Fundamentals of Neural Networks: for Undergraduate Students

Corey Warren II

California State University San Bernardino
Professor Haiyan Qiao

Abstract. Typically, when first encountering neural networks as a student, the task of actually understanding the terminology is very daunting. First, there is the programming language barrier encountered when using C++ or Java. Using Python helps with this. Second, there is the mathematical language barrier when encountering phrases like "perceptron," "propagation," "sigmoid functions," "synapses," "weights," "weighted sums," "hidden layers," "activation functions. . . " the list goes on and on when it comes to neural networks. There are so many terms which are hardly glossed over in prerequisite classes, but are suddenly massively integral to even begin to understand how a neural network operates. In this independent study, I intend to tackle these fundamental problems through a deep understanding of each component, all while using minimal Python libraries combined with a simple programming style to make the prominent parts of a neural network incredibly simple to understand. Rather than obscuring the concepts with complex jargon, and advanced mathematical concepts meant only for post-graduates, I would like to break down the essential components of a neural network and explain what each part does, and how it relates to the network as a whole, and its goals.

Keywords: Neural Networks · Computer Science · Terminology · Beginner · Introduction · Independent Study.

1 Introduction

Neural networks are unique in that the way they learn is autonomous. Their initial conditions are randomized at first, but through certain methods, neural networks become more efficient. Another way of phrasing this is that neural networks rely on lots of input data and certain feedback loops in order to improve their accuracy over time. As we progress through this report, the actual terms for these processes will be named and explained.[4]

2 Basic Concepts

2.1 The Layers

The three layers of a neural network are: 1) input layer, 2) hidden layer, and 3) output layer. The input layer is where the network is fed-in information from

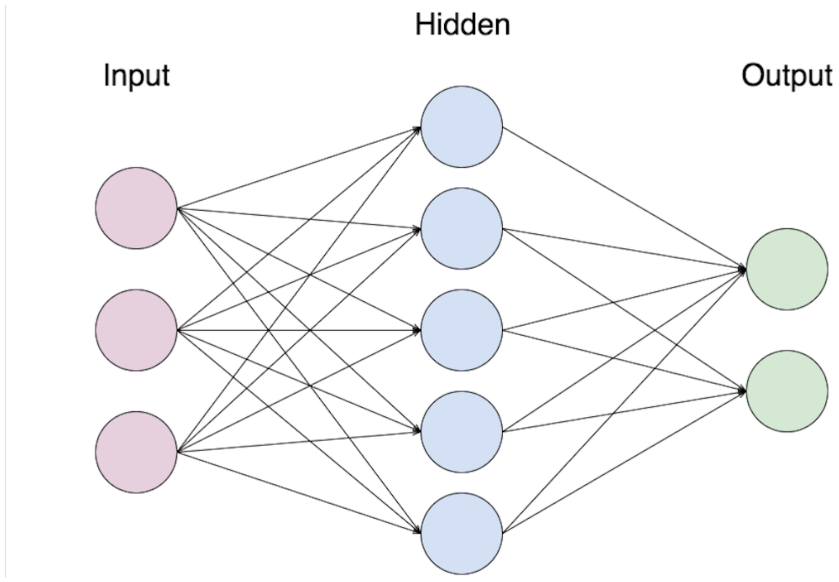


Fig. 1. These are the layers of a basic neural network. [10]

some outside source. The hidden layer is exactly as it sounds. To an end user, the hidden layer is completely hidden. But, the hidden layer is where all the work happens in a neural network. Finally, the output layer is where your neural network outputs its data to some other part of your program. Further operations can take place after output such as data cleansing, but these are typically outside the realm of the neural network itself. Although input and output layers can only be one layer of neurons, the hidden layer can consist of many layers of neurons which consist of staggered sub-layers that work differently than if they were all in one consolidated layer. In essence, a hidden layer's neurons are able to connect to each other in certain situations, because they are further separated into their own sub-layers. Only neurons in different layers or sub-layers connect to one another. [4]

2.2 Neurons

Neurons within the framework of a neural network are essentially points where activation functions come into play within the network. The three primary layers of a neural network are made up of some number of neurons. The number of neurons that make up any layer of the network will depend on the needs of the network.

2.3 Activation Function

Not to be overthought by those new to neural networks, an activation function is simply a function within the neuron that takes into account the activation *values* of all previous neurons, multiplied by their weight with respect to the current neuron. The activation function will sum up all of these weight/activation value products and modify them somehow. For those new to neural networks, activation functions should simply be seen as a function to ”plug into” their network. Refer to later sections in the report to learn what kinds of activation functions can be plugged in to a given neural network. Different activation functions serve different needs, but until those needs are ascertained, any activation function will do.

2.4 Synapses and Weights

Synapses are what connect neurons to one another. A weight is essentially a value unique to a single neuron, with respect to any given *previous* neuron. All previous neurons connect to your current neuron through a synapse. Each synapse connects to only one other neuron. Each synapse will only have one weight to it for your neuron, but any neuron will be connected to every previous neuron in the previous layer. The exclusion of connections between neurons is what defines layers and sub-layers. Neurons in the same layer will not connect to one another through synapses. The exception to this rule applies to when the hidden layer is made up of sub-layers, in which case sub-layers will connect to one another within the hidden layer.

2.5 Bias

Bias is some flat value within your activation function which is added to the sum of the products of activation values and synapse weights for a given neuron. In essence, bias is the natural likelihood, regardless of whatever neurons are connected to this neuron, that this neuron will be activated.

2.6 Training the Network

A neuron network will train itself either with labelled training data, and making errors and correcting them, or by being rewarded and punished for different behaviors. Learning from labelled training data is known as supervised learning. Meanwhile, learning from rewards and punishment incentives is known as unsupervised learning. Supervised learning intends to recognize classification or regression patterns within data, whereas unsupervised learning seeks to discover hidden patterns within data without the need for human intervention. One example of supervised learning would be the differentiation between cat and dog images by a neural network. Those images are first labelled by a human and then the network is trained on these labelled images. On the other hand, an example of unsupervised learning would be the implementation of a neural network in order to recommend videos or products to a customer using a website, with the hopes that the client is likely to watch those videos or purchase those products.[5]

3 Intermediate Concepts

3.1 Sigmoid Activation Function

The sigmoid activation function encapsulates the sigma sum of all the products of the weights and activation values, plus bias. Once this whole sum is calculated, it is input into the sigmoid activation function. Sigmoid will limit the final value between 0 and 1. Because these products will never reach infinity or negative infinity, an actual result of 0 or 1 will never happen, but the value will get close to 0 and 1 for large positive or negative values.

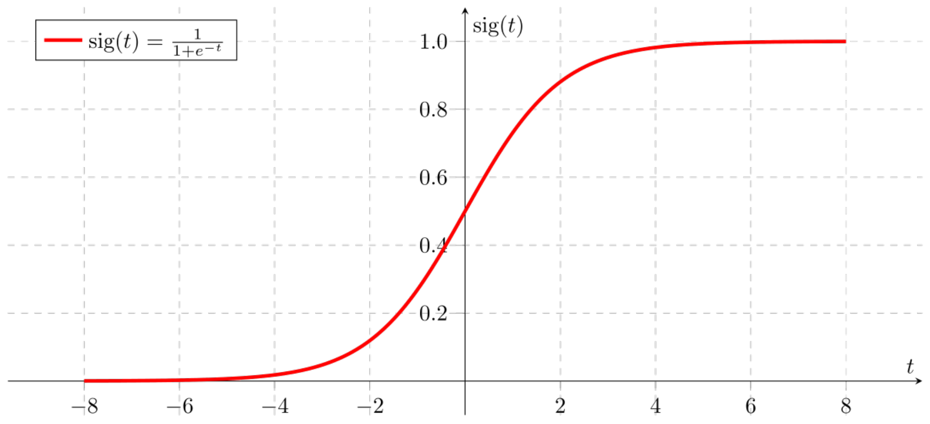


Fig. 2. The equation and graph for a sigmoid activation function. Note that its values are limited between 0 and 1 for any value of x. [11]

$$\sum_{i=1}^{m} w_i x_i + bias = w_1 x_1 + w_2 x_2 + w_3 x_3 + bias$$

Fig. 3. Here is the general equation for any neuron activation, before encapsulating the result with an activation function such as sigmoid or ReLU.

3.2 ReLU Activation Function

The ReLU activation function is a more often used function in modern neural networks. Generally speaking, Sigmoid functions tend to train more slowly in

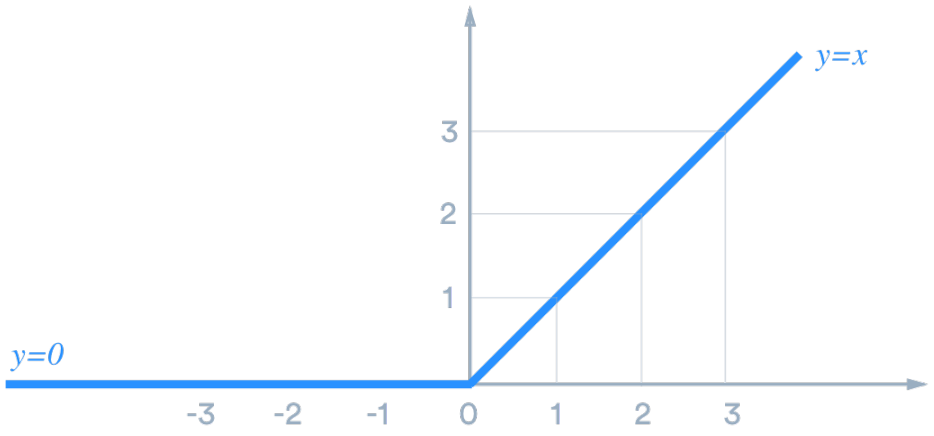


Fig. 4. A graphical representation of the ReLU activation function. [8]

the cases in which neural networks tend to be used today. ReLU activation functions train more quickly. It is worth re-emphasizing that as a beginner, one should simply choose an activation function and roll with it; just to get that first neural network code up and running.

3.3 Over-Fitting

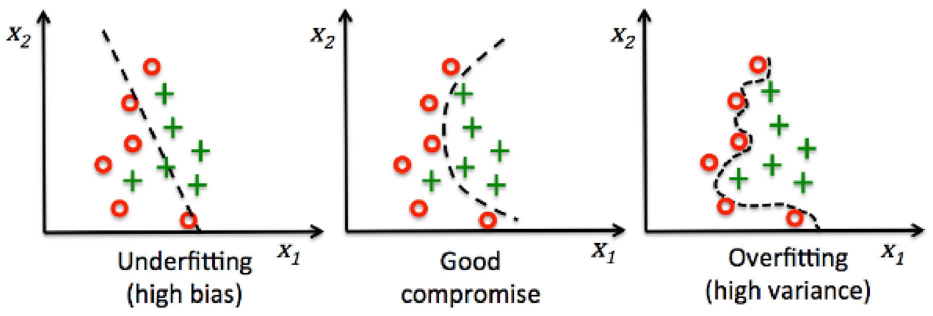


Fig. 5. These are examples of under-fitting, over-fitting, and a good balance between the two extremes.

Over-fitting is a negative attribute of a neural network after training in which the network is overly confident or precise with its predictions. Over-fitting tends to occur when a network is trained too much on a limited set of training data. This can lead to incorrect or inaccurate results.

One example of such a situation would be a neural network classifying images of cats versus images of dogs. An over-fit network in this case might mistakenly classify an image of a dog as an image of a cat due to a black pixel in some arbitrary position. Reason being is because this neural network has been over-fit to its training data, believing that small, irrelevant bits and pieces of data are what it should look for when outputting its predictions. This is not a desired behavior. Generally, neural networks are supposed to look at larger shapes and patterns, just as humans are. This is typically where the true usefulness of a neural network resides. But, when a neural network is over-fit, and it sees an arbitrary bit of data as a reason to classify an image or object, the potential intelligence of a neural network is wasted.

3.4 Under-Fitting

Under-fitting is the opposite of over-fitting, and occurs under a totally different set of conditions. Under-fitting generally means that your network has a high bias, but low variance. All this means is that your network is not sophisticated enough to understand the problem and predict its outcomes properly. This can be due to a number of reasons.

In some cases, your neural network may be under-trained and may simply require more training. In other cases, the neural network has too few neurons; this would lead to generalizations in the predictions from the neural network that are far too broad to be of any use practically. To provide a counter-example to the dog-cat classification example: imagine that all the neural network model can see is two very blurry blobs of pixels. You then ask that model to take its best guess as to if blurry blob X is a cat or a dog. The model simply does not have the processing power to ascertain that answer with accuracy, no matter how long the model is trained for.

3.5 Regularization

Regularization refers to a way of reducing over-fitting by re-introducing generalization. Since over-fitting occurs due to the over-training, and over-fitting harms our generalization [2], regularization finds a way to re-introduce that generalization so that variance is low enough to allow the network to generalize its pattern recognition enough to be accurate again.

Different types of regularization include: L1 or L2 method, augmenting the data, stopping training early before over-fitting occurs, or dropping out neurons from neural networks that are already rather large (refer to Fig. 6).

4 Advanced Concepts

4.1 Regression

While not a very advanced concept itself, it does sound rather advanced and can confuse new programmers. Regression is simply another type of analysis that neural networks can perform. Another type of analysis that has already been

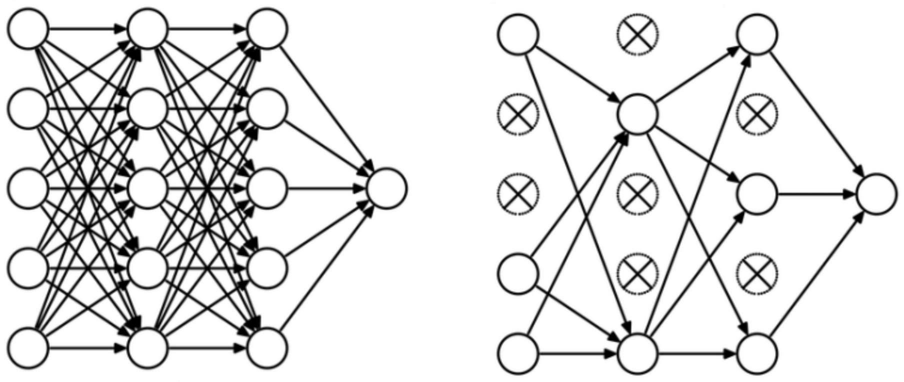


Fig. 6. Dropout Method applied to a neural network. This shows some of the nodes in a neural network being removed in order to reduce variance and improve generalization within the network. Note that a real neural network needing regularization would likely have several magnitudes more neurons than shown in this figure.

discussed is classification. Regression differs from classification in that regression, rather than finding the differences between two or more classes, seeks to find the mathematical relationship between variables. For example, the goal of some regression model might be to find the relationship between the value of a house and many other variables, such as number of bedrooms, number of bathrooms, size in square feet, number of stories, and selling prices of other homes nearby [12]. Given sufficient training data, a well-trained neural network may be able to predict the price of a house given just these points of data, just as a human home evaluator might.

It is worth noting that the terms ”logistic regression” versus ”linear regression” can be the cause of much confusion for those familiar with only one of these terms. Though they share very similar names, they entail totally different processes which are worth explaining here.

To elaborate, while linear regression uses regression and is not used for classification, logistic regression, on the other hand, **is** used for classification. This strange nomenclature can be the source of much frustration for beginners to this field. Still, both linear and logistic regression are forms of supervised learning. One more confusing aspect of these terms is that linear regression can create more than just a straight line of predictions (”linear” is misleading). This occurs when the data has multiple independent variables [13].

4.2 Loss Functions

Understanding what a loss function is key to grasping neural networks as a whole. Loss as a whole is a simple concept to understand, but without knowing how it relates to neural networks, one’s understanding of a neural network is fundamentally limited.

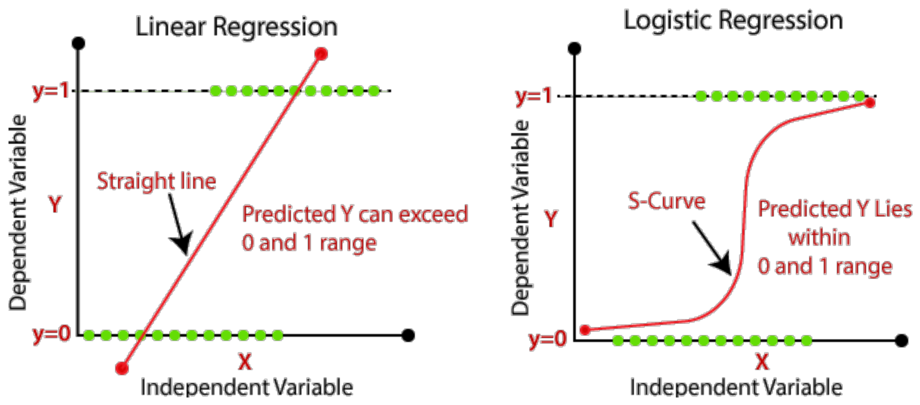


Fig. 7. A graphic depicting some differences between logistic and linear regression. [6]

Loss functions are some way of handling the error values output after data is propagated forward through the network from input, to the hidden layer, to the output layer. Error is the difference between the prediction that the neural network made at the point of output, minus the true, expected value. Loss, however, takes that error and applies a function to it, modifying it in some way in order to make it usable in other aspects of the neural network, particularly those pertaining to back-propagation (covered later).

An example of a type of loss function is Mean Squared Error (MSE). This takes the error and squares it, giving us a loss value. This loss value can later be used in back-propagation, but usually after being factored in to the cost function if the network has multiple training sets [7].

4.3 Cost Functions

Cost functions are very similar to loss functions. The main difference between the two is that a cost function applies to a number of training sets for the neural network, rather than just one. Another way of phrasing this is that a cost function is a function that takes loss functions in as input [7].

4.4 Forward-Propagation

Forward propagation is standard procedure that anyone using a neural network can understand. All forward propagation is is the movement of data from the beginning to the end of the neural network. Data that is forward-propagated through a neural network simply goes from input to hidden layer to output. At the end of forward propagation, the error is calculated between the predicted result and the expected/true result [3].

4.5 Back-Propagation

While forward propagation is the most obvious flow of data in a neural network, back-propagation is the more complex, defining feature of a neural network, which sets it apart from other models. Back-propagation refers to the self-correcting nature of a neural network. One short version of this word that you may encounter is "backprop." What back-propagation allows a neural network to do is allow data to flow backward through the network in order to find the gradient. Finding the gradient allows the network to find an optimal way to reduce the loss or error of the current state of the network by adjusting weights and biases of some or all of the neurons within the network. All this jargon means is that the network is able to learn based off of the results of back-propagation.

One key distinction to note is that while back-propagation is important, it is not precisely what makes up the entire learning algorithm for neural networks. Rather, back-propagation is responsible for computing the gradient, which then allows some other algorithm to improve the accuracy of the network by using that newfound gradient value for gradient descent [3].

4.6 Gradient Descent

Using the gradient, which is essentially just slope, gradient descent will allow the neural network to efficiently reduce its cost function by adjusting some or all of the weights and biases of its neurons.

Gradient descent in and of itself is not incredibly hard to understand, so long as all the pieces which it involves are understood at a basic level. All gradient descent is meant to do is to take the gradient (calculated by back-propagation), make it negative, and use that vector to find the direction of steepest descent down the multi-dimensional space of the cost function. In order to understand this, it may help to simplify the cost function down to 2 dimensions, w, and C(w); wherein w is a single variable representing all weights and biases in the neural network, and C(w) represents the cost function for said variable. One can plot this relationship by plotting w on the horizontal axis and C(w) on the vertical axis (See: Fig. 8).

In the example depicted in Fig 8., we seek to reach a local minimum with regard to C(w). To find this minimum, we adjust our w (weights and biases) in one direction or another. We calculate precisely which way to adjust this w value by using the gradient value we already calculated during back-propagation. Through many small adjustments to w using gradient descent, we get closer and closer to a local minimum of the cost function. This is how cost is reduced.

Cost is specifically reduced after all weights and biases are multiplied by the negative gradient. After this multiplication occurs, these new weights and biases are simply added back to the original values of the weights and biases of the network before they were multiplied by the negative gradient. The new sums of these values will serve as the next set of weights and values for the next time we forward-propagate through the neural network.

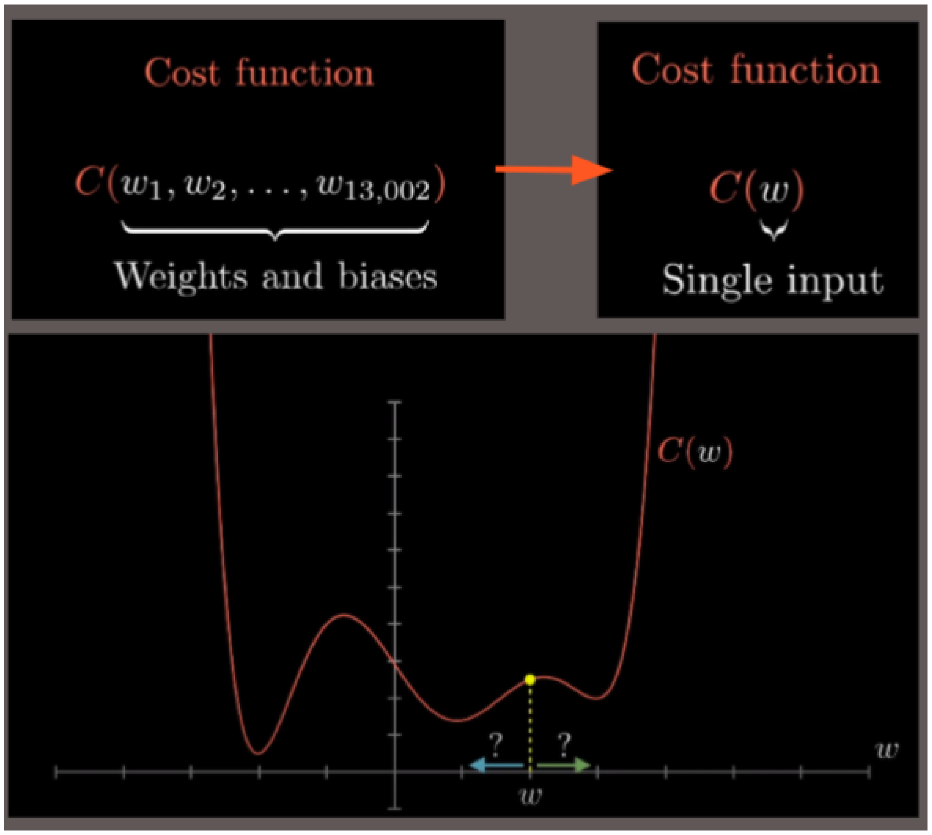

Cost function
$C(\underbrace{w_1, w_2, \ldots, w_{13,002}}_{\text{Weights and biases}})$
Cost function
$C(\underbrace{w}_{\text{Single input}})$
$C(w)$
?
?
w
w

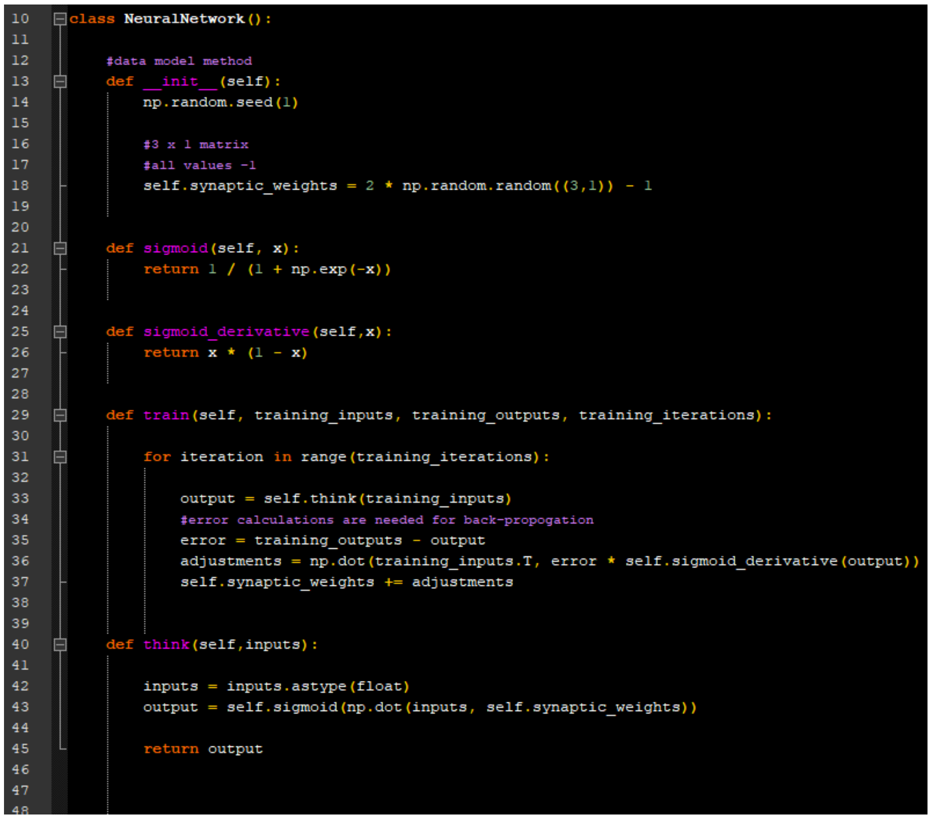
Fig. 8. A graphic provided by [1], which helps to understand the relationship between cost functions, weights and biases, and gradient descent.

$$w_{i+1} = w_i - \nabla C(w_i)$$

Fig. 9. A simple formula for calculating our current weights and biases versus our next forward-propagation's weights and biases. The next iteration shall have new weights and biases derived from the current weights and biases minus the gradient times the result(s) of our cost function.

5 Experiment

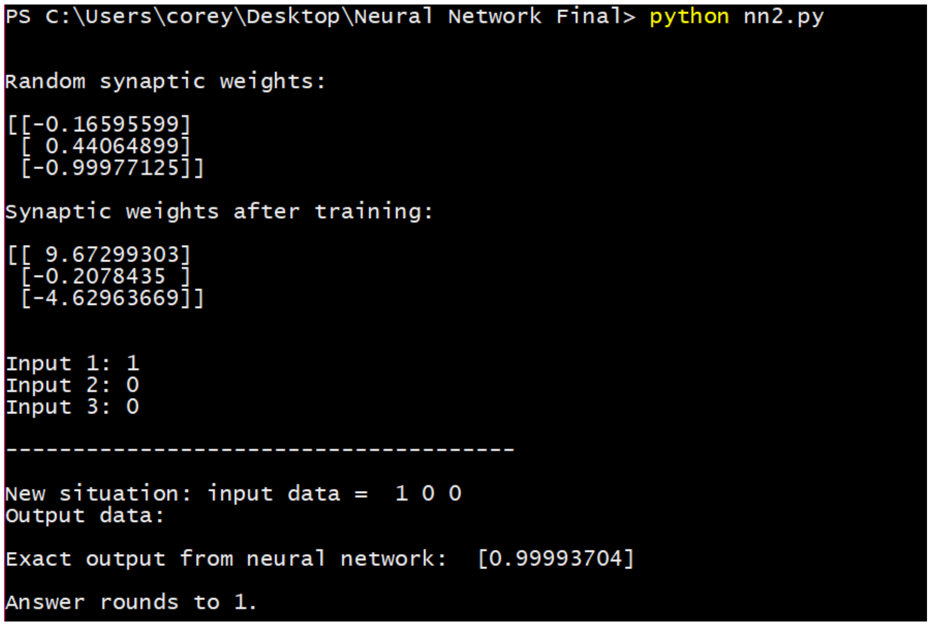
In the proceeding figure is the class definition for NeuralNetwork(). This defined the network's seed value for random(), the sigmoid functions used as the activation functions for my neurons, the training function, and the output/forward-propagation function aptly named "think."



```
class NeuralNetwork():

    #data model method
    def __init__(self):
        np.random.seed(1)

        #3 x 1 matrix
        #all values -1
        self.synaptic_weights = 2 * np.random.random((3,1)) - 1


    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))


    def sigmoid_derivative(self,x):
        return x * (1 - x)


    def train(self, training_inputs, training_outputs, training_iterations):

        for iteration in range(training_iterations):

            output = self.think(training_inputs)
            #error calculations are needed for back-propogation
            error = training_outputs - output
            adjustments = np.dot(training_inputs.T, error * self.sigmoid_derivative(output))
            self.synaptic_weights += adjustments


    def think(self,inputs):

        inputs = inputs.astype(float)
        output = self.sigmoid(np.dot(inputs, self.synaptic_weights))

        return output
```

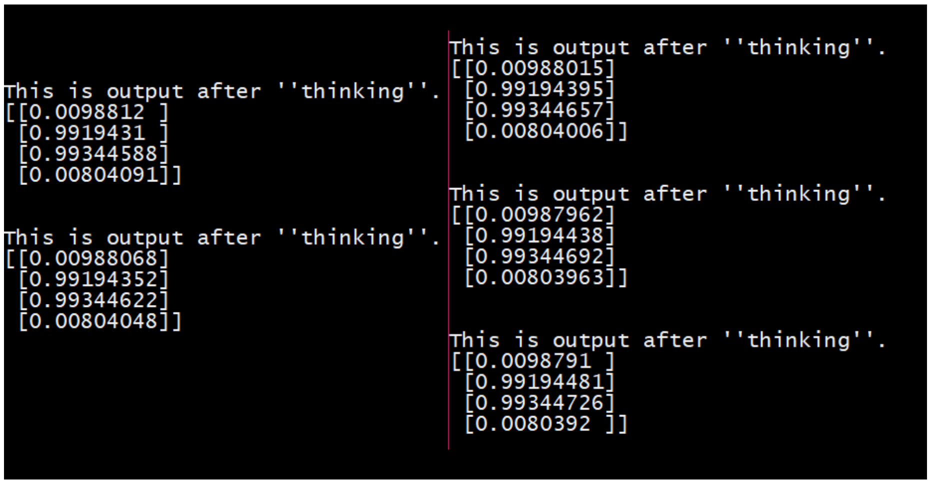
Fig. 10. Class definition section of my Python code, with help from [9].

The program is rather simple. It analyzes 4 predefined number sets, each consisting of 3 binary numbers which can have either value 0 or 1. Each time it "thinks" (or forward-propagates), it calculates the error of its predictions versus the expected value. The true/expected output is very simple, as it is only either a 0 or a 1. Finally, some basic gradient descent does take place, which is represented by taking the dot product of the training inputs transposed, times the product of error this run times the derivative of the sigmoid function with respect to output. Repeating this process allows the network to train itself to become more and more correct, minimizing the cost function as it progresses.



```
PS C:\Users\corey\Desktop\Neural Network Final> python nn2.py


Random synaptic weights:

[[-0.16595599]
 [ 0.44064899]
 [-0.99977125]]

Synaptic weights after training:

[[ 9.67299303]
 [-0.2078435 ]
 [-4.62963669]]


Input 1: 1
Input 2: 0
Input 3: 0

--------------------------------------

New situation: input data =  1 0 0
Output data:

Exact output from neural network:  [0.99993704]

Answer rounds to 1.
```

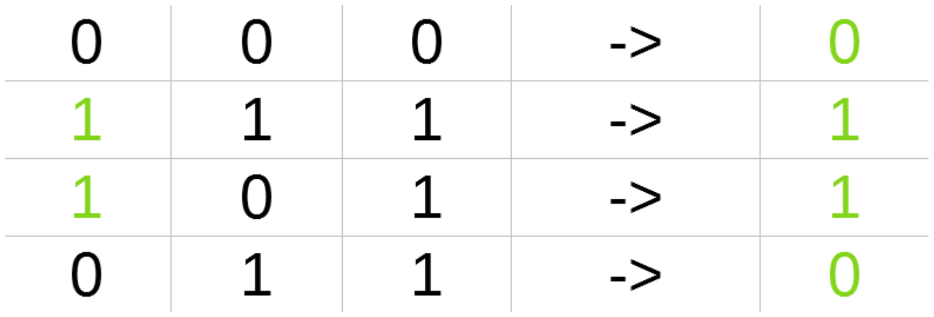
Fig. 11. Output of my Python code. The output was originally '0.9999,' but clearly this answer rounds to 1. Recall that the sigmoid function will never be able to output a nice '1' value on its own, as that would imply that x has reached infinity. That is why the result is rounded up to 1 afterward, but not in the raw output.



```
This is output after ''thinking''.
[[0.0098812 ]
 [0.9919431 ]
 [0.99344588]
 [0.00804091]]


This is output after ''thinking''.
[[0.00988068]
 [0.99194352]
 [0.99344622]
 [0.00804048]]
```

```
This is output after ''thinking''.
[[0.00988015]
 [0.99194395]
 [0.99344657]
 [0.00804006]]


This is output after ''thinking''.
[[0.00987962]
 [0.99194438]
 [0.99344692]
 [0.00803963]]


This is output after ''thinking''.
[[0.0098791 ]
 [0.99194481]
 [0.99344726]
 [0.0080392 ]]
```

Fig. 12. Depicted here is the process of my neural network adjusting its outputs as it learns. Its outputs were initially random, as the weights and biases it had were also initially random. The image reads starting from the left, from top to bottom first. Its results will converge to approx. [0, 1, 1, 0].



0	0	0	->	0
1	1	1	->	1
1	0	1	->	1
0	1	1	->	0

Fig. 13. Graphic representing the training data that my basic neural network used. All the network had to learn was that the binary value in column 1 is equal to 1 if and only if the output is also 1. The same applies for the 0 value.

6 Conclusion

While this program, designed with the help of [9], is rather simple and practically useless, it did highlight a key aspect of neural network programming that fellow computer science students around me have struggled with: while one can easily find code somewhere online and copy its entire structure or even every line of code; this does not guarantee that the student has learned anything.

This was also true for my experience with this Python code. While referencing someone else's code and accompanying video tutorial did help me understand how simple a perceptron neural network could be, and refreshed my memory on terms such as "training set," "weights," and "sigmoid activation function;" it did not grant me the knowledge of the elementary basics of a neural network that would help me understand how to recreate such a program on my own without additional aid.

It is with this understanding that I set forth to choose this topic as the subject of my independent studies this semester under Professor Qiao. By learning these fundamental concepts one-by-one, perusing several student-oriented research papers, and even just browsing some reputable websites, I was much more able to grasp the concepts that tend to beguile unprepared computer science students.

I realize my tone throughout this report may be somewhat informal, but that is simply due to my intended audience. My goal from the beginning was to explain these basic building blocks of neural networks to students who would be in the position I was a year ago. Those who would be encountering this confusing yet alluring field of computer science for the first time. While yes, neural networks are more complicated than one might expect, that does not mean that understanding them at a basic level is necessarily difficult. It just requires a reasonable path to understanding, starting from the basic concepts and progressing gradually to the advanced.

Part of my own understanding came from writing this very research paper and performing my senior presentation on neural networks. I would highly recommend this path to any other student who has an interest in neural networks (or any other complex subject, for that matter) but is still confused about the sizeable vocabulary set that comes with the subject.

References

1. 3Blue1Brown: Gradient descent, how neural networks learn — deep learning, chapter 2, https://www.youtube.com/watch?v=IHZwWFHWa-w
2. CapableMachine: Regularization in neural networks https://capablemachine.com/2020/08/20/regularization-in-neural-networks/
3. Goodfellow, Ian; Bengio, Y.C.A.: Deep Learning. MIT Press (2016), https://www.deeplearningbook.org/contents/mlp.html#pf25
4. IBM: Neural networks https://www.ibm.com/cloud/learn/neural-networks
5. IBM: Supervised vs. unsupervised learning: What's the difference? https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning
6. javaTpoint: Linear regression vs logistic regression, https://www.javatpoint.com/linear-regression-vs-logistic-regression-in-machine-learning
7. Medium: Cost, activation, loss function, neural network, deep learning. what are these?, https://medium.com/@zeeshanmulla/cost-activation-loss-function-neural-network-deep-learning-what-are-these-91167825a4de
8. Medium: A practical guide to relu, https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7
9. Polycode: Create a simple neural network in python from scratch, https://www.youtube.com/watch?v=kft1AJ9WVDk
10. towardsdatascience: Classical neural network: What really are nodes and layers?, https://towardsdatascience.com/classical-neural-network-what-really-are-nodes-and-layers-ec51c6122e09
11. towardsdatascience: Derivative of the sigmoid function, https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e
12. TuringBot: A regression model example and how to generate it https://turingbotsoftware.com/blog/regression-model-example/
13. Vadapalli, P.: Linear regression vs. logistic regression: Difference between linear regression & logistic regression https://turingbotsoftware.com/blog/regression-model-example/