Backpropagation and Gradient Descent

Ava Lakmazaheri
Emily Lepert

January 20, 2018

1 Introduction

In order to solidify our knowledge at the end of Linearity II, we were interested in studying the interplay of linear algebra with computational algorithms. In this report, we investigate the mathematical formulation and application of gradient descent as an iterative optimization algorithm in order to train artificial neural networks for image classification.

2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models capable of capturing complex relationships between inputs and outputs, with the ability to change their parameters to "learn" better mappings for a given task. Common applications of neural networks include pattern recognition and image processing. For example, given a grayscale value for each pixel in an image, a trained neural network may be able to detect novel features and effectively classify the difference between a dog and a cat, even for picture it has never seen before!

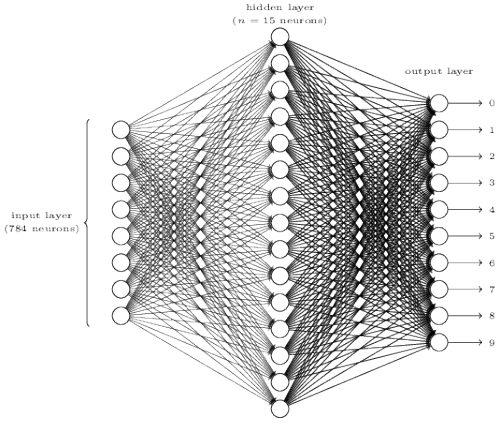


Figure 1: Neural Network

2.1 Basic Structure

As shown in Figure 1, this complex mapping system can be broken down into nodes and connections. Since artificial neural networks are loosely based upon the biology of the brain, nodes are referred to as **neurons** and their connections **synapses**.

Each neuron has an **activation** and a **bias**. Every synapse has a unique **weight**. In general terms, these components decide how strongly a neuron fires and how it then influences the neurons it is connected to.

It is further worth noting that neurons are grouped into layers. Any ANN contains up to three types of layers: the input layer (for data entry), output layer (the result), and hidden layers (for mapping). Note that any neuron will share synapses with every single neuron in the following layer.

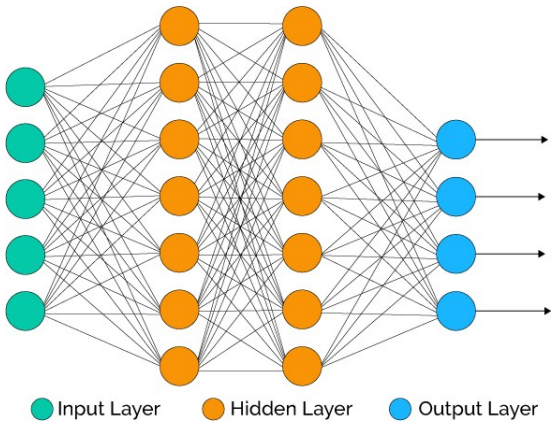


Figure 2: Neural Network Layers

We have introduced several terms without providing a clear description of the role they play in the network. Understanding how to interpret activations, biases, and weights as well as the behavior of hidden layers can be very challenging in abstraction. Here, we provide an extended metaphor for a more intuitive explanation of how information flows through a neural network.

2.2 The Chocolate Factor

In exciting local news, there is a chocolate festival coming to town! Luke and Leia are trying to make a decision about how likely they are to attend. What factors will be influencing their decision? What in turn decides those factors?

Let us consider a simplified model, in which every person's decision to attend this festival depends only on their peers making similar decisions.

In this network, each neuron represents a person. Their **activation** is how likely they are to attend the festival. The **weight** that corresponds with one person's connection to another has to do with how positively or negatively influenced they are by another person's decision to attend. For example, Luke really doesn't want to go to the festival if Darth Vader is going (a large negative weight connecting the two), but doesn't really care whether or not Han Solo attends (weight near zero). Likewise, assuming Leia is highly likely to go if Han Solo is going and not go if Obi-Wan does not; they would both be connected with a high positive weight.

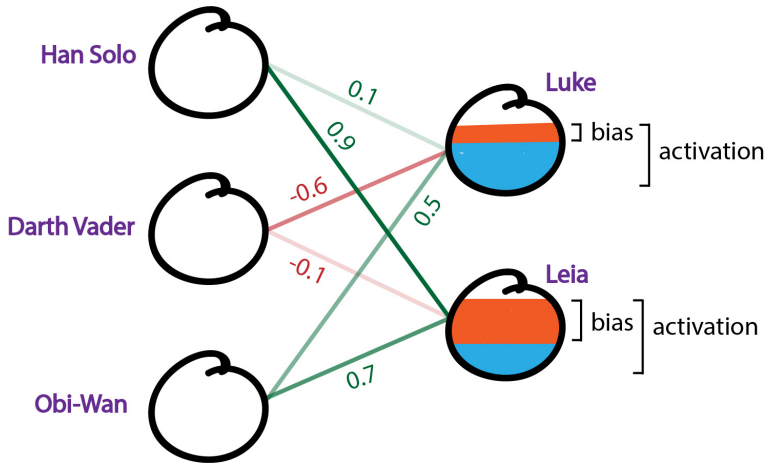


Figure 3: Analogy Diagram

In addition to this, everyone has a personal **bias** on if they will attend. This factor is more self-contained to the individual, for example, how much they love chocolate. If Leia has a chocolate affinity, her probability of attending the festival is higher regardless. In ANN terminology, this means her activation tends to be higher/the neuron is more likely to fire.

Note that each neuron's activation is *fully* decided by its bias plus the weighted sum of its synapses. When an artificial neural network is created, the only controlled parameters are the weights on each synapse and the bias of each neuron. The activations in place at any time arise directly as a result of these numbers.

This model assumes only two layers: the output (Luke and Leia) and the input (Darth Vader, Han Solo, and Obi-Wan). However, this system can get increasingly complex if we consider what factors influence Darth Vader, Han Solo, and Obi-Wan's decisions to attend. They

each have personal activations, weights to a preceding layer of people, and biases. With this metaphor, we can imagine an extremely complex neural network with hundreds of hidden layers mapping who knows who and who likes who.

2.3 Image Classification

A more popular machine learning task is the classification of handwritten numbers. Each neuron in the input layer corresponds to a pixel in the image whose activation is a grayscale value. Hidden layers can be used to identify edges in the image and further detect features such as lines, circles, curved edges, etc. These calculations come together to an output layer of 10 neurons, where the activation of each neuron gives the probability that the input image was the corresponding 0-9 digit.

With this real-life example, we can understand why an artificial neural network needs to be trained. There is only one correct classification of a handwritten 5, and the system must tweak its own parameters (i.e., its weights and biases) until it maps the input data to get this correct value.

3 Backpropagation and Gradient Descent

So how do artificial neural networks (ANN) learn? When ANN makes decisions, each layer of neurons interacts with the next to finally come to a "decision." But how does it know what values to give its weights, biases, and activations? One method is through backpropagation and gradient descent. Backpropagation is a method of **supervised learning** for deep neural networks. It needs three things:

1. An input and output dataset
2. An artificial neural network
3. A cost function

Backpropagation uses a method that consists of giving ANN data for which it already knows what the output should be, figures out how well ANN did in finding the right output through a cost function, computes the **gradient** of that cost function, and uses the gradient descent to make tweaks to the weights and biases of ANN. This learning is supervised because ANN

is learning based on already known data, and a program actively makes decisions on how it should change based on that known data.

3.1 Gradient

Gradient

Definition 1. Given a function $F(x, y, z)$ the gradient of F is:

$$\nabla F(x, y, z) = \frac{\partial F}{\partial x}\hat{\boldsymbol{\imath}} + \frac{\partial F}{\partial y}\hat{\boldsymbol{\jmath}} + \frac{\partial F}{\partial z}\hat{\boldsymbol{k}}$$

The gradient of a function points in the direction of steepest ascent. It indicates in which direction to go in to maximize the increase in the function. Gradient descent is simply the negative of gradient ascent. It is useful for backpropagation because once the cost function is computed, gradient descent indicates how to change all of the values to maximize the decrease of the cost function.

3.2 Notation and Activation calculation

3.2.1 Notation

To explain the math behind backpropagation, notation to describe all of the neurons, weights, and biases is necessary. Let's use a_j^L to denote the jth neuron in the Lth layer.

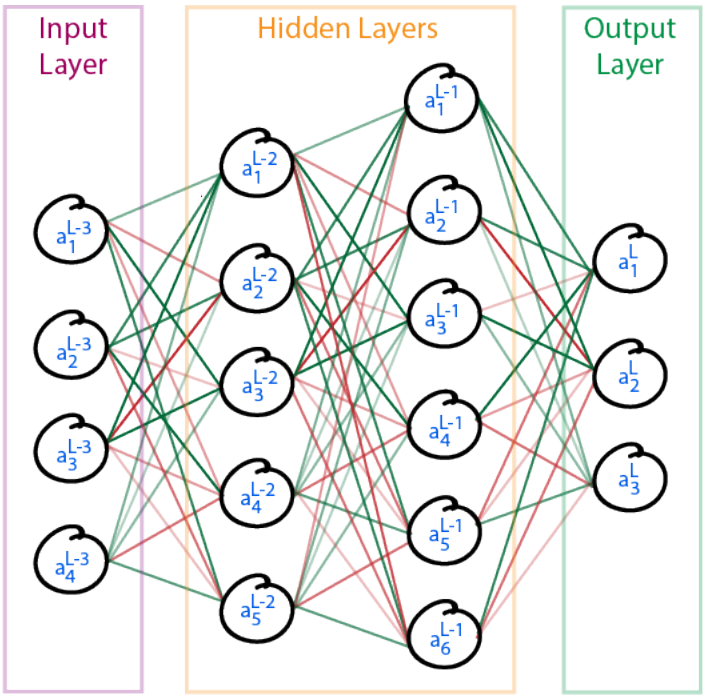


Figure 4: Neural Network

We'll use w_{jk}^L to denote the weight connecting the a_j^L neuron to the a_k^{L-1} neuron and b_j^L to denote the bias associated with the a_j^L neuron.

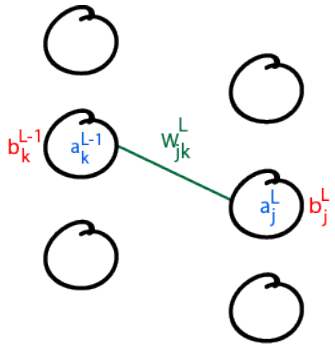


Figure 5: Simple Neural Network Notation

3.2.2 How to calculate the activation of a neuron

The ANN doesn't store the values of each neuron, rather it remembers the values of all the weights and biases and calculates the activation based on those values. Here is how an ANN

calculates the activation of one of its neurons:

$$a_j^L = \sigma(w_{j1}^L * a_1^{L-1} + ... + w_{jk}^L * a_k^{L-1} + b_j^L)$$

First we find the **weighted sum** which is the sum of all the activations of the previous layer, times the weights connecting those neurons to the neuron we're evaluating plus the bias associated with that neuron.

$$\text{weighted sum} = w_{j1}^L * a_1^{L-1} + ... + w_{jk}^L * a_k^{L-1} + b_j^L$$

We then apply the sigmoid function σ to normalize the weighted sum to a value between 0 and 1.

3.3 Cost function

The cost function tells the ANN how close its output values are to the expected one.

Cost Function

Definition 2. Given a dataset of inputs (i_n where n is the index of the input) and outputs (y_j where j is the index of the output) and all of the weights (w_{jk}) and biases (b_j^L) in all of the layers of the ANN, the cost function runs through the ANN with the inputs given and finds how well the ANN did. The function is:

$$C(w_{11}^1, b_1^1, ..., w_{jk}^L, b_j^L) = [(a_1^L - y_1)^2, (a_2^L - y_2)^2, ..., (a_m^L - y_m)^2]$$

Essentially, the cost function takes the square of the difference between the output layer values and the expected output value. If the value of the cost function is large, the ANN did poorly, if it is small, it did well.

3.4 Gradient Descent

We now have the cost function and all of the values of the weights and biases. We need to find the gradient of the cost function. This will tell us how to change each weight and bias value to reduce the cost function. Let's take a simple example of a network with one node in each hidden layer.

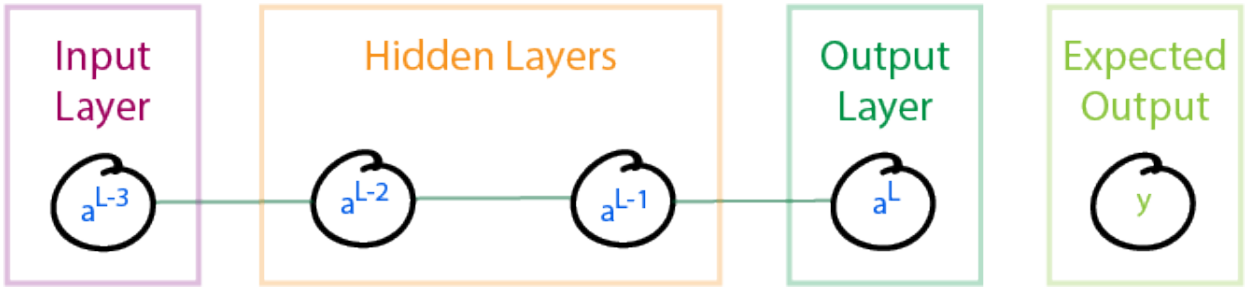


Figure 6: Very Simple Neural Network

The value of the cost function in this case would be:

$$C = (a^L - y)^2$$

Let's just consider the weight connecting a^L and a^{L-1}. Since the gradient descent tells us how much we should change weights and biases, it is useful to think of it as how much changing w affects C. In other words how big is $\frac{\partial C}{\partial w}$? We can't find this number directly however. Below is a tree of the intermediate steps we will need to go through.

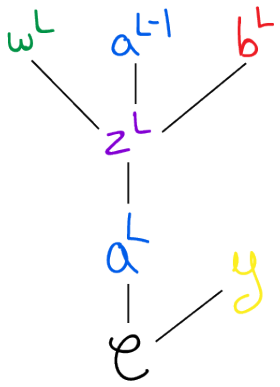


Figure 7: Backpropagation Tree

First, let's say that z^L is the weighted sum such that:

$$z^L = w^L * a^{L-1} + b^L$$

Here are all the values we will need:

$$z^L = w^L * a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$

$$C = (a^L - y)^2$$

The only values that directly affect C are a^L and y. We can't change y because it's a fixed value. We can't change a^L directly, however we can change it indirectly by changing z^L. We can't change z^L directly, but we can change w^L and b^L. So to find $\frac{\partial C}{\partial w^L}$ we'll need to see how much changing w^L changes z^L and then how much changing z^L changes a^L and then finally how much changing a^L changes C. To summarize:

$$\frac{\partial C}{\partial w^L} = \frac{\partial z^L}{\partial w^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C}{\partial a^L}$$

Using the values we found above, we can solve each partial derivative:

$$\frac{\partial z^L}{\partial w^L} = a^{L-1}$$

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L)$$

$$\frac{\partial C}{\partial a^L} = 2(a^L - y)$$

So the final solution is:

$$\frac{\partial C}{\partial w^L} = a^{L-1}\sigma'(z^L)2(a^L - y)$$

$\frac{\partial C}{\partial w^L}$ represents how much we'll want to change w^L by the next time we run through all of the data. This is the same thing as gradient descent. So we've found one of the values of our gradient descent! We still need to find $\frac{\partial C}{\partial b^L}$ and $\frac{\partial C}{\partial a^{L-1}}$.
We can find $\frac{\partial C}{\partial b^L}$ through the same process as with $\frac{\partial C}{\partial w^L}$ simply replacing a few terms. The terms that change are:

$$\frac{\partial C}{\partial b^L} = \frac{\partial z^L}{\partial b^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C}{\partial a^L}$$

$$\frac{\partial z^L}{\partial b^L} = 1$$

$$\frac{\partial C}{\partial b^L} = 1\sigma'(z^L)2(a^L - y)$$

Same thing again for $\frac{\partial C}{\partial a^{L-1}}$:

$$\frac{\partial C}{\partial a^{L-1}} = \frac{\partial z^L}{\partial a^{L-1}}\frac{\partial a^L}{\partial z^L}\frac{\partial C}{\partial a^L}$$

$$\frac{\partial z^L}{\partial a^{L-1}} = w^L$$

$$\frac{\partial C}{\partial a^{L-1}} = w^L \sigma'(z^L) 2(a^L - y)$$

Part of the gradient descent in this case can be represented as:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^L} \\ \frac{\partial C}{\partial b^L} \end{bmatrix}$$

Notice that we haven't included $\frac{\partial C}{\partial a^{L-1}}$ because we don't actually change a^{L-1} directly. This is where the idea of backpropagation comes in. The only thing that can change a^{L-1}'s value is z^{L-1}. And what changes z^{L-1} is w^{L-1}, b^{L-1}, and a^{L-2}, etc... To find out all of the ways we can change C we need to go backwards through our neural network.

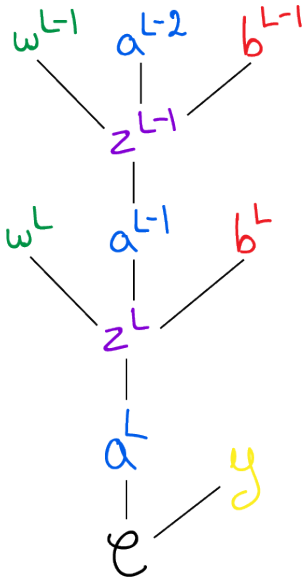


Figure 8: 2 layered tree

For example to find $\frac{\partial C}{\partial w^{L-1}}$ we would go through $\frac{\partial C}{\partial a^{L-1}}$:

$$\frac{\partial C}{\partial w^{L-1}} = \frac{\partial z^{L-1}}{\partial w^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}} \frac{\partial C}{\partial a^{L-1}} = \frac{\partial z^{L-1}}{\partial w^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}} \frac{\partial z^L}{\partial a^{L-1}} \frac{\partial a^L}{\partial z^L} \frac{\partial C}{\partial a^L}$$

If we zoom back out to a full neural net with multiple neurons in each layer, the full gradient

descent of the cost function can be expressed as:

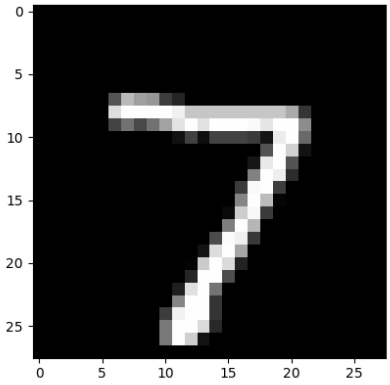
$$\nabla C = \begin{bmatrix} \frac{\partial C_1}{\partial w_{jk}^L} + ... + \frac{\partial C_j}{\partial w_{jk}^L} \\ \frac{\partial C_1}{\partial b_j^L} + ... + \frac{\partial C_j}{\partial b_j^L} \\ \frac{\partial C_1}{\partial w_{jk}^{L-1}} + ... + \frac{\partial C_j}{\partial w_{jk}^{L-1}} \\ \frac{\partial C_1}{\partial b_{jk}^{L-1}} + ... + \frac{\partial C_j}{\partial b_{jk}^{L-1}} \\ \vdots \\ \vdots \end{bmatrix}$$

Each of the values in the gradient details how the next iteration of the ANN should change its weights and biases. For example, w_{jk}^L should change by the value of $\frac{\partial C_1}{\partial w_{jk}^L} + ... + \frac{\partial C_j}{\partial w_{jk}^L}$.

4 Implementation

Though the primary goal of this project was to develop a conceptual understanding of gradient descent and backpropagation, our team was also excited to look in more detail about the software implementation of training a neural network for image classification. Since this was not a priority for us to develop from scratch, we downloaded existing Python code from the free online book Neural Networks and Deep Learning. This program used a three layer neural network to classify the MNIST handwritten digit data set.

The ANN has 784 neurons in the input layer (for a 28x28 pixel image), 30 neurons in the hidden layer, and 10 neurons in the output layer (for a classification from 0-9). With these components alone and 5000 images to train from, the system can autonomously transition from randomly assigned weights and biases to a carefully tuned network with over 95 percent correct classification.



(a) MNIST Handwritten Digit

0	0.0006894
1	1.07414284e-05
2	9.92845911e-05
3	0.02830999
4	4.44267752e-05
5	2.98902237e-05
6	1.95348710e-06
7	0.99599903
8	0.00036898
9	0.00252773

(b) ANN Output Layer

Figure 9: Image Classifier

Figure 9 shows a sample image from the MNIST dataset and how the ANN represented it via the activations of output layer neurons. Note that the activation for 7 is extremely high in comparison to the other digits; the system classified this number correctly!

5 Video

We also made a video about this topic here: https://youtu.be/TzRnwc2RHgY

6 References

https://brilliant.org/wiki/backpropagation/
https://machinelearnings.co/text-classification-using-neural-networks-f5cd7b8765c6
https://hackernoon.com/overview-of-artificial-neural-networks-and-its-applications-2525c1addff7