Neural Networks: Overview

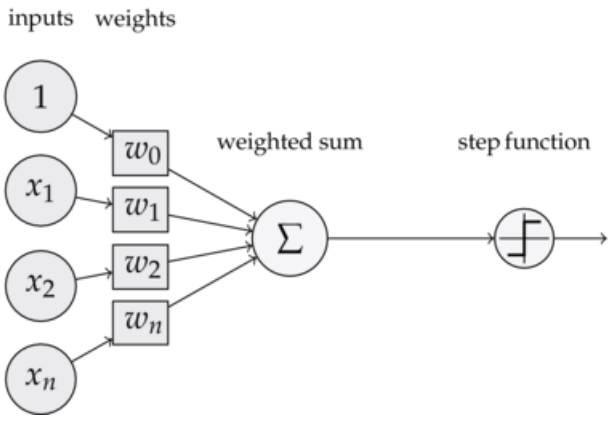
Nikhil Sardana and Mihir Patel

November 2017

1 Introduction

Neural networks are fundamental to modern machine learning. In order to understand Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Generative Adversarial Networks (GANs), it is essential to understand the theory behind Neural Networks. This lecture presents a high-level overview of neural networks. For a more in-depth look into neural networks, see `tjmachinelearning.com` for the Neural Network lecture series.

2 The Perceptron



2.1 Definition

A perceptron is the fundamental unit of a Neural Network (which is even called a Multi-Layer Perceptron for this reason). Refer to the diagram above. Perceptrons contain two or more inputs, a weight for each input, a bias, an activation function (the step function) and an output. For the perceptron above with 2 inputs, the intermediate value $f(x)$ is as follows

$$f(x) = w_1x_1 + w_2x_2 + b$$

The final output y is just the step function:

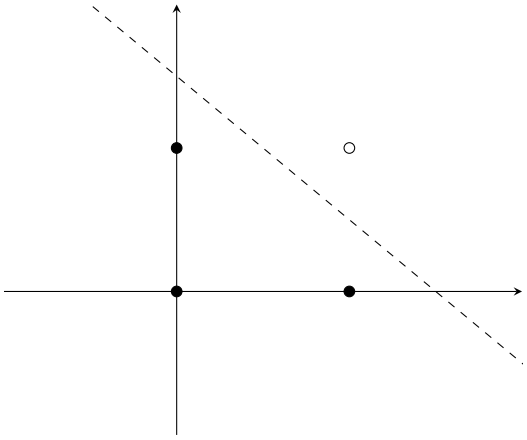
$$y = \begin{cases} 0 & \text{if } f(x) < 0 \\ 1 & \text{if } f(x) > 0 \end{cases}$$

2.2 Visualization

The purpose of a perceptron is to classify data. Consider the function AND.

x1	x2	out
0	0	0
0	1	0
1	0	0
1	1	1

Let's graph this data.



The line $y = -x + 1.5$ splits this data the best. Let's rearrange this to get $x + y - 1.5 = 0$. Going back to the perceptron formula

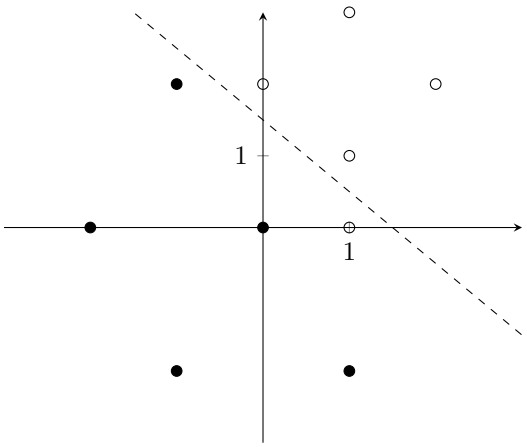
$$f(x) = w_1x_1 + w_2x_2 + b$$

we can see that for the optimal perceptron, w_1 and w_2 are the coefficients of x and y, and $b = -1.5$. If $f(x) > 0$, then $x + y - 1.5 > 0$. We can see through this example that perceptrons are nothing more than linear functions. Above a line, perceptrons classify data points 1, below the line, they are 0.

2.3 Learning

How do perceptrons "learn" the best possible linear function to split the data? Perceptrons adjust the weights and bias to iteratively approach a solution.

Let's consider this data:



The perceptron that represents the dashed line $y + x - 1.5 = 0$ has two inputs, x_1, x_2, with corresponding weights $w_1 = 1, w_2 = 1$, and bias $b = -1.5$. Let y represent the output of this perceptron. In the data above, the point $(1, 0)$ is the only misclassified point. The perceptron outputs 0 because it is below the line, but it should output a 1.

For some data point (input) i with coordinates (i_1, i_2), the perceptron adjusts its weights and bias according to this formula:

$$w_1 = w_1 + \alpha(d - y)(i_1)$$
$$w_2 = w_2 + \alpha(d - y)(i_2)$$
$$b = b + \alpha(d - y)$$

Where d is the desired output, and α is the learning rate, a constant usually between 0 and 1. Notice that the equation degenerates to $w = w$ and $b = b$ when the desired output equals the perceptron output. In other words, the perceptron only learns from misclassified points.

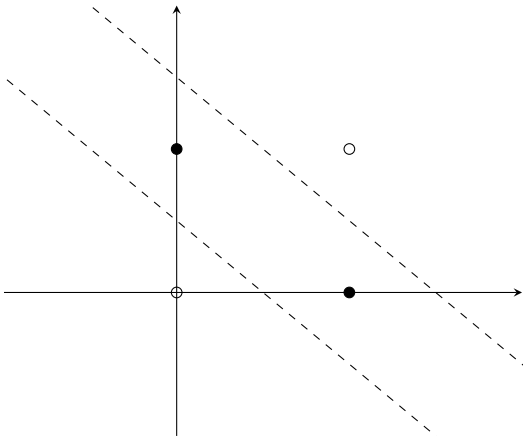
3 Multi-Layer Perceptron

3.1 Non-Linearly Separable Data

Consider the function XOR:

x1	x2	out
0	0	1
0	1	0
1	0	0
1	1	1

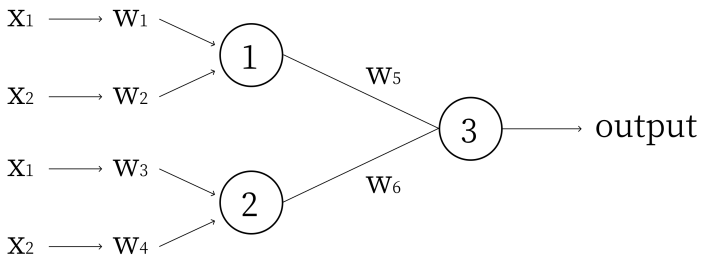
Let's graph this data.



We need two lines to separate this data! A perceptron will never reach the optimal solution. However, multiple perceptrons can learn multiple lines, which can be used to classify non-linearly separable data.

A neural network (NN) or Multi-Layer Perceptron (MLP) is a bunch of these perceptrons glued together, and can be used to approximate multi-dimensional non-linearly separable data. Let us again consider XOR. How do we arrange perceptrons to represent the two functions?

Clearly, we need two perceptrons, one for each function. The output of these two perceptrons can be used as inputs to a third perceptron, which will give us our output. Refer to the diagram below.



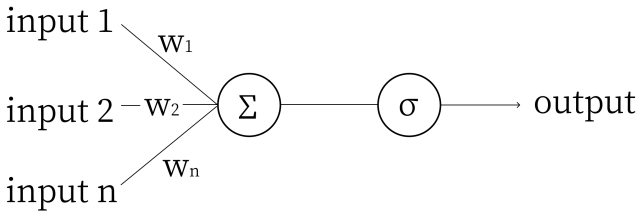
Let perceptron 1 model $y + x - 1.5 = 0$ (the upper line), and perceptron 2 model $y + x - 0.5 = 0$ (the lower line). Because the weights are the coefficients of these functions, $w_1 = 1, w_2 = 1, w_3 = 1, w_4 = 1$ and the biases $b_1 = -1.5$ and $b_2 = -0.5$.

The output of Perceptron 1 will be a 1 for points above the upper line, and a 0 for the points below the upper line. The output of Perceptron 2 will be a 1 for points above the lower line, and a 0 for points below the lower line. Thus, above both lines, we get 2. In between the lines, we get 1. Below the lines, we get 0. However, in order to create a threshold to separate the points between the lines from the points outside, we would like the outputs for points between the lines to be additive.

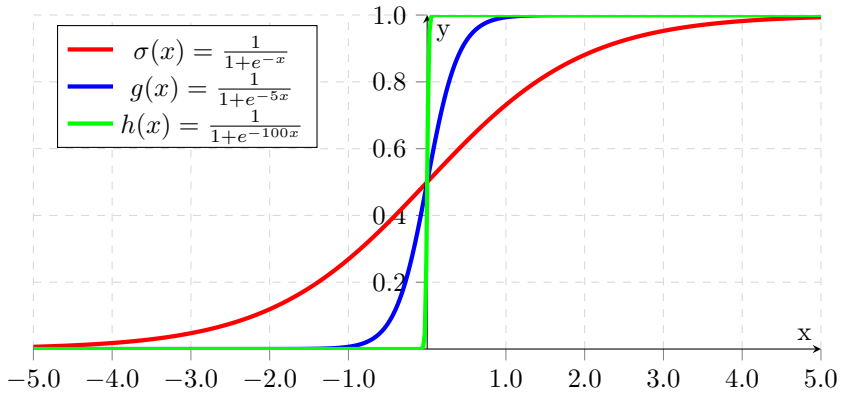
In other words, we would like the inputs of Perceptron 3 to cancel outside the lines, and have a maximum for points inside the lines. Thus, we let $w_6 = 1$ and $w_5 = -1$. This gives us an output of 1 for points between the lines, and an output of 0 for points outside the lines. Thus, we can set the bias for Perceptron 3: $b_3 = -0.5$.

4 The Neuron

A single node of a neural network (a neuron) differs from a perceptron in one way: the activation function. Consider this diagram of a neuron:



The symbol σ represents the Sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$.



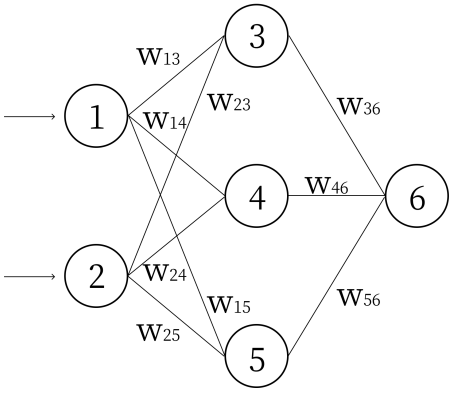
Notice how as the coefficent of x approaches infinity, $\sigma(x)$ approaches the step function from before. We use $\sigma(x)$ is because it is differenciable, which is necessary for networks to learn. Other activation functions include $\tanh(x)$ and ReLU, but we will use Sigmoid for our examples.

The rest of a neuron is identical to a perceptron: multipy each input by its weight, add them up and the bias and compute the activation function of the sum.

5 Forward Propagation

5.1 Non-Vectorized Forward Propagation

Forward Propagation is a fancy term for computing the output of a neural network. We must compute all the values of the neurons in the second layer before we begin the third, but we can compute the individual neurons in any given layer in any order. Consider the following network:



We denote the value of node i as n_i, and the bias of node i as b_i. Computing the network using these variables, we get:

$$n_3 = \sigma(w_{13}n_1 + w_{23}n_2 + b_3)$$

$$n_4 = \sigma(w_{14}n_1 + w_{24}n_2 + b_4)$$

$$n_5 = \sigma(w_{15}n_1 + w_{25}n_2 + b_5)$$

$$n_6 = \sigma(w_{36}n_3 + w_{46}n_4 + w_{56}n_5 + b_6)$$

Continuing this example of forward propagation, let's assign some numbers and compute the output of this network. Let $n_1 = 0.2$ and $n_2 = 0.3$. Let $w_{13} = 4, w_{14} = 5, w_{15} = 6, w_{23} = 5, w_{24} = 6, w_{25} = 7, w_{36} = 9, w_{46} = 10$ and $w_{56} = 11$, just so they are easy to remember. Let all the biases $b_{3..6} = 1$ (input nodes do not have biases, the "input nodes" are simply values given to the network). In practice, weights and biases of a network are initialized randomly between -1 and 1. Given these numbers, we compute:

$$n_3 = \sigma(4 * 0.2 + 5 * 0.3 + 1) = \sigma(3, 3) = 0.964$$

$$n_4 = \sigma(5 * 0.2 + 6 * 0.3 + 1) = \sigma(3.8) = 0.978$$

$$n_5 = \sigma(6 * 0.2 + 7 * 0.3 + 1) = \sigma(4.3) = 0.987$$

$$n_6 = \sigma(9 * 0.964 + 10 * 0.978 + 11 * 0.987 + 1) = \sigma(30.313) = 1$$

This example actually illustrates one of the weak points of the Sigmoid function: it quickly approaches 1 for large numbers.

6 Backpropagation

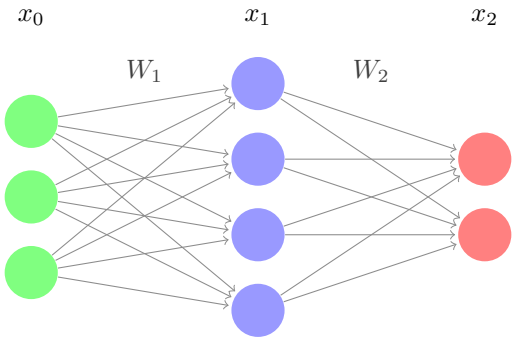
Backpropagation is how neural networks learn. We will present a high-level overview of backpropagation rather than a mathematically rigorous explanation. For a mathematically rigorous explanation, see "Neural Networks: Part 3" on `tjmachinelearning.com`.

6.1 Learning

A neural network learns when it is given training data and labels. The data (inputs) can be in the form of text, images, numbers, etc. The label is the ground truth, the correct answer for the given input. Given enough data-label pairs, a network can learn to generalize the relationship between the data and label. After training, it is tested or validated on a set of data it has never seen before (i.e. data not part of the training set). This validation accuracy shows just how well a network has learned to generalize through training. Backpropagation is the method of updating the weights and biases of the network to minimize the error when training.

6.2 Error

Consider the following network:



For the input x_0, let y represent the target vector, or the ground truth. We define the error as

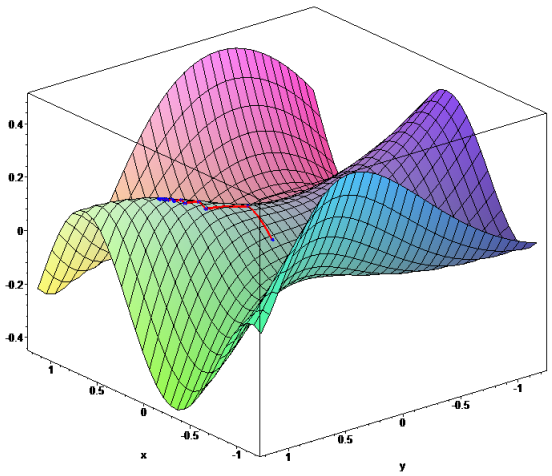
$$E = \frac{1}{2}||x_2 - y||^2$$

Essentially, this is the magnitude of the difference between the target and the network's output. In order for a network to become more accurate, we want to minimize this error.

Let's think of E as a function. Only x_2 can vary, and we can only control this by changing the weight matrices (and the bias). Thus, for a neuron with n weights and a bias, the error can be graphed as an $n+2$ dimensional function ($y = f(x)$ has 1 input, so it is graphed in two dimensions). For this network, each of the weights ($3*4+4*2 = 20$) and the biases (6) determines the error, so the error has many, many dimensions. If we get to the minimum of this function, we have minimized the error and trained the network.

6.3 Gradient Descent

We can't visualize that many dimensions (at least, I can't), so lets pretend we are working with a three dimensional function. How to we get to the minimum? We use gradient descent, of course!



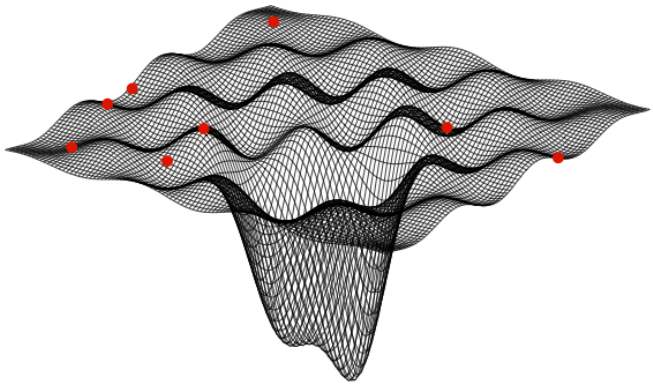
A multi-dimensional function. Look at that minimum!

Gradient descent is simple: Starting at some point, we move in the direction of steepest decline for a certain length. Then, at our new point, we again compute the direction of steepest decline, and move in that direction for a certain length. We repeat this process over and over until every single direction is an incline, at which point we are at the minimum.

This has three issues. First, how do we know how long our steps are? Take a step too long, and we could overshoot the minimum. Take a step too short and it will take us many steps to reach the minimum.

The step length is actually just a constant set by the programmer, and normally ranges from 0.1 to 0.0001. Adjusting the constant to get the best result is an important practical topic for getting the best result, and we will discuss it in Part 3 of the lecture. For now, just know its a constant.

Secondly, doesn't gradient descent just get us to a minimum? What if there are multiple minima, and we just happen to land in a local minimum, like the many in the function below?



Getting out of local minima to reach the global minimum is another important machine learning topic. Different optimizers can help the network pop out of local minima using momentum, but this topic is complex and modern, so it is covered later in this lecture. For the purposes of explaining gradient descent, we'll just pretend we're working with an error function with one minimum.

The third and final issue is: how do we know which direction is the steepest? We can't just sample each direction, as there are infinite possibilities. Instead, we mathematically compute the best direction by finding the gradient.

Let's do an example. Given $f(x, y) = 2x^2 + 3xy + y^3$, the partial derivatives are:

$$\frac{\partial f}{\partial x} = 4x + 3y$$

$$\frac{\partial f}{\partial x} = 3x + 3y^2$$

The gradient of $f(x, y)$, or $\nabla f(x, y)$ is just the vector:

$$(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$$

For our example, the gradient is:

$$(4x + 3y, 3x + 3y^2)$$

This is the direction of steepest ascent. How do we know that? First, lets consider the directional derivative. $\nabla_u f(x_0, y_0)$ is the rate of change of $f(x, y)$ at the point (x_0, y_0) in the direction $\vec{u}$. It is also defined in terms of the gradient as:

$$\nabla_u f(x_0, y_0) = \nabla f(x, y) \cdot \vec{u}$$

We know from our standard dot product rule:

$$\vec{a} \cdot \vec{b} = ||\vec{a}||||\vec{b}|| \cos(\theta)$$

And $\cos(\theta)$ is maximized at $\theta = 0$. Thus, when two vectors are in the same direction, their dot product is maximized. From this information, the maximum of the directional derivative must be when $\nabla f(x, y)$ and $\vec{u}$ are in the same direction. This means that the direction of the steepest ascent (maximum rate of change) is the direction of the gradient.

Great! Now our third issue has been solved. In order to find the minimum of a multi-dimensional function, we just need to compute the gradient, move in that direction for a certain length, and repeat until the gradient is 0.

The only problem is.... how do we compute the gradient? Our function is

$$E(W, b) = \frac{1}{2}||x_L - t||^2$$

Where x_L is the network output at t is the target. Since the error is in terms of the weights and biases, that means that we need to compute:

$$(\frac{\partial E}{\partial W_1}, \frac{\partial E}{\partial W_2}, ..., \frac{\partial E}{\partial b_n})$$

This is why backpropagation is a fundamental concept in machine learning. It allows us to compute this gradient in a computationally efficient manner.

6.4 Computing Backpropagation

This is the point where we take you through backpropagation, and how to compute the errors of a neural network. However, this takes too long, and is mathematically rigorous, so for now, just know that theres a method of calculating all the partial derivatives of the weights, and it's called backpropagation because we compute the errors of the weights back to front.

7 Optimizing Performance

Neural networks are very powerful tools that are one of the few types of machine learning approaches robust enough to tackle a variety of problems. Their method of learning, while requiring large amounts of data, can face virtually any challenge. However, they also have considerably more hyperparameters to tune compared to previous methods. How do we determine the number of layers? The number of nodes? How do we regularize? The answers to these questions are critical to squeezing out maximum performance.

8 Network Design

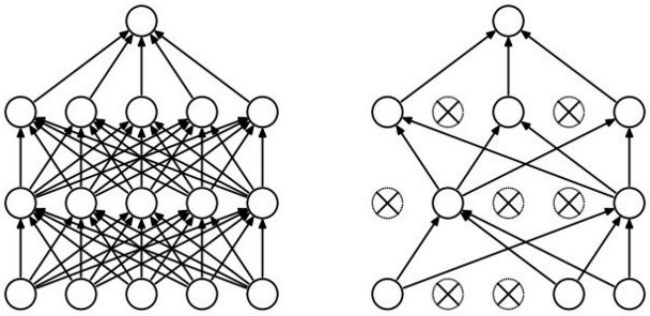
The first aspect to any structure is determining the size and shape of it. Unfortunately, there really is no good rule on how to do this, or we would have already automated out writing ML algorithms with networks. The obvious ones are the input and output layers, which have to match the specifications of the problem, pairing with the number of features and number of output classes respectively.

As for hidden layers, there really is no good answer. There are some useful tips however. In general, the deeper the network, the more advanced patterns it can learn but the more data and time it requires. Deeper networks are also much more prone to overfitting, as they could potentially just memorize large chunks of the dataset and not learn the general pattern. With deep learning and networks, we assume that our network will not overfit given good hyperparameters and so typically, the deeper the better as long as you have the computational resources for it.

Another good rule is to always narrow down the size of each layer. This helps the network condense information into important higher level features and saves computational resources as you need more lower level features than higher level features. In general, things like this simply just come with experience and trying new combinations if previous methods fail. Usually, the answer is just add more layers if you have the data for it.

9 Regularization

As I mentioned earlier, neural networks can be prone to overfitting, especially with deep networks. Overfitting, if you don't remember, involves a machine learning algorithm learning a sub-pattern or exploit that boosts training accuracy but diminishes knowledge of the overall problem and therefore hurts validation accuracy. The best way to prevent this is to stop training when validation accuracy begins to increase. However, we want to supress this from happening as long as possible so that we can potentially better learn the generalized pattern. The first approach involves modifying the loss function.



(a) Standard Neural Net (b) After applying dropout.

1. L1 regularization: Adds the absolute value of the network weights to the error function.

$$L + \lambda \sum_{i=1}^{k} |w_i|$$

   In this case, some hyperparameter lambda is multiplied by the absolute value of the weights. L1 has been shown to be better in feature selection for spare feature spaces. This is a bit complicated to demonstrate and involves Laplace transformations. If you're interested feel free to look it up.

2. L2 regularization: Adds the square of the network weights to the error function.

$$L + \lambda \sum_{i=1}^{k} w_i^2$$

   This has been shown to be more effective in basically every other scenario.

Both L1 and L2 regularization help prevent overfitting. λ controls the extent to which the regularization term matters and is generally around 0.01. They work by punishing large weights, preventing large numbers and adding small amounts of noise to help generalize the pattern.

10 Dropout

Another method for regularization is dropout. At each backpropagation step, some random percentage of nodes are ignored. See the diagram above to see how this works. This helps prevent highly dependent nodes and ensure each node actually learns something significant. Another way to think about this is stacking several different learners, where each network post-dropout is a different learner. The only difference is that all these networks are stacked into one model. This has been shown to produce several impressive improvements in performance and robustness. Good values of dropout range from 0.2 to 0.4 and depend on the layer type, with lower values for things like convolutional layers.

11 Batch Size

Often, large datasets might contain certain wrong data points or the pattern might not be easy to deduce from a single image (see MRI scans). To better adjust weights, we often forward propagate multiple data points before backpropagating on the results to stabilize training, preventing bad weight adjustments from poor data points.

By setting a batch size, we can control how many points we look at before backpropagating. In general, we pick the largest value that our machine supports based on its RAM size. The larger it is, the more stable the network becomes.

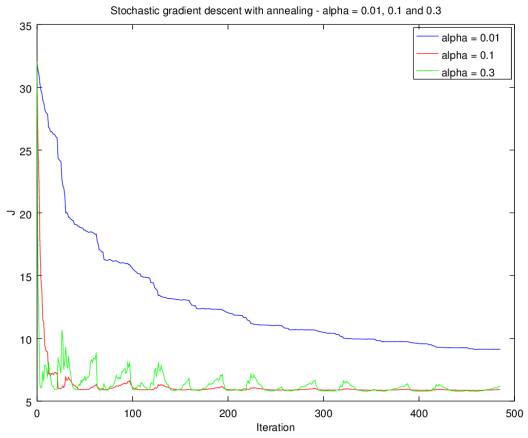
12 Learning Rate

Learning rate represents the α in the backpropagation for updating weights and biases:

$$W_i = W_i - \alpha \frac{\partial E}{\partial W_i}$$

$$b_i = b_i - \alpha \delta_i$$

This value determines how much we shift the weights at each step. A high learning rate means we might skip over the ideal weights (we can never reach 2.5 if we go from 2 to 3 at a step size of 1) and a low learning rate means it takes forever to train. A low learning rate means that we could also get stuck in a local minimum and not be able to climb out because each side has a higher error for our learning rate step size.

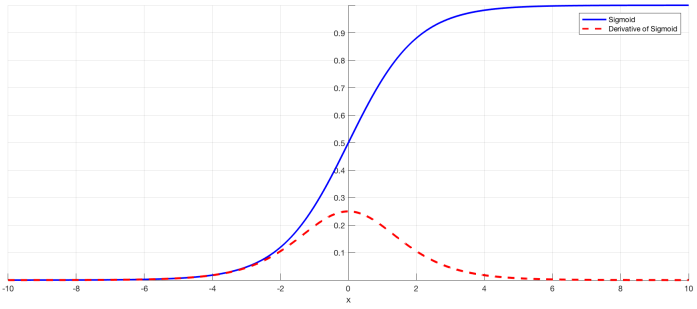


As this figure shows, a low learning rate (blue) means it goes very slowly. By contrast, a high rate (green) leads to instability and fluctuation. The perfect rate (red) has fast, stable training. In general, we pick a value and then adjust based on speed of training, looking at if the error is slow to adjust or sling-shotting around. This is problem specific and there is no good value, though most libraries provide a default.

In most cases, we use dynamic manipulations of the learning rate. Once we begin to see marginal improvements, we decrease the learning rate so that we can better reach the global minimum. The higher initial rate also speeds up training.

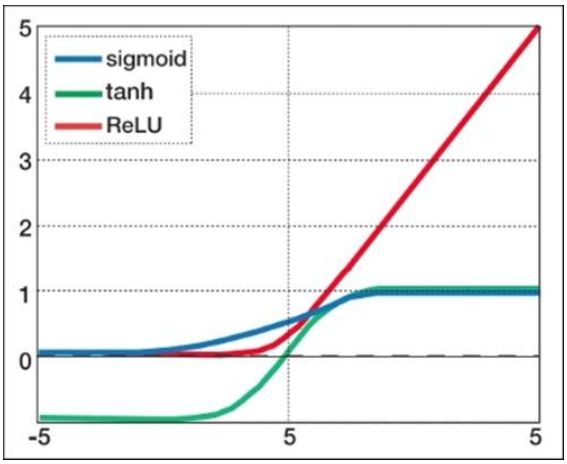
13 Vanishing Gradient Problem

Earlier I mentioned that adding more layers helps networks better learn patterns. It turns out this isn't exactly the case. When we reach deeper layers, we see that they begin to train slower and slower, quickly rendering them useless. Why is this the case? We know that once a node converges, the derivative of the sigmoid drops off very quickly. As we see from the graph, once a neuron learns its pattern and when to say 0/1, its derivative drops off. This is good, because we don't want to mess with that value. However, this means all subsequent layers have lower and lower deltas, learning much slower. This means we can't make things too deep.



14 Activation Functions

One way to fix the vanishing gradient problem is by modifying the activation function, which changes the derivative.



1. Sigmoid:

$$\frac{1}{1+e^{-}x}$$

This is the function we've been traditionally using and solid in general purpose.

2. Tanh:

$$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

This function has a more complicated derivative so it is a bit slower to compute. However, it prevents flatlining as the values range from -1 to 1 instead of 0 to 1. When you don't multiply by 0, the values aren't decreasing as much as much. This leads to better performance. There are a few proofs showing this.

3. Relu:

$$max(0,x)$$

This function looks very simple. It ignores everything negative and on the right just takes the positive value. This leads to much worse performance. However, the derivative is very, very easy to compute. It's 0 on the left of the y-axis and 1 on the right. This means it is very fast and doesn't have the vanishing gradient problem as much because the derivative is constant. In this way, we hurt the performance of each layer but just stack on a ton more layers.

15 Optimizers

We've seen that preventing getting stuck in local minima and efficiently updating weights is hard. To account for this, various modifications to the backpropagation algorithm have been propose to produce more efficient weight updating and increased stabilization. We won't go into how they work because it is decidedly non-trivial and as complex as learning backpropagation in the first place. However, there are a few key features that we will highlight.

1. Learning Rate: We already covered this! Go back a few pages.
2. Momentum: Momentum is a trick to prevent getting stuck in local minima. At each update, we factor in the change that we did the previous step, maintaining larger strides if our previous strides were big and vice-versa.
3. Decay: At each step, the weights are all multiplied by a constant less than one. This prevents exploding values just like L1 and L2 regularization.
4. Epsilon: At each step, some fuzz / noise is applied to the weights and biases, helping increase regularization.
5. Lots of other variables: Read the documentation for specific algorithms.

Here are a brief sample of some optimizers that are most commonly used. There are lots of other ones, though in general they are very ad-hoc or outdated.

1. (Stochastic/Mini-batch/Batch) Gradient Descent: Normal updates! Basically the run-of-the-mill algorithm that involves the least amount of computation.
2. RMSProp: Divides learning rate for a given weight by a running average the magnitude of recent gradients. Useful in RNNs.
3. Adam: Also adapts learning rates for given weights. Similar to RMSProp, but also includes momentum-like functionality. In general, this is the best one to use.