, 180 degrees, 270 degrees, a horizontal reflection, and a vertical reflection. Given this idea, we will use a short list of boards with the optimal move, apply two random transformations, and feed that into our neural network to learn.



Since tic-tac-toe is a deterministic game, it is worthwhile to note that whomever goes first should either win or draw. We will hope for a model that can respond to our moves optimally and result in a draw.

If we annotate X's by 1, O's by -1, and empty spaces by zero, then here is how we consider a board position and optimal move as a row of data.



Here we illustrate how to consider a board and optimal move as a row of data. Note that X = 1, O = -1, and empty spaces are 0, and we start indexing at 0.

In addition to the model loss, to check how our model is doing, we will do two things. The first check we will perform is to remove a position and optimal move row from our training set. This will allow us to see if the neural network model can generalize out to a move it hasn't seen before. The second method we will take to evaluate our model is to actually play a game against it at the end.

The list of possible boards and optimal moves can be found on the github directory for this recipe here: https://github.com/nfmcclure/tensorflow_cookbook/tree/master/06_Neural_Networks/08_Learning_Tic_Tac_Toe

How to do it

1. We start by loading the necessary libraries for this script.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import csv
import random
import numpy as np
import random
```

2. We declare our batch size for training our model.

```
batch_size = 50
```

3. To make visualizing the boards a bit easier, we will create a function that outputs the tic-tac-toe boards with X's and O's.

```
def print_board(board):
    symbols = ['O', ' ', 'X']
    board_plus1 = [int(x) + 1 for x in board]
    print(' ' + symbols[board_plus1[0]] + ' | ' + symbols[board_plus1[1]] + ' | ' + symbols[board_plus1[2]])
    print('_____')
    print(' ' + symbols[board_plus1[3]] + ' | ' + symbols[board_plus1[4]] + ' | ' + symbols[board_plus1[5]])
    print('_____')
    print(' ' + symbols[board_plus1[6]] + ' | ' + symbols[board_plus1[7]] + ' | ' + symbols[board_plus1[8]])
```

4. Now we have to create a function that will return a new board and optimal response position under a transformation.

```
def get_symmetry(board, response, transformation):
    '''
    :param board: list of integers 9 long:
    opposing mark = -1
    friendly mark = 1
```

```

        empty space = 0
    :param transformation: one of five transformations on a board:
        'rotate180', 'rotate90', 'rotate270', 'flip_v', 'flip_h'
    :return: tuple: (new_board, new_response)
    """
    if transformation == 'rotate180':
        new_response = 8 - response
        return(board[::-1], new_response)
    elif transformation == 'rotate90':
        new_response = [6, 3, 0, 7, 4, 1, 8, 5, 2].index(response)
        tuple_board = list(zip(*[board[6:9], board[3:6],
board[0:3]]))
        return([value for item in tuple_board for value in item],
new_response)
    elif transformation == 'rotate270':
        new_response = [2, 5, 8, 1, 4, 7, 0, 3, 6].index(response)
        tuple_board = list(zip(*[board[0:3], board[3:6],
board[6:9]]))[::-1]
        return([value for item in tuple_board for value in item],
new_response)
    elif transformation == 'flip_v':
        new_response = [6, 7, 8, 3, 4, 5, 0, 1, 2].index(response)
        return(board[6:9] + board[3:6] + board[0:3], new_
response)
    elif transformation == 'flip_h': # flip_h = rotate180, then
flip_v
        new_response = [2, 1, 0, 5, 4, 3, 8, 7, 6].index(response)
        new_board = board[::-1]
        return(new_board[6:9] + new_board[3:6] + new_board[0:3],
new_response)
    else:
        raise ValueError('Method not implmented.')

```

5. The list of boards and their optimal response is in a csv file in the directory. We will create a function that will load the file of boards and responses and store it as a list of tuples.

```

def get_moves_from_csv(csv_file):
    """
    :param csv_file: csv file location containing the boards w/
responses
    :return: moves: list of moves with index of best response
    """
    moves = []
    with open(csv_file, 'rt') as csvfile:
        reader = csv.reader(csvfile, delimiter=',')

```

```

        for row in reader:
            moves.append(([int(x) for x in row[0:9]],int(row[9])))
    return(moves)

```

6. Now we tie everything together to create a function that will return a randomly transformed board and response.

```

def get_rand_move(moves, rand_transforms=2):
    """
    :param moves: list of the boards w/responses
    :param rand_transforms: # random transforms performed on each
    :return: (board, response), board is a list of 9 integers,
    response is 1 int
    """
    (board, response) = random.choice(moves)
    possible_transforms = ['rotate90', 'rotate180', 'rotate270',
    'flip_v', 'flip_h']
    for i in range(rand_transforms):
        random_transform = random.choice(possible_transforms)
        (board, response) = get_symmetry(board, response, random_
transform)
    return(board, response)

```

7. Now we initialize our graph session, load our data and create a training set.

```

sess = tf.Session()
moves = get_moves_from_csv('base_tic_tac_toe_moves.csv')

# Create a train set:
train_length = 500
train_set = []
for t in range(train_length):
    train_set.append(get_rand_move(moves))

```

8. Remember that we want to remove one board and optimal response from our training set to see if the model can generalize out to make the best move. The best move for the board below will be to play at index number six.

```

test_board = [-1, 0, 0, 1, -1, -1, 0, 0, 1]
train_set = [x for x in train_set if x[0] != test_board]

```

9. We can now create functions to create our model variables and our model operations. Note that we do not include the softmax() activation function in the model because it is included in the loss function.

```

def init_weights(shape):
    return(tf.Variable(tf.random_normal(shape)))

def model(X, A1, A2, bias1, bias2):

```

```
layer1 = tf.nn.sigmoid(tf.add(tf.matmul(X, A1), bias1))
layer2 = tf.add(tf.matmul(layer1, A2), bias2)
return(layer2)
```

10. Now we declare our placeholders, variables, and model.

```
X = tf.placeholder(dtype=tf.float32, shape=[None, 9])
Y = tf.placeholder(dtype=tf.int32, shape=[None])
```

```
A1 = init_weights([9, 81])
bias1 = init_weights([81])
A2 = init_weights([81, 9])
bias2 = init_weights([9])
```

```
model_output = model(X, A1, A2, bias1, bias2)
```

11. Next, we declare our loss function, which will be the average softmax of the final output logits. Then we declare our training step and optimizer. We also need to create a prediction operation if we want to be able to play against our model in the future.

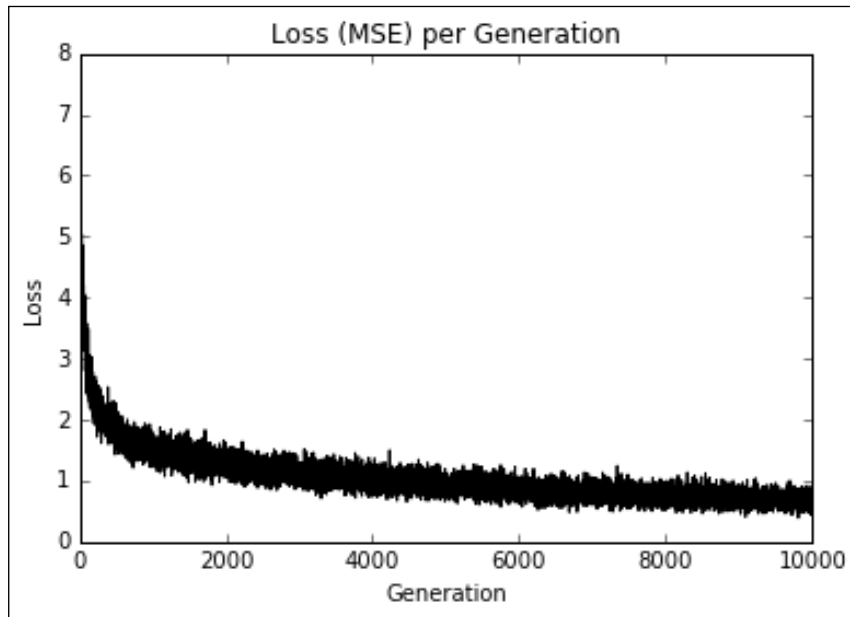
```
loss = tf.reduce_mean( tf.nn.sparse_softmax_cross_entropy_with_
logits(model_output, Y))
train_step = tf.train.GradientDescentOptimizer(0.025).
minimize(loss)
prediction = tf.argmax(model_output, 1)
We can now initialize our variables and loop through the training
of our neural network.
init = tf.initialize_all_variables()
sess.run(init)

loss_vec = []
for i in range(10000):
    rand_indices = np.random.choice(range(len(train_set)), batch_
size, replace=False)
    batch_data = [train_set[i] for i in rand_indices]
    x_input = [x[0] for x in batch_data]
    y_target = np.array([y[1] for y in batch_data])
    sess.run(train_step, feed_dict={X: x_input, Y: y_target})

    temp_loss = sess.run(loss, feed_dict={X: x_input, Y: y_
target})
    loss_vec.append(temp_loss)
    if i%500==0:
        print('iteration ' + str(i) + ' Loss: ' + str(temp_loss))
```

12. Here is code to plot the loss over the model training.

```
plt.plot(loss_vec, 'k-', label='Loss')
plt.title('Loss (MSE) per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```



Here we plot the loss over the training steps.

1. To test the model, we see how it performs on the test board that we removed from the training set. We are hoping that the model can generalize and predict the optimal index for moving, which will be the index six. Most of the time the model will succeed here.

```
test_boards = [test_board]
feed_dict = {X: test_boards}
logits = sess.run(model_output, feed_dict=feed_dict)
predictions = sess.run(prediction, feed_dict=feed_dict)
print(predictions)
```

[6]

2. In order to play against our trained model, we have to create a function that will check for a win. This way we know when to stop asking for more moves.

```
def check(board):
    wins = [[0,1,2], [3,4,5], [6,7,8], [0,3,6], [1,4,7], [2,5,8],
            [0,4,8], [2,4,6]]
    for i in range(len(wins)):
        if board[wins[i][0]]==board[wins[i][1]]==board[wins[i]
[2]]==1.:
            return(1)
        elif board[wins[i][0]]==board[wins[i][1]]==board[wins[i]
[2]]==-1.:
            return(1)
    return(0)
```

3. Now we can loop through and play a game with our model. We start with a blank board (all zeros). Then we start by asking the user to input an index (0-8) of where to play and then feed that into the model for a prediction. For the model's move, we take the largest available prediction that is also an open space. A sample game is shown at the end. From this game, we can see that our model is not perfect.

```
game_tracker = [0., 0., 0., 0., 0., 0., 0., 0., 0.]
win_logical = False
num_moves = 0
while not win_logical:
    player_index = input('Input index of your move (0-8): ')
    num_moves += 1
    # Add player move to game
    game_tracker[int(player_index)] = 1.

    # Get model's move by first getting all the logits for each
    index
    [potential_moves] = sess.run(model_output, feed_dict={X:
[game_tracker]})
    # Now find allowed moves (where game tracker values = 0.0)
    allowed_moves = [ix for ix,x in enumerate(game_tracker) if
x==0.0]
    # Find best move by taking argmax of logits if they are in
    allowed moves
    model_move = np.argmax([x if ix in allowed_moves else -999.0
for ix,x in enumerate(potential_moves)])

    # Add model move to game
    game_tracker[int(model_move)] = -1.
    print('Model has moved')
    print_board(game_tracker)
    # Now check for win or too many moves
```

```

if check(game_tracker)==1 or num_moves>=5:
    print('Game Over!')
    win_logical = True

```

```

Input index of your move (0-8): 4
Model has moved

```

```

O |   |
-----
  | X |
-----
  |   |

```

```

Input index of your move (0-8): 6
Model has moved

```

```

O |   |
-----
  | X |
-----
X |   | O

```

```

Input index of your move (0-8): 2
Model has moved

```

```

O |   | X
-----
O | X |
-----
X |   | O
Game Over!

```

How it works

We trained a neural network to play tic-tac-toe by feeding in board positions, a nine-dimensional vector, and predicted the optimal response. We only had to feed in a few possible tic-tac-toe boards and apply random transformations to each to increase the training set size.

To test our algorithm, we removed all instances of a specific board and saw if our model could generalize to predict the optimal response. At the end, we also played a sample game against our model. While it is still not perfect, we could try different architectures and training procedures to improve it.

