Introduction to Neural Networks

Overview

- Logistic regression
- Introducing neural network architecture
- High level description of how to learn neural networks and specific challenges in doing so
- A simple example

Logistic Regression

What is Logistic Regression?

Linear regression

type of regression we use for a continuous, normally distributed response variable

Logistic regression

the type of regression we use for a binary response variable that follows a Bernoulli distribution

Bernoulli distribution

- $Y \sim Bernoulli(p)$ takes values in $\{0, 1\}$,
  - e.g. a coin toss
- $Y = 1$ for a success, $Y = 0$ for failure,
- $p =$ probability of success, i.e. $p = P(Y = 1)$,
  - e.g. $p = \frac{1}{2} = P(\text{heads})$
- Mean is p, Variance is $p(1 - p)$.

Bernoulli probability density function (pdf):

$$f(y; p) = \begin{cases} 1 - p & \text{for } y = 0 \\ p & \text{for } y = 1 \end{cases}$$
$$= p^y (1 - p)^{1-y}, \quad y \in \{0, 1\}$$

Why can’t we use Linear Regression for binary outcomes?

- The relationship between X and Y is not linear.
- The response Y is not normally distributed.
- The variance of a Bernoulli random variable depends on its expected value p_X.
- Fitted value of Y may not be 0 or 1, since linear models produce fitted values in $(-\infty,+\infty)$

Logistic Regression Model

- Let Y be a binary outcome and X a covariate/predictor.
- We are interested in modeling $p_x = P(Y = 1|X = x)$, i.e. the probability of a success for the covariate value of $X = x$.

Define the **logistic regression model** as

$$logit(p_X) = \log\left(\frac{p_X}{1 - p_X}\right) = \beta_0 + \beta_1 X$$

- $\log\left(\frac{p_X}{1-p_X}\right)$ is called the **logit** function
- $p_X = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}}$
- $\lim_{x\to-\infty} \frac{e^x}{1+e^x} = 0$ and $\lim_{x\to\infty} \frac{e^x}{1+e^x} = 1$, so $0 \leq p_x \leq 1$.

A simple decision

Say you want to decide whether you are going to attend a music concert 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

We 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

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

$$x_1 . w_1 + x_2 . w_2 + x_3 . w_3 >= \text{Threshold}$$

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

A simple decision

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

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

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

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

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

A simple decision

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

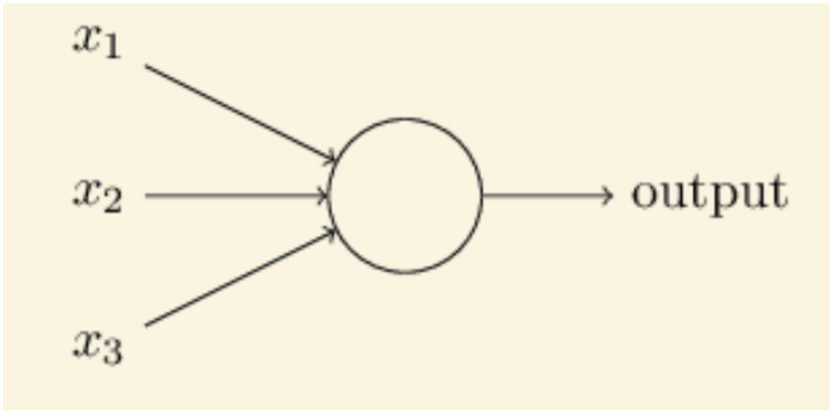
$$y = \begin{cases} 0, & x \cdot w + b < 0 \\ 1, & x \cdot w + b \geq 0 \end{cases}$$

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

Perceptron

A Perceptron

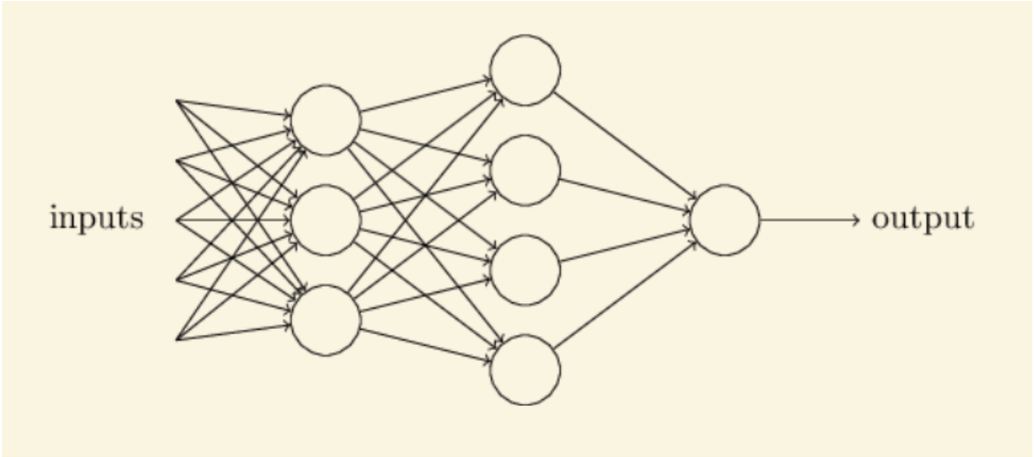
We can graphically represent this decision algorithm as an object that takes 3 binary inputs and produces a single binary output:



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

A Network of Perceptrons

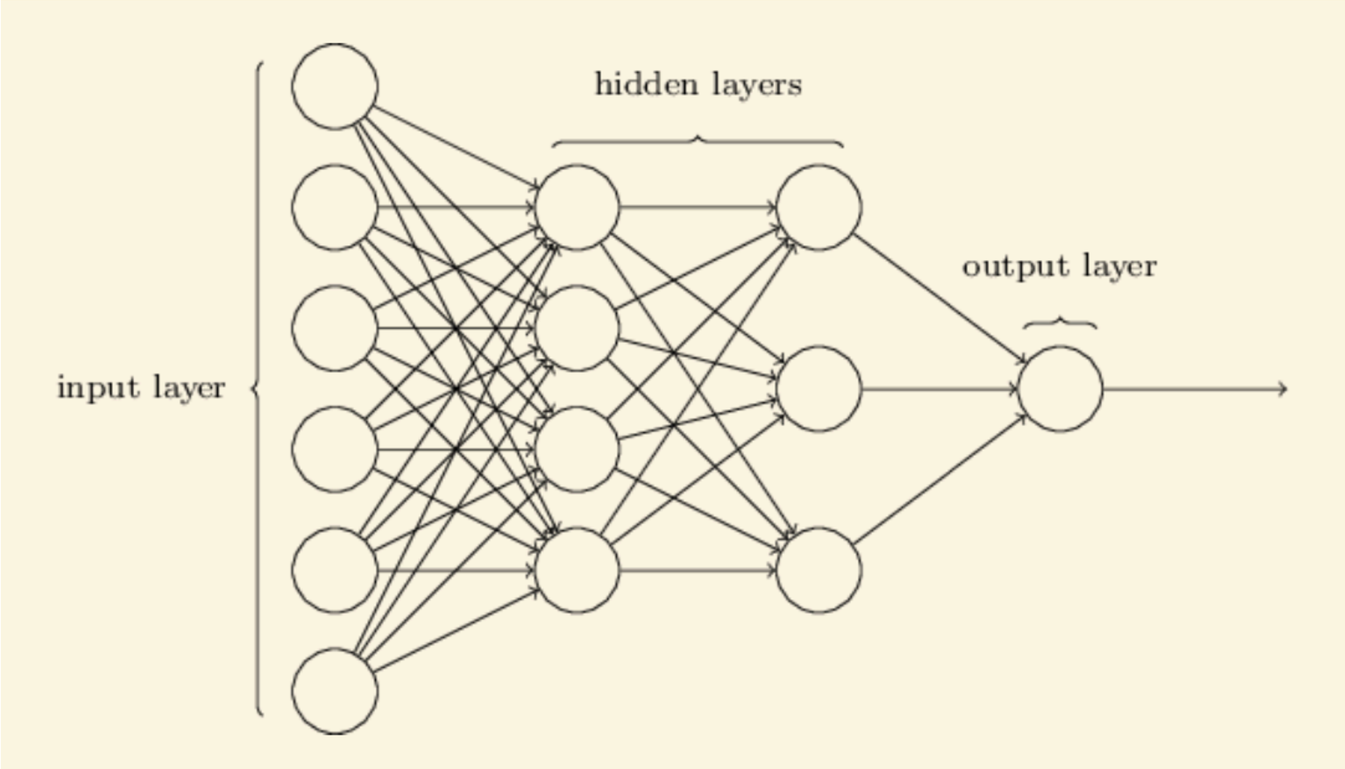
A perceptron takes a number of binary inputs and emits a binary output. Therefore it is easy to build a network of such perceptrons, where the output from some perceptrons are used in the inputs of other perceptrons:



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

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

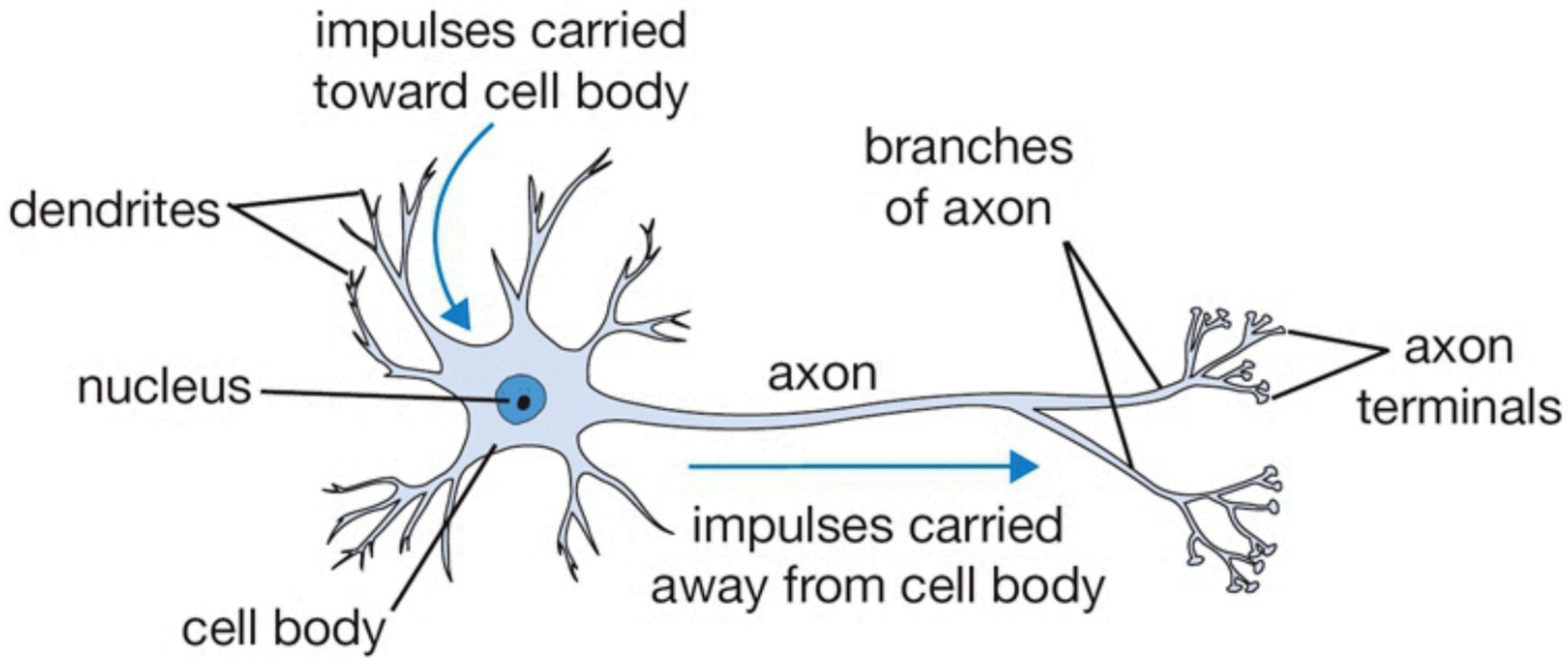


A Network of Perceptrons

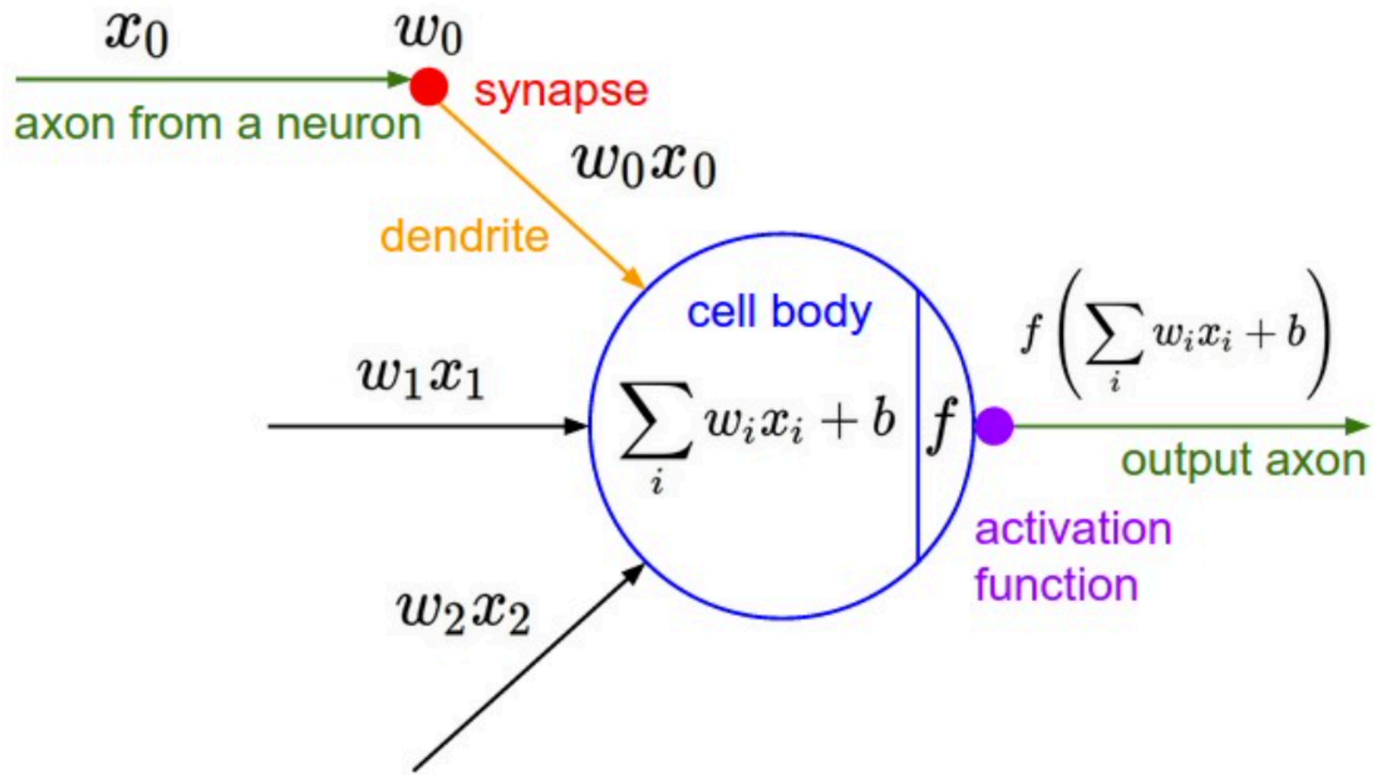
The biological interpretation of a perceptron:

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.

A Real Neuron



An Artificial Neuron



A Network of Perceptrons

Notice that the network of nodes shown above 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 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.

Sigmoid Neuron

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 . 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

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 any changes needed to the formulas.

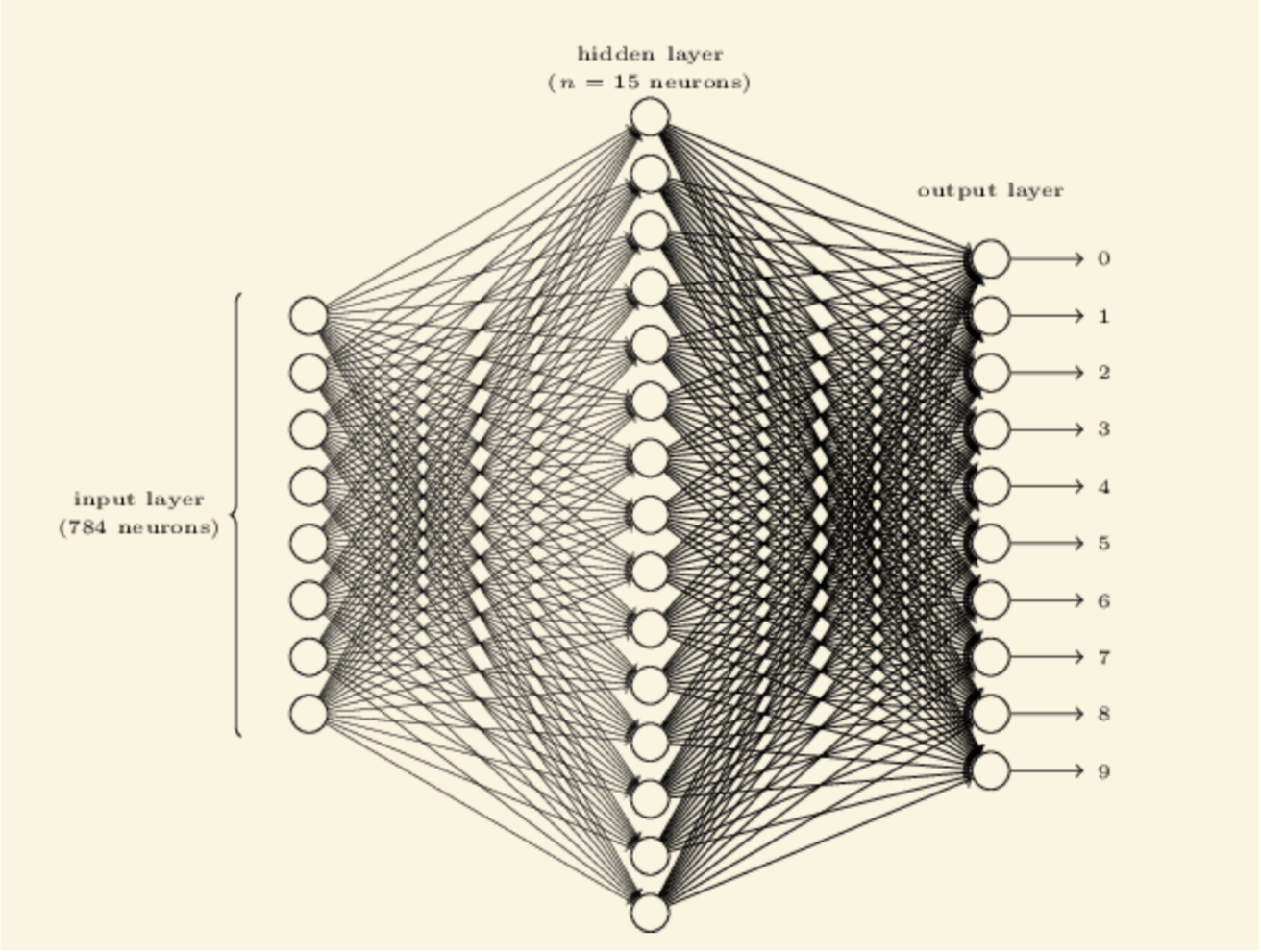
Activation Functions

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: $\text{ReLU}(z) = \max(0, z)$
3. leaky rectified linear unit
4. maxout

Example NN


hidden layer
(n = 15 neurons)
output layer
input layer
(784 neurons)
0
1
2
3
4
5
6
7
8
9

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

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:

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

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

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

Consider now taking a subset $M \subseteq \{1, 2, \ldots n\}$ 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, . . . , $M_{n/m}$. 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.

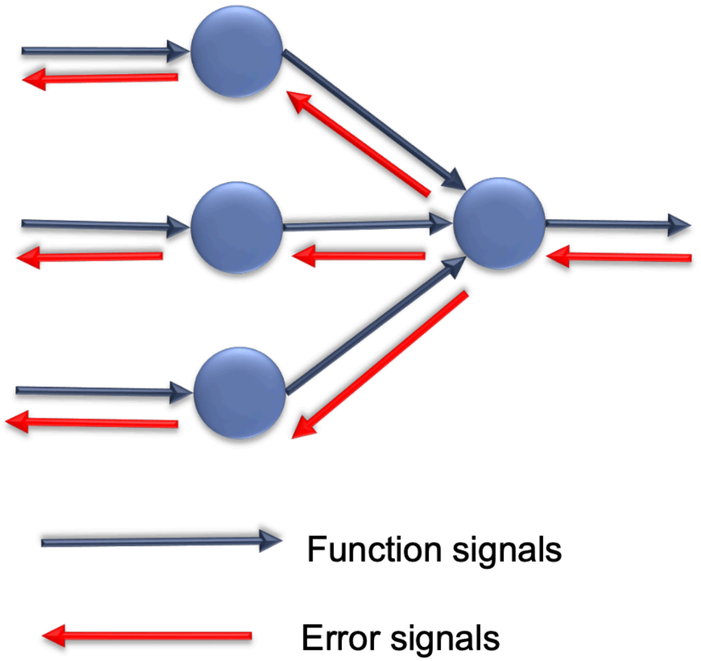
Stochastic Gradient Descent (SGD)

The main tuning parameters for this technique are:

- the size of the mini-batch (m)
- the learning rate (η)
- the number of epochs (E) to use.

NN Training - Signal Propagation

- Function Signals
  - An input signal that is fed to the system through inputs and propagates forward and emerges at the output
- Error Signals
  - An error that originates at the output and propagates backwards through the network



NN Training Phases

- Forward Phase
  - The inputs for a training instance are fed into the neural network. The final predicted output can be compared to that of the training instance and the derivative of the loss function with respect to the output is computed.
- Backward Phase (Back-propagation)
  - The main goal of the backward phase is to learn the gradient of the loss function with respect to the different weights by using the chain rule of differential calculus. These gradients are used to update the weights.

Summary

- Logistic Regression
- Formation of the concept - A decision making example
- Perceptron
- Network of Perceptrons
- Activation Functions
- Gradient Descent

More Resources

- 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

References

- Slides modified from “Introduction to Neural Networks” by Taylor B. Arnold. 2016. Yale Statistics.
- Slides modified from “An Introduction to Logistic Regression” by Emily Hector. 2019. University of Michigan.
- Slides modified from “Neural Networks” by Nisansa de Silva. University of Moratuwa.