# Introduction to Deep Learning

Robert A. Brown, PhD

robert.brown@mcgill.ca

Draft 1
January 5, 2016

# Introduction to Deep Learning

Deep learning is a branch of machine learning that emphasizes the use of multiple layers of successive abstraction. Deep learning is often associated with *artificial neural networks* (ANNs) since this type of model is particularly suitable for, and widely used in deep learning.

These notes attempt to provide the background information necessary to develop an intuitive understanding of basic deep learning concepts, and introduces simple deep learning algorithms using practical examples in TensorFlow, an open source Python and C computational library produced by Google for deep learning. While Google provides excellent TensorFlow tutorials, an intuitive understanding may help you more successfully apply these tools to real-world problems.

## Graph Theory

Graph theory models relationships using "graphs," which are composed of two elements: nodes and edges. Nodes are also called vertices, points or, in deep learning, neurons. Edges are also called arcs, lines or connections. A graph is composed of a set of nodes connected by a set of edges. Graphs may be *directed*, where the connections are unidirectional, or *undirected*, where the connections are bidirectional.

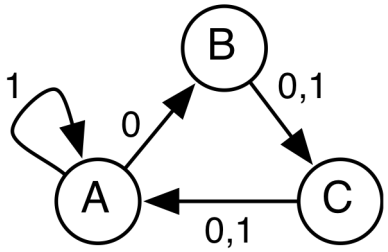


*Figure 1: A simple finite state machine expressed as a graph. The nodes encode states A, B and C while the edges show allowed transitions among states. Each edge is labelled with the binary input that will cause that transition.*

Graphs are useful for describing many different systems, including finite state machines, relationships between individuals (*e.g.* social network graphs), computer networks, statistical models, and even computations. TensorFlow itself models computations as graphs.

## Machine Learning

Machine learning involves specifying a model, then fitting that model to training data. Input data generally consists of one or more measurements (or *features*) made on several samples (*e.g.* height and weight measured for a group of people). Training (or learning) consists of fitting the model: deducing values for the model's parameters that allow the best prediction of the data. This process is called optimization, and for some simple models (*e.g.* linear regression using least squares) analytic solutions exist for the fitting step. For more complex models an iterative algorithm must be used (*e.g.* gradient descent).

In optimization, an equation is specified that measures how well the model fits. By convention this equation has smaller values for better fitting models, and is called a *cost function*. Cost functions often measure the mismatch or distance between predictions of the model and the desired output. A very common example is *mean squared error*. Once a cost function is

specified, optimization involves choosing the model parameter values that minimize the cost function.

Machine learning algorithms can be classified into two types: *supervised* and *unsupervised*. Supervised machine learning algorithms require training data where each sample is labelled with the desired result (or "truth"). For example, if we wanted to train a supervised algorithm to differentiate pictures of cats and dogs we would require a dataset of pictures of cats or dogs with each picture labelled with the animal it contained. Examples of supervised machine learning algorithms are *Bayesian classifiers* and *support vector machines*. In contrast, an unsupervised machine learning algorithm tries to detect patterns in data without knowing the "truth." In this case we could use only pictures, without the need for a person to go through the training set and manually identify the animal. A common unsupervised algorithm is *k-means*. A third type of algorithm compares new data directly to an existing database; these are similar to supervised learning in that they require labelled examples, but it might be a stretch to call these learning algorithms because no optimization takes place. An example is *k-nearest neighbour*.

## Machine Learning Algorithms as Graphs

Many common machine learning techniques can be represented as graphs, including deep learning. In this section we will look at a few common examples, which will provide context for subsequent sections.

### Decision Trees

Decision tree learning tries to predict a target variable using a hierarchy of criteria, based on a set of input variables. Consider an example (modified from Wikipedia) of a decision tree to estimate the chances of survival for passengers on the titanic. Each criterion is in the form of a true or false question. This model can be represented as a graph (Figure 2) and the decision tree learning algorithm is used to identify important criteria and construct this graph. Once this is done, the chances of survival for an individual passenger can be estimated by traversing the graph based on the passenger's characteristics.

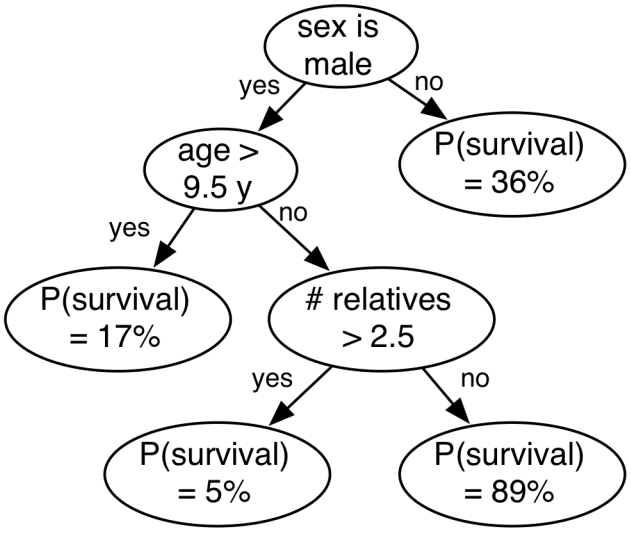


*Figure 2: A graph showing a decision tree model predicting probability of survival on the Titanic.*

### Bayesian Models

Naïve Bayesian classifiers try to estimate the probability that an example, represented by a set of features, belongs to a particular class. This is done by applying Bayes' theorem under the

assumption that each feature's contribution is independent of that of all the other features (thus the naïve). Naïve Bayesian classifiers are usually represented by graphs where a node representing a particular class (C) is connected to nodes representing the probability of finding particular features in samples from class C (Figure 3). If relationships among the features are allowed, the model is a more general Bayesian network.

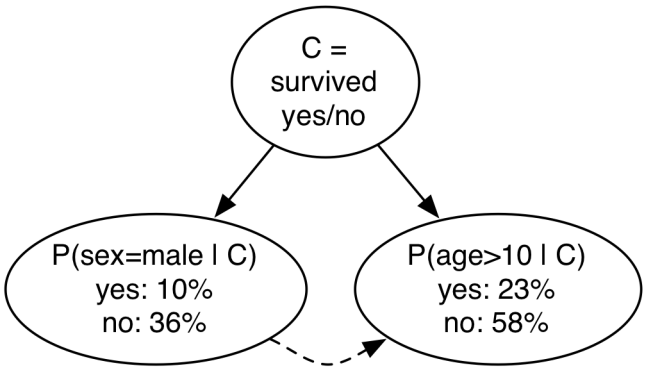


*Figure 3: A graph representing a Bayesian model for estimating survival on the Titanic based on passenger age and sex (values are made up). If the dotted arc is eliminated, the model is a naïve Bayesian classifier (features are independent of each other). If the arc is included, the model is a more general Bayesian network.*

## Regression

Regression is a class of statistical models that try to estimate the value of a dependent variable given values of one or more independent variables (features). The most familiar example is linear regression, where the input variable values are multiplied by adjustable parameters, then added together to produce the prediction. A closely related technique is logistic regression, where the dependent variable is categorical. Figure 4 (left) shows the simplest linear regression model, $y = mx_1 + b$ where *m* and *b* are parameters of the model commonly known as the *slope* and i*ntercept*. Note that the graph in Figure 4 also describes a computation consisting of multiplication and addition operations.

The graph on the right in Figure 4 predicts two variables, $y_1$ and $y_2$. If we added an operation to identify whichever *y* had the maximum value, we would have a logistic regression model.

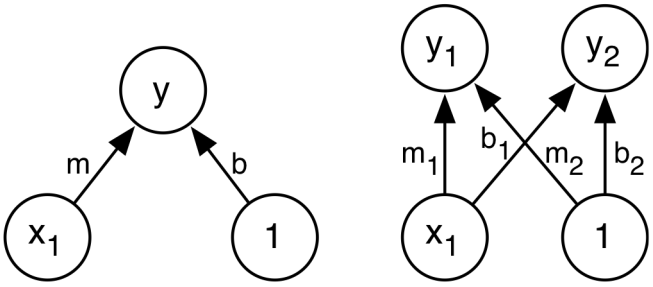


*Figure 4: Graphs representing regression with a single independent variable ($x_1$). In linear regression (left) the slope (m) and intercept (b) are parameters of the model. This graph also represents the computation $y = mx_1 + b$. This graph can be expanded (right) to predict two values, $y_1$ and $y_2$. Adding an operation to select the y with the maximum value would convert the graph into logistic regression.*

## Intermezzo: TensorFlow

Many software libraries exist that use the computational graph formalism to enable features like automatic optimization, auto vectorization and efficient compilation for a variety of architectures, including CPUs and GPUs. Many of these are specifically designed for machine learning in general and deep learning in particular. The similarities between graphical models of algorithms and graphical models of computation introduced in the previous section

may have given you a hint as to why this might be. One particularly important feature of this type of computational model is that derivatives can be calculated automatically. The derivative of the cost function is particularly critical for efficient optimization.

TensorFlow is an open source library for graph-based computation, written by Google. Although it is a general computational library, it is clearly aimed at deep learning.

## Concepts and General Model

TensorFlow is based on a graph model where data, in the form of tensors (multidimensional arrays) "flows" through the graph, being transformed by operations at nodes. Nodes can be assigned to different computational devices (CPUs or GPUs) and execute when all of their input tensors are available.

## Boilerplate

Our examples will generally use some common boilerplate code to set things up. First, import TensorFlow and also the Python math module:

```
import tensorflow as tf
import math
```

If the example requires the MNIST dataset, also include the following (you will need the input_data.py file from the TensorFlow examples).

```
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

Finally, set up some TensorFlow variables:

```
sess = tf.InteractiveSession()
x = tf.placeholder('float',shape=[None,784],name='input')          #
Input tensor
y_ = tf.placeholder('float', shape=[None,10],name='correctLabels')
     # Correct labels
```

The first line creates a TensorFlow interactive session. You can think of this as creating a blank graph for us to start adding operations to. The second line creates a TensorFlow "placeholder" variable called *x*, which we will fill with our input data. Finally, the variable *y_* is another TensorFlow placeholder that we will use to hold the correct labels. Note that *x* and *y_* are created to hold floating point data, and that we (partially) specify their shapes in advance. Each MNIST image is 28x28, but is stored as a 1-dimensional array with 784 elements. Our label will be a 10 item vector in "one-hot" format: zeros for all items except the one corresponding to the correct digit, from 0 to 9. Both variables have *None* as the shape of the first dimension; we will train with multiple MNIST examples at a time in a "minibatch." We do not want to specify the size of the minibatch in advance, so we use *None*, indicating to TensorFlow that it will have to figure it out at runtime.

Placeholder variables are just that: space reserved in the computational graph for data that will be supplied at runtime.

## Artificial Neural Networks

An *artificial neural network* (ANN), loosely inspired by natural neural networks in animal brains, consists of a set of "neurons" with *connections* between them, analogous to synapses. Each connection has a parameter called the *weight* and multiplies the signal from its input "neuron" by its weight. Each "neuron" sums up the signals from its input connections, modifies this value according to an *activation function*, then propagates this value through its outgoing connections.

The simple regression models in Figure 4 are also examples of very simple ANNs. Input neurons (*x* and a special neuron that always signals with a value of 1) produce signals related to the input data. These signals propagate through connections, which multiply the signals by the connection weights *m* and *b*. Finally, the output neuron(s) *y* add up the incoming signals. This particular type of ANN has the (very common) simplifying property that neurons can be divided up into layers (input and output in this case) and there are no connections within layers, or connections that produce feedback (loops in the graph).

### Softmax Network

In the linear regression model in Figure 4 (left) the output neurons have an identity activation function: they simply take the value of whatever the sum of incoming signals is. If we instead use the model in Figure 4 (right), but make the activation function the *softmax* function, we have a softmax regression model. This model is related to logistic regression (and is often called logistic regression in deep learning literature) but instead of identifying a single class the output is a set of class probabilities. If the class with the maximum value is then identified, we have true logistic regression.

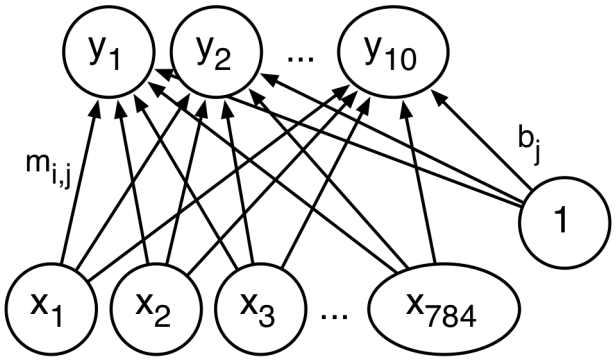


*Figure 5: A softmax regression model (or one-layer ANN) where the pixel values from a MNIST example are represented by input neurons (x), signals from these are propagated over connections with weights $m_{i,j}$ and then summed, along with biases $b_j$ by the output neurons y. The y neurons use the softmax activation function.*

We can easily build a softmax model in TensorFlow as our first ANN. Starting with the boilerplate code from above, add the lines:

```
W = tf.Variable(tf.zeros([784,10]),name='W_logistic')
b = tf.Variable(tf.zeros([10]),name='b_logistic')
y = tf.nn.softmax(tf.matmul(x,W) + b)
```

That's it. *W* (corresponding to *m* in Figure 4) holds the weights connecting each of the 784 MNIST input features to the 10 output neurons. *b* holds the intercept parameters, which can be thought of as the weights of the connections from special always on neurons, one for each output neuron (called biases in ANN literature). *W* and *b* are TensorFlow variables, which are similar to placeholders, but are not intended to be assigned external values (except once, at initialization). *W* and *b* are initialized to tensors with all elements set to zero.

This model is represented by the equation:

$$\boldsymbol{y} = softmax(\boldsymbol{xW}) + \boldsymbol{b}$$

and the graph shown in Figure 5. Note that the last line of the code snippet is a direct translation of this equation.

The network can be shown a MNIST example using the code:

```
tf.initialize_all_variables().run()
batch = mnist.train.next_batch(1)
y.eval(feed_dict={x:batch[0]})
```

The first line tells TensorFlow to create the computational graph and initialize all the variables. The second line retrieves a (random) sample from the MNIST dataset (a minibatch size of 1). The sample is a tuple of the form (*x*,*y_*) The third line replaces the value of the placeholder variable *x* with the contents of batch[0], then does all the computations necessary to evaluate *y*, the model output. If you run this code you should get a numpy array with ten values of 0.1. These are the output class probabilities. They are all equal because all the connections are equally zero. Now we need to train our model.

We train an ANN by showing it examples, *x*, (usually several examples at once, assembled into a minibatch), computing the value of a cost function that measures how far away our prediction, *y* is from the truth*, y_*, then minimizing the cost. This is usually done many times, with different minibatches.

First we need a cost function. We can determine whether our prediction (defined as the class with the highest probability) was correct and calculate our accuracy over the minibatch using:

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction,'float'))
```

However, accuracy does not behave well as a cost function. We'd prefer a smoother function, which is easier to optimize. Instead, we can use the cross entropy, calculated as:

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y))
```

We can then set up one of the TensorFlow optimizers to minimize our cost function, and iterate through several minibatches. We'll also print our accuracy periodically so we can see how things are going:

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
for i in range(1000):         # Do some training
     batch = mnist.train.next_batch(100)

     if i%100 == 0:
       train_accuracy = accuracy.eval(feed_dict={x:batch[0],y_:batch[1]})
       print 'Accuracy at step %d: %g ' % (i,train_accuracy)

     train_step.run(feed_dict={x:batch[0],y_:batch[1]})

print 'Test accuracy: %g' % accuracy.eval(feed_dict={x:mnist.test.images,
y_:mnist.test.labels})
```

This code creates *train_step* which is the Adam optimizer run on our *cross_entropy* cost function. It then loops through 1000 minibatches of 100 examples each, computing and printing the accuracy if we are on a minibatch divisible by 100, then executing *train_step*. Finally, the accuracy is computed for the MNIST test set, a set of examples reserved for evaluating model performance. It is important that the overall performance of a model be tested on data that was not used during training.

The complete code for this example is included on the next page. If you have TensorFlow installed on your computer, try running it. You should end up with a softmax regression model that can identify MNIST digits with about 85% accuracy. On my 2.5 GHz dual i7 2015 Macbook Pro this took a second or two to execute.

```
import tensorflow as tf
import math
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

# Common setup
sess = tf.InteractiveSession()
x = tf.placeholder('float',shape=[None,784],name='input')         # Input
tensor
y_ = tf.placeholder('float', shape=[None,10],name='correctLabels')
    # Correct labels


# MODEL DEFINITION

# Softmax regression
W = tf.Variable(tf.zeros([784,10]),name='W_logistic')
b = tf.Variable(tf.zeros([10]),name='b_logistic')
y = tf.nn.softmax(tf.matmul(x,W) + b)

# END MODEL DEFINITION

cross_entropy = -tf.reduce_sum(y_*tf.log(y))
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction,'float'))

tf.initialize_all_variables().run()       # Take initial values and actually
put them in variables

for i in range(1000):                                  # Do some training
    batch = mnist.train.next_batch(100)
        if i%100 == 0:
            train_accuracy =
accuracy.eval(feed_dict={x:batch[0],y_:batch[1]})
                print 'Accuracy at step %d: train: %g' % (i,train_accuracy)

    train_step.run(feed_dict={x:batch[0],y_:batch[1]})

print 'Test accuracy: %g' % accuracy.eval(feed_dict={x:mnist.test.images,
y_:mnist.test.labels})
```

## Multilayer Perceptrons

Try replacing the code in the MODEL DEFINITION section with the following:

```
# Two-layer softmax regression
Wlin = tf.Variable(tf.truncated_normal([784,784],stddev= 1.0 /
math.sqrt(784)),name='W_linear')
blin = tf.Variable(tf.zeros([784]),name='b_linear')
y_intermediate = tf.matmul(x,Wlin) + blin

W = tf.Variable(tf.zeros([784,10]),name='W_logistic')
b = tf.Variable(tf.zeros([10]),name='b_logistic')
y = tf.nn.softmax(tf.matmul(y_intermediate,W) + b)
```

This code stacks our previous model on top of an additional layer of neurons (and their connections). Now we have our input *x* propagated through *Wlin* to an intermediate layer of neurons called *y_intermediate*. These neurons have a linear activation function (they just sum up their inputs) and these values are passed to our softmax regressor from before. There is one other difference in the first layer: *Wlin* is initialized not to zeros but to "tf.truncated_normal." This function draws random samples from a normal distribution with the tails cut off. The random initialization is necessary to "break the symmetry" of the network. If all the connections have identical values, the optimizer to choose a direction move in.

If you run this model it will take longer to train (five to ten seconds) but it also achieves greater accuracy (around 90%). This is an example of the power of *deep learning*. Our regular softmax regressor couldn't encode relationships between inputs, analogous to the naïve Bayesian classifier. This two-layer network *can* encode those relationships, in the *y_intermediate* layer, which are then used by the softmax layer to produce a better classification. The intermediate layer allows the model to learn a more abstracted representation, where input features are combined to form a set of learned features. If you are a statistician, you might think of this as analogous to adding interaction terms to your model.

Our two-layer network has an additional problem: it is still linear. In order to learn the complicated nonlinear relationships in most real-world data we need to introduce nonlinearity into our network. We do this using a nonlinear activation function. Early ANNs used activation functions known as sigmoid functions, such as the hyperbolic tangent. These activation functions caused a variety of problems during training, so most modern deep networks use the rectified linear (ReLU) function. This function takes a value of zero when the input is negative, and is linear for all positive values. This activation function is also more biologically plausible: it effectively models a neuron that doesn't fire at all until sufficiently stimulated, then increases its activity in proportion to its input. We can modify our two-layer ANN to include ReLU activation functions for the middle layer neurons (changes in bold):

```
# Two-layer perceptron
Wlin = tf.Variable(tf.truncated_normal([784,784],stddev= 1.0 /
math.sqrt(784)),name='W_linear')
blin = tf.Variable(tf.zeros([784]),name='b_linear')
```

```
y_intermediate = tf.nn.relu(tf.matmul(x,Wlin) + blin)

W = tf.Variable(tf.zeros([784,10]),name='W_logistic')
b = tf.Variable(tf.zeros([10]),name='b_logistic')
y = tf.nn.softmax(tf.matmul(y_intermediate,W) + b)
```

The resulting model is a full fledged multi-layer perceptron (MLP) with input, hidden and output layers, and was state of the art in the 1980s. It performs only slightly better (about 91%) but remember, this is a reduction of 10% in our error rate! Also, the difference becomes more pronounced if we train for more than 1000 minibatches (try 5000, for example).

Note that the multi-layer perceptron was a type of ANN generally including nonlinear units and trained via optimization. A *perceptron* was a hardware device constructed in the 1950s that has more in common with our softmax model, although it did use nonlinear units.

## Deep Learning

In the 1980s and 90s interest in two-layer MLPs waned because they did not outperform *support vector machines*, which could be trained more quickly and were considered simpler and easier to understand. Although it was thought that deeper models (ANNs with more than two layers) would be more powerful, they were very difficult and slow to train. In the 2000s ANNs experienced a renaissance when methods for training deep networks were discovered.

The era of modern deep learning began when Geoff Hinton demonstrated that deep networks could be efficiently trained using *unsupervised pre-training*. One of the simplest approaches to unsupervised pre-training uses the *autoencoder*. A simple autoencoder consists of two layers, similar to our ANN in the previous section. However, these layers are symmetric: the first layer learns to *encode* the input in an abstract form, and the second layer learns to *decode* this and reproduce the input as closely as possible. If the cost function includes incentives for the network to produce an efficient encoding, *e.g.* using fewer neurons than there are features in the input, then the encoder tends to learn to identify more abstract features, such as edges in images. A second two-layer autoencoder is then inserted between the first encoder and decoder, and trained. When the required number of encoders have been trained, the decoders can be removed, a final layer, such as our softmax layer, can be appended, and the deep network “fine-tuned” using regular optimization. Using this approach, no more than two layers (the encoder and the decoder) are ever trained at once, and the final whole-network fine tuning is much easier than training a deep network from scratch.

It turns out that using ReLU activation functions in place of sigmoids solves many of the problems with training deep networks. However, unsupervised pre-training may still be useful if, as is the usual case, most of the available data is unlabeled. Unsupervised training is also of interest as it spontaneously discovers successively more abstract features in the data. For images, the first layer tends to identify edges, the second corners, the third simple shapes, and so on, until higher level layers may spontaneously develop neurons that respond specifically to cats, or Halle Berry. Google Translate uses a deep learning architecture in which deep encoders translate a source language to a common internal representation, then decoders translate back to the desired target language.

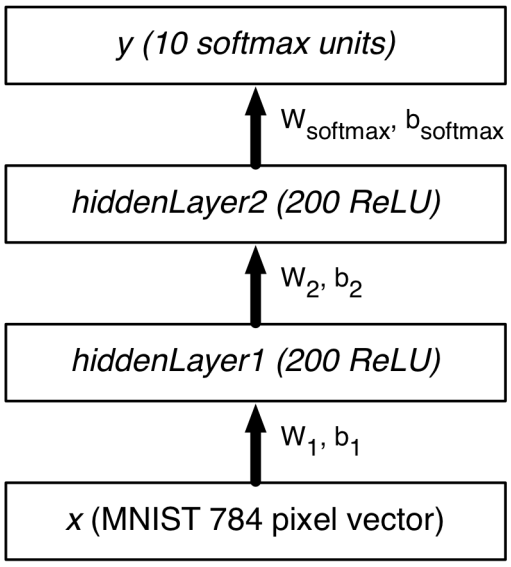


*Figure 6: A simple deep network with two hidden ReLU layers and a softmax output layer*

You can certainly experiment with autoencoders in TensorFlow, but for now combine everything we have learned to this point in this basic three-layer deep network using ReLU (illustrated in Figure 6):

```
# Three-layer Deep Network
hiddenUnitN1 = 200; hiddenUnitN2 = 200
W1 = tf.Variable(tf.truncated_normal([784,hiddenUnitN1],stddev= 1.0 /
math.sqrt(hiddenUnitN1)),name='W1')
b1 = tf.Variable(tf.zeros([hiddenUnitN1]),name='b1')
hiddenLayer1 = tf.nn.relu(tf.matmul(x,W1) + b1)

W2 =
tf.Variable(tf.truncated_normal([hiddenUnitN1,hiddenUnitN2],stddev=
1.0 / math.sqrt(hiddenUnitN2)),name='W2')
b2 = tf.Variable(tf.zeros([hiddenUnitN2]),name='b2')
hiddenLayer2 = tf.nn.relu(tf.matmul(hiddenLayer1,W2) + b2)

WSoftmax = tf.Variable(tf.zeros([hiddenUnitN2,10]),name='W_softmax')
bSoftmax = tf.Variable(tf.zeros([10]),name='b_softmax')
y = tf.nn.softmax(tf.matmul(hiddenLayer2,WSoftmax) + bSoftmax)
```

This network achieves similar performance to our two-layer perceptron (91%), but uses about half as many neurons and is faster to train. The advantages for deeper networks also tend to be more pronounced for problems that are more difficult than MNIST.

## Convolutional Networks

I don’t think we’ll get to here in the first session, so this part isn’t finished yet.  ;)