Lecture 12
Introduction to Neural Networks

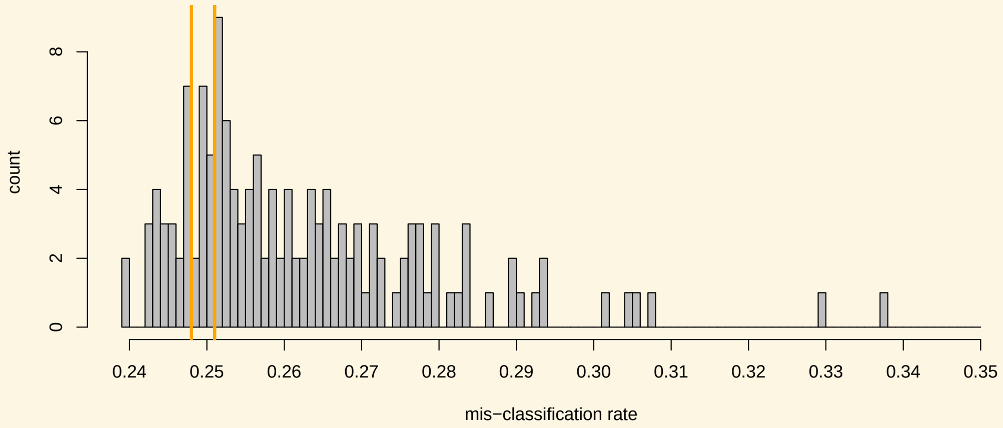
29 February 2016

Taylor B. Arnold
Yale Statistics
STAT 365/665

Yale

Notes:

- Problem set 4 is due this Friday (SVM implementation)
- Problem set 5 will be posted prior to class tomorrow (neural net ‘implementation’)


count
0
2
4
6
8
0.24
0.25
0.26
0.27
0.28
0.29
0.30
0.31
0.32
0.33
0.34
0.35
mis−classification rate

Today

- Introducing neural network architecture
- high level description of how to learn neural networks and specific challenges in doing so
- simulation example of SGD

There are a large set of introductions to neural networks online. Popular ones that I like are:

- Andrej Karpathy's Hacker's guide to Neural Networks: `http://karpathy.github.io/neuralnets/`
- Andrej Karpathy's lecture notes: `http://cs231n.github.io/`
- Geoffrey E. Hinton, Yann LeCun, and Yoshua Bengio (video; NIPS 2015): `http://research.microsoft.com/apps/video/default.aspx?id=259574`
- Michael Nielsen's Neural Networks and Deep Learning: `http://neuralnetworksanddeeplearning.com/`

I think these are all worthwhile, and approach the subject from slightly different angles and with different learning outcomes. I am going to (very) closely follow Michael Nielsen's notes for the next two lectures, as I think they work the best in lecture format and for the purposes of this course. We will then switch gears and start following Karpathy's lecture notes in the following week.

A simple decision

Say you want to decide whether you are going to attend a cheese festival this upcoming weekend. There are three variables that go into your decision:

1. Is the weather good?
2. Does your friend want to go with you?
3. Is it near public transportation?

We'll assume that answers to these questions are the only factors that go into your decision.

A simple decision, cont.

I will write the answers to these question as binary variables x_i, with zero being the answer 'no' and one being the answer 'yes':

1. Is the weather good? x_1
2. Does your friend want to go with you? x_2
3. Is it near public transportation? x_3

Now, what is an easy way to describe the decision statement resulting from these inputs.

A simple decision, cont.

We could determine weights w_i indicating how important each feature is to whether you would like to attend. We can then see if:

$$x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 \geq \text{threshold}$$

For some pre-determined threshold. If this statement is true, we would attend the festival, and otherwise we would not.

A simple decision, cont.

For example, if we really hated bad weather but care less about going with our friend and public transit, we could pick the weights 6, 2 and 2.

A simple decision, cont.

For example, if we really hated bad weather but care less about going with our friend and public transit, we could pick the weights 6, 2 and 2.

With a threshold of 5, this causes us to go if and only if the weather is good.

A simple decision, cont.

For example, if we really hated bad weather but care less about going with our friend and public transit, we could pick the weights 6, 2 and 2.

With a threshold of 5, this causes us to go if and only if the weather is good.

What happens if the threshold is decreased to 3? What about if it is decreased to 1?

A simple decision, cont.

If we define a new binary variable y that represents whether we go to the festival, we can write this variable as:

$$y = \begin{cases} 0, & x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 < \text{threshold} \\ 1, & x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 \geq \text{threshold} \end{cases}$$

Is this starting to look familiar yet?

A simple decision, cont.

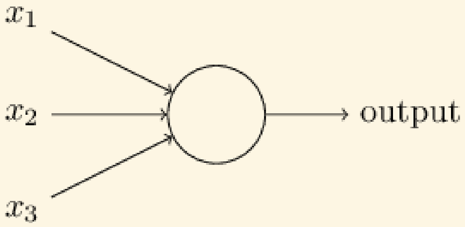
Now, if I rewrite this in terms of a dot product between the vector of of all binary inputs (x), a vector of weights (w), and change the threshold to the negative bias (b), we have:

$$y = \begin{cases} 0, & x \cdot w + b < 0 \\ 1, & x \cdot w + b \geq 0 \end{cases}$$

So we are really just finding separating hyperplanes again, much as we did with logistic regression and support vector machines!

A perceptron

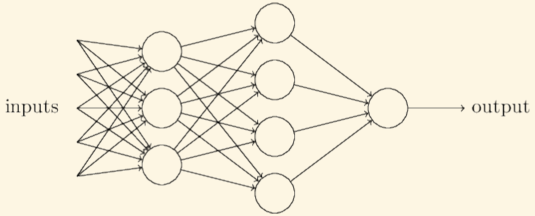
We can graphically represent this decision algorithm as an object that takes 3 binary inputs and produces a single binary output:



This object is called a perceptron when using the type of weighting scheme we just developed.

A network of perceptrons

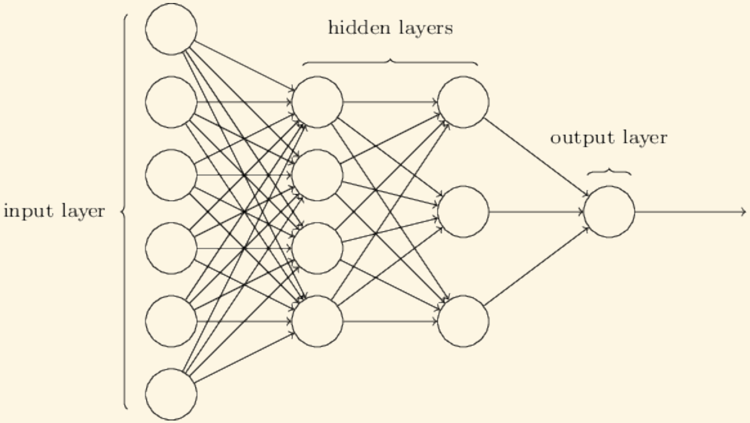
A perceptron takes a number of binary inputs and emits a binary output. Therefore it is easy to build a network of such perceptrons, where the output from some perceptrons are used in the inputs of other perceptrons:



Notice that some perceptrons seem to have multiple output arrows, even though we have defined them as having only one output. This is only meant to indicate that a single output is being sent to multiple new perceptrons.

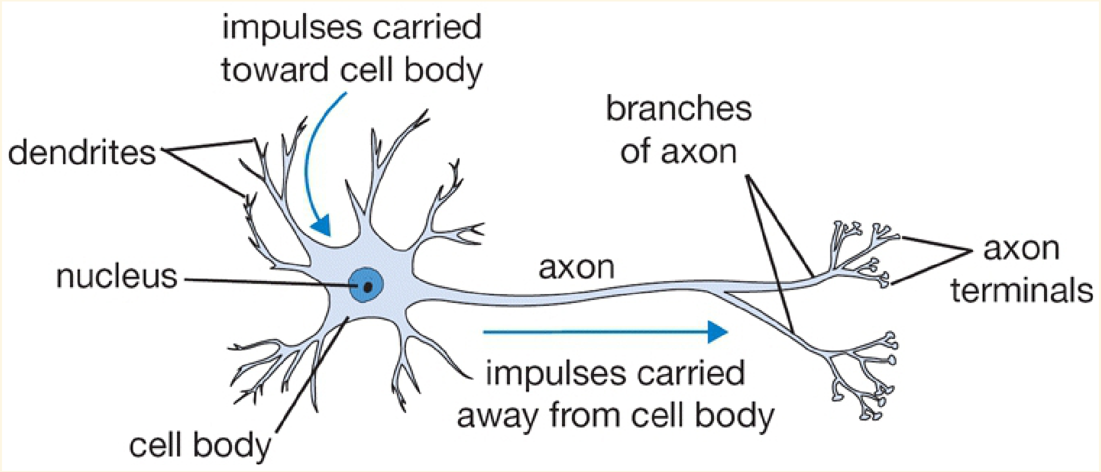
A network of perceptrons, cont.

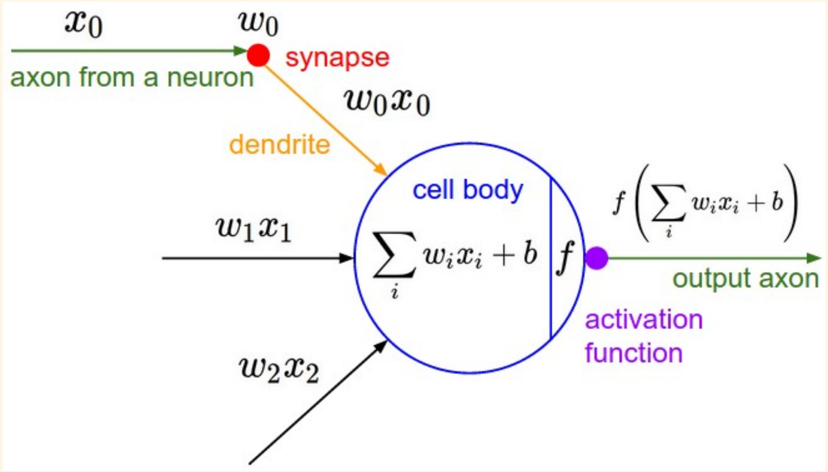
The input and outputs are typically represented as their own neurons, with the other neurons named hidden layers



A network of perceptrons, cont.

The biological interpretation of a perceptron is this: when it emits a 1 this is equivalent to ‘firing’ an electrical pulse, and when it is 0 this is when it is not firing. The bias indicates how difficult it is for this particular node to send out a signal.


impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body


x_0
w_0
synapse
axon from a neuron
$w_0 x_0$
dendrite
cell body
$w_1 x_1$
$\sum_i w_i x_i + b$
f
$f\left(\sum_i w_i x_i + b\right)$
output axon
activation function
$w_2 x_2$

A network of perceptrons, cont.

Notice that the network of nodes I have shown only sends signals in one direction. This is called a feed-forward network. These are by far the most well-studied types of networks, though we will (hopefully) have a chance to talk about recurrent neural networks (RNNs) that allow for loops in the network. The one-directional nature of feed-forward networks is probably the biggest difference between artificial neural networks and their biological equivalent.

Sigmoid neuron

An important shortcoming of a perceptron is that a small change in the input values can cause a large change the output because each node (or neuron) only has two possible states: 0 or 1. A better solution would be to output a continuum of values, say any number between 0 and 1.

Most tutorials spend a significant amount of time describing the conceptual leap from binary outputs to a continuous output. For us, however this should be quite straightforward.

Sigmoid neuron, cont.

As one option, we could simply have the neuron emit the value:

$$\sigma(x \cdot w + b) = \frac{1}{1 + e^{-(x \cdot w + b)}}$$

For a particularly positive or negative value of $x \cdot w + b$, the result will be nearly the same as with the perceptron (i.e., near 0 or 1). For values close to the boundary of the separating hyperplane, values near 0.5 will be emitted.

Sigmoid neuron, cont.

This perfectly mimics logistic regression, and in fact uses the logit function to do so. In the neural network literature, the logit function is called the sigmoid function, thus leading to the name sigmoid neuron for a neuron that uses it's logic.

Notice that the previous restriction to binary *inputs* was not at all needed, and can be easily replaces with continuous input without an changes needed to the formulas.

Activation functions

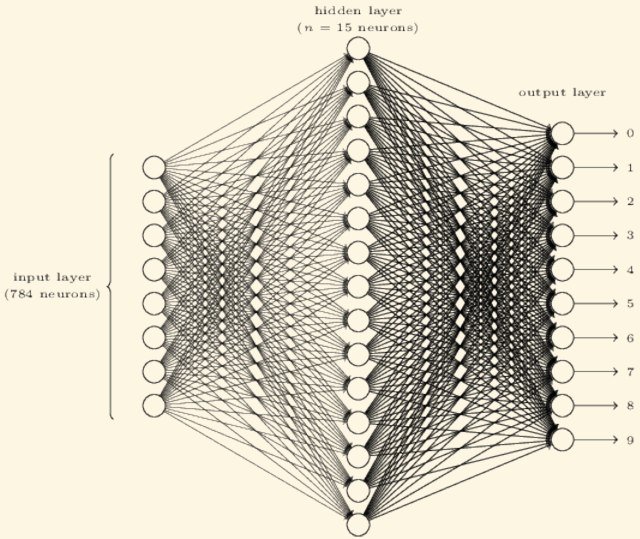
In the sigmoid neuron example, the choice of what function to use to go from $x \cdot w + b$ to an output is called the activation function. Using a logistic, or sigmoid, activation function has some benefits in being able to easily take derivatives and the interpret them using logistic regression.

Other choices have certain benefits that have recently grown in popularity. Some of these include:

1. hyperbolic tan: $tanh(z) = 2\sigma(2x) - 1$
2. rectified linear unit: $ReLU(z) = max(0, z)$
3. leaky rectified linear unit
4. maxout

We will explore the pros and cons of these in upcoming lectures.

MNIST example



MNIST example, classification

To determine which class to put a particular input into, we look at which of the output neurons have the largest output.

Learning neural networks

We now have an architecture that describes a neural network, but how do we learn the weights and bias terms in the model given a set of training data?

As an important side note, notice that with just one node, we could define a learning algorithm which perfectly replicates a support vector machine or logistic regression.

Cost function

The primary set-up for learning neural networks is to define a cost function (also known as a loss function) that measures how well the network predicts outputs on the test set. The goal is to then find a set of weights and biases that minimizes the cost.

Cost function, cont.

One example of a cost function is just squared error loss:

$$C(w, b) = \frac{1}{2n} \sum_i (y_i - \widehat{y}(x_i))^2$$

Or, for classification, the hinge loss:

$$C(w, b) = \sum_i \left[1 - y_i \cdot \widehat{y}(x_i)\right]_+$$

As with the activation functions, we'll explore the different cost functions over the the next several weeks.

Optimization problem

How does one actually do the optimization required in fitting neural networks? With very few exceptions, every technique is somehow related to gradient descent. That is, we calculate the gradient function, move a small amount in the opposite direction of the gradient (because we are minimizing), and then recalculate the gradient on the new spot.

Gradient descent

Mathematically, we can describe these updates as:

$$w_{k+1} = w_k - \eta \cdot \nabla_w C$$
$$b_{k+1} = b_k - \eta \cdot \nabla_b C$$

For some value $\eta > 0$. This tuning parameter, as in gradient boosted trees, is called the learning rate. Too low, and learning takes a very long time. Too small, and it is likely to have trouble finding the true minimum (as it will keep 'overshooting' it).

Decomposable cost function

One particularly important aspect of all of the cost functions used in neural networks is that it the are able to be decomposed over the samples. That is:

$$C = \frac{1}{n} \sum_i C_i$$

For the individual costs C_i of the i'th sample.

Decomposable cost function, cont.

Consider now taking a subset $M \subseteq \{1, 2, \dots n\}$ with size m of the training set. It would seem that we can approximate the cost function using only this subsample of the data:

$$\frac{\sum_{i \in M} \nabla C_i}{m} \approx \frac{\sum_{i=1}^{n} \nabla C_i}{n} \approx \nabla C$$

So it seems that we can perhaps estimate the gradient using only a small subset of the entire training set.

Stochastic gradient descent (SGD)

Stochastic gradient descent uses this idea to speed up the process of doing gradient descent. Specifically, the input data are randomly partitioned into disjoint groups $M_1, M_2, \ldots, M_{n/m}$. We then do the following updates to the weights (biases are done at the same time, but omitted for sake of space):

$$\begin{aligned} w_{k+1} &= w_k - \frac{\eta}{m} \sum_{i \in M_1} \nabla C_i \\ w_{k+2} &= w_{k+1} - \frac{\eta}{m} \sum_{i \in M_2} \nabla C_i \\ &\vdots \\ w_{k+n/m+1} &= w_{k+n/m} - \frac{\eta}{m} \sum_{i \in M_{n/m}} \nabla C_i \end{aligned}$$

Stochastic gradient descent (SGD)

Stochastic gradient descent uses this idea to speed up the process of doing gradient descent. Specifically, the input data are randomly partitioned into disjoint groups $M_1, M_2, \ldots, M_{n/m}$. We then do the following updates to the weights (biases are done at the same time, but omitted for sake of space):

$$\begin{aligned} w_{k+1} &= w_k - \frac{\eta}{m} \sum_{i \in M_1} \nabla C_i \\ w_{k+2} &= w_{k+1} - \frac{\eta}{m} \sum_{i \in M_2} \nabla C_i \\ &\vdots \\ w_{k+n/m+1} &= w_{k+n/m} - \frac{\eta}{m} \sum_{i \in M_{n/m}} \nabla C_i \end{aligned}$$

Each set M_j is called a mini-batch and going through the entire dataset as above is called an epoch.

Stochastic gradient descent (SGD), cont.

You'll notice that the algorithm you need to implement in problem set 4 uses stochastic gradient descent to find a solution the support vector machine optimization problem.

Stochastic gradient descent (SGD), cont.

The main tuning parameters for this technique are the size of the mini-batch (m), the learning rate (η) and the number of epochs (E) to use. Again, we will discuss these in the upcoming weeks by way of many examples.

Note: Some texts refer to SGD as only describing the case with a mini-batch size of 1, whereas others use SGD to refer to this more generic algorithm. I believe that the technical term for the generic algorithm is *mini-batch gradient descent* (MGD), but you will rarely hear the term.

Calculating the gradient function

The last bit that we need in order to run SGD on a neural network is a way to actually calculate the gradient for a given mini-batch. This will be the sole focus of Wednesday's lecture, when I derive the equations that allow us to calculate the gradient in a particularly efficient way.

SGD Example: OLS

To simulate stochastic gradient descent, consider using it to find the ordinary least squares estimator of a multivariate regression function:

$$\begin{aligned} f(\beta) &= \frac{1}{n} \cdot \sum_i f_i(\beta) \\ &= \frac{1}{n} \cdot \sum_i (y_i - x_i\beta)^2 \\ &= \frac{1}{n} \cdot \sum_i (y_i^2 + \beta^t x_i^t x_i \beta - 2 y_i x_i^t \beta) \end{aligned}$$

SGD Example: OLS, cont.

Now, the gradient of f is given by:

$$\begin{aligned}\nabla f &= \frac{1}{n} \cdot \sum_i f_i(\beta) \\ &= \frac{1}{n} \cdot \sum_i \nabla f_i(\beta) \\ &= \frac{2}{n} \cdot \sum_i (x_i^t x_i \beta - x_i^t y)\end{aligned}$$

And can be approximated by:

$$\nabla f \approx \frac{2}{m} \cdot \sum_{i \in M} (x_i^t x_i \beta - x_i^t y)$$

For a mini-batch M of size m.