Project Based Learning in Data Science

With Applications of Neural Networks

Mid-Term Report

**Group Members**

**2/c Cody Bellamy \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[Cody.R.Bellamy@uscga.edu](mailto:Cody.R.Bellamy@uscga.edu)

**2/c Justin Steiner \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[Justin.C.Steiner@uscga.edu](mailto:Justin.C.Steiner@uscga.edu)

Table of Contents

[1 Introduction to Neural Networks 3](#_Toc33720023)

[1.1 Inputs, Outputs, and Neurons 3](#_Toc33720024)

[1.2 Activation Functions 4](#_Toc33720025)

[1.2.1 Linear 4](#_Toc33720026)

[1.2.2 Sigmoid 5](#_Toc33720027)

[1.2.3 Rectified Linear Unit (ReLU) 6](#_Toc33720028)

[1.3 Feed Forward 7](#_Toc33720029)

[1.4 Back Propagation 7](#_Toc33720030)

# Introduction to Neural Networks

When considering a neural network for applications in data science, it is important to understand the basic structure, process, and common algorithms that are used in application. There are two primary branches of study revolving around neural networks. The first branch is artificial intelligence. Artificial intelligence is the study of decision making. The goal of artificial intelligence is to design a neural network that considers many factors and simulates natural intelligence to solve a problem. The second branch of study is machine learning. Machine learning is the process of analyzing data and making decisions based on previous data. The goal of machine learning is to reach an optimal solution without regard for the process and/or emulating intelligence. A machine learning algorithm only cares for a solution that is optimal and will “learn” to disregard any non-optimal solutions. In this directed study, we have placed an emphasis on machine learning and related sub-branches such as deep learning. This introduction will provide a basis into the basic structure and functionality of a deep-learning network.

## Inputs, Outputs, and Neurons

As the name implies, a neural network is designed with the human brain as inspiration. Each node in the network is called a neuron. Each neuron in the network contains a series of inputs and outputs like the synapsis in a brain.

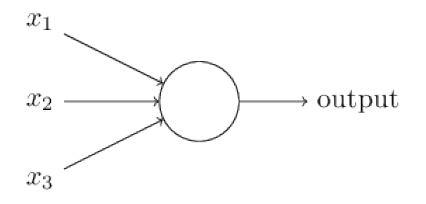


Fig. 1.1 – A neuron with 3 inputs and 1 output

In figure 1.1, we can see a neuron which as 3 inputs and one output. We can represent the inputs and outputs as the vectors and y respectively. It is important to note that each neuron contains a variable number of inputs and only one output. Neurons contain activation function that determines its output based on its inputs. Each neuron in the network calculates its output based on a predetermined function called an “activation function.” Using this basic structure, we can create a neural network with multiple neurons as shown in figure 1.2.

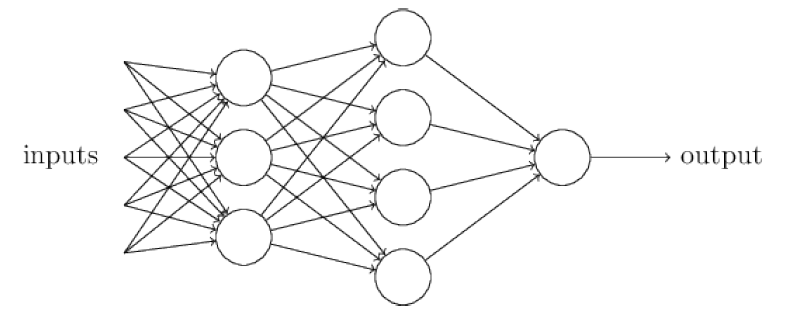


Fig. 1.2 – A deep learning neural network with 5 inputs, 2 hidden layers[[1]](#footnote-1), and 1 output

## Activation Functions

The learning behavior of a neural network is dependent on many factors. But, one of the largest factors in the ability of a neural network to learn is the activation function. Depending on what results are desired, a different activation function will be chosen. Three activation functions that we will be covering are Linear, Sigmoid, and Rectified Linear Unit (ReLU).

### Linear

The linear function is a simple representation of a linear function. Neurons with the linear activation function are given the name, “perceptron.” To understand this function, we will add some features to our understanding of a neural network. First, each input will be given a weight, w, that is specific to only that input. Second, each neuron will contain an overall bias, b. This activation function is powerful in small networks. For example, the weights and biases could be hard-coded in such a way to create logic gates.

Function 1.1 – Linear activation function

Although linear models would appear to be the obvious choice when attempting to model machine learning, this is not always the case. As we can see by function 1.1, our output is a binary response as a function of the inputs. Put in simpler terms, a linear activation function would allow inputs of any value but could only output a 1 or 0 as shown in figure 1.3. This useful in applications where binary decisions are required. But, this model comes with its flaws. Since the response is binary, y could be equal to .000001. Which, to a human would be almost blatantly obvious that the value should be 0. In a linear model, this value is changed to 1. That is a lot of information that is lost in the network that could be passed on to the next neuron as input. Another issue with linear models is the fact that a small change to the weights and biases could result in a drastic change within the network. Take the last example for instance. If the bias is adjusted such that Δb= -0.000001, the neuron would now flip its output completely changing the behavior of the network.

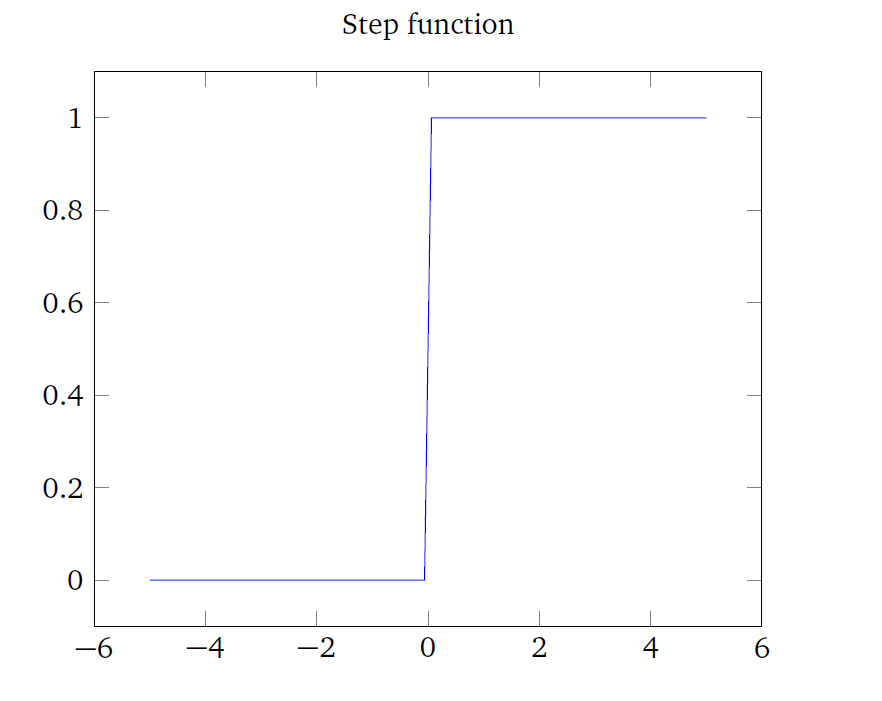


Fig 1.3 – The response of a linear activation function

### Sigmoid

The sigmoid function is a solution to the issues presented at the end of the previous section, 1.2.1. We would like to keep many of the same traits to allow for the capability of training our network. For example. Since the response of a linear network is a step function, we would like positive values of y to result in values closer to 1 and negative values of y to result in values closer to 0.

Function 1.2 – Sigmoid activation function

This equation is structured such that as z >> 0, y≈1 and as z << 0, y≈0. This behavior mimics the behavior of a step function with the added benefit of allowing a smoother transition between values. At values where z≈0, the value of y≈0.5 and the rate of change of y is greatest as shown in figure 1.4. This helps to discourage the values of y to be close to 0.5 and to have a higher probability of being closer to 0 or 1. Another important note is that the output is exactly the output of the sigmoid function. This behavior of neurons allows for more dynamic learning behavior.

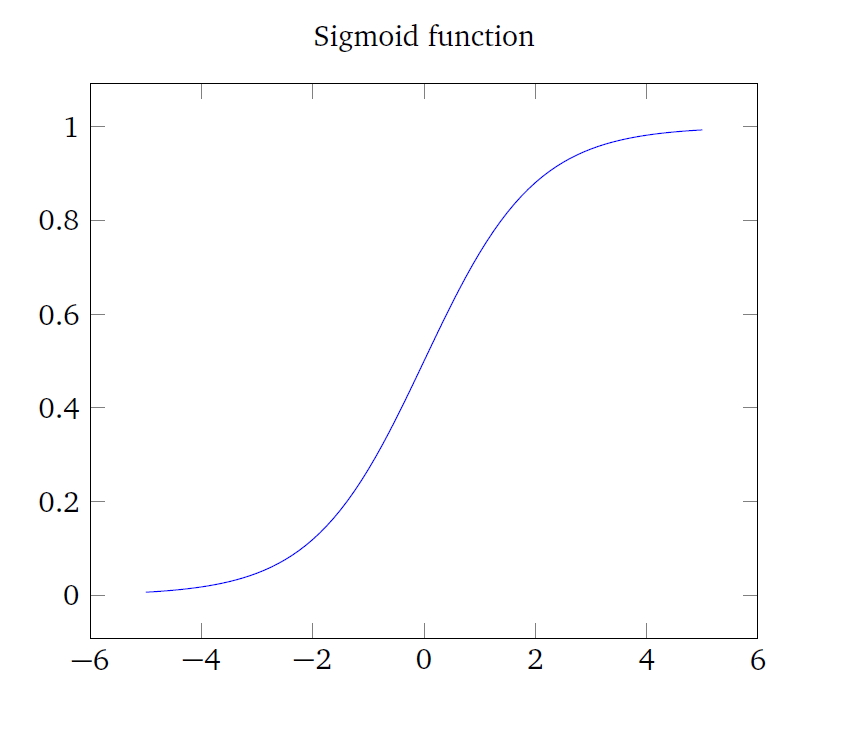


Fig 1.4 – The response of a sigmoid neuron

Like perceptron neurons, there are downsides to the sigmoid. Most notably, is the mathematical efficiency with respect to time. In the feed forward[[2]](#footnote-2) process, the outputs are calculated for each neuron independently. Calculating exponentials becomes a very expensive process for large networks. For smaller networks, this is not a big deal. But, the sigmoid function has an exponential in big-O.

### Rectified Linear Unit (ReLU)

The final activation function that we will explore will be the ReLU function. Although there are several derivations and deviations of the ReLU function, we will only explore one for the sake of simplicity.

Function 1.3 – ReLU activation function

As shown in function 1.3, the ReLU function almost brings us full circle back to the linear activation function. The major difference is that the output of the ReLU function has the range [0,y). The values are no longer 0, 1, or a range between the two. We now have the capability of reaching values of y that approach infinity. Although this might seem to break our concept of a neural network that we have been building, this has little to no negative repercussions. Recall that the values for our inputs can be any value and are simply a function of our weights and biases. At first, our neural network will start with random weights and biases and over time, these will adjust to our inputs to achieve desired results. With this in mind, we quickly realize that the outputs do not matter if they are inputs to another layer.

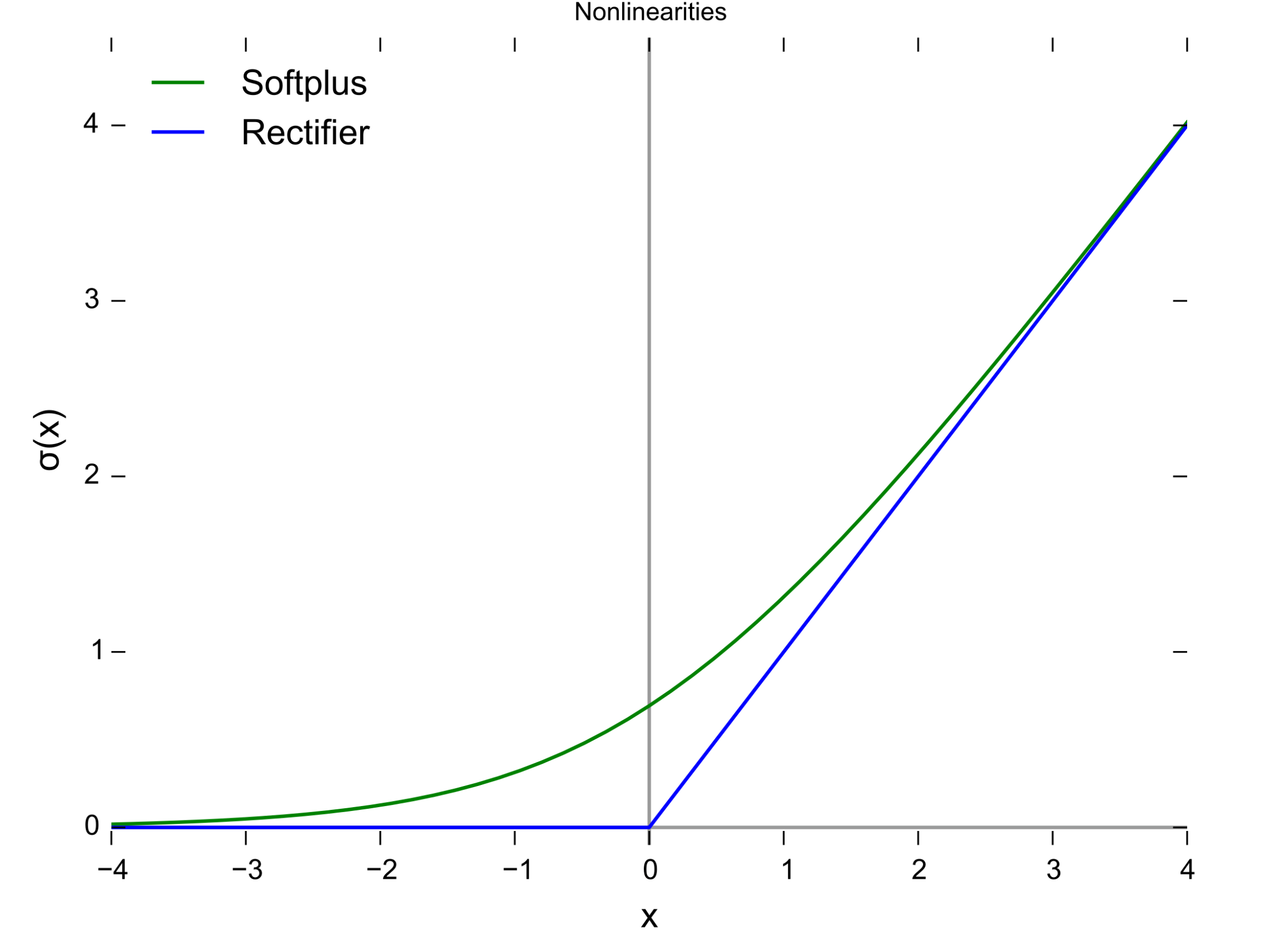


Fig 1.5 – ReLU activation function (blue) and SmoothReLU/Softplus activation function (green)

## Feed Forward

Almost all neural networks follow this model. Feed forward describes the process of accepting inputs, performing calculations to determine output, and feeding the output “forward” to the next layer. In a feed forward model, this allows us the freedom to design our neural network with any structure as we please so long as the overall network’s inputs and outputs satisfy our needs.

## Back Propagation

At this point, we’ve discussed the setup and layout of a simple neural network; but, what of the learning process? To achieve learning from a neural network, we need a way to adjust our weights and biases in a way that achieves beneficial results. This process is called back propagation. So called because the error in our network is calculated at each layer and propagated backwards. At each layer, we determine what the inputs and outputs were and in what direction to change our values in a direction towards our desired results.

### Calculating Error

The first step in back propagation is to determine what our error is. We will consider our final output, desired output, all weights, and all biases in our network. To consider all of these and quantify the results of our network, we need to define a cost function. For simplicity, we will discuss the mean squared error (MSE) function as our cost function. Since MSE is a strictly positive exponential function, we can quantify how well our network is doing by minimizing the result of MSE. Since our goal is to adjust our weights and biases, our MSE will be a function of weights and biases.

Function 1.4 – Mean squared error cost function

As shown in function 1.4, n is the number of training inputs (commonly referred to the batch size), y is the vector of outputs from our neural network, x is a vector of inputs to our neural network, a is the actual result from our training data, w is our weights, and b is our biases. Using this function, we are challenged with minimizing C(w,b). Our cost function can be visualized in figure 1.6.

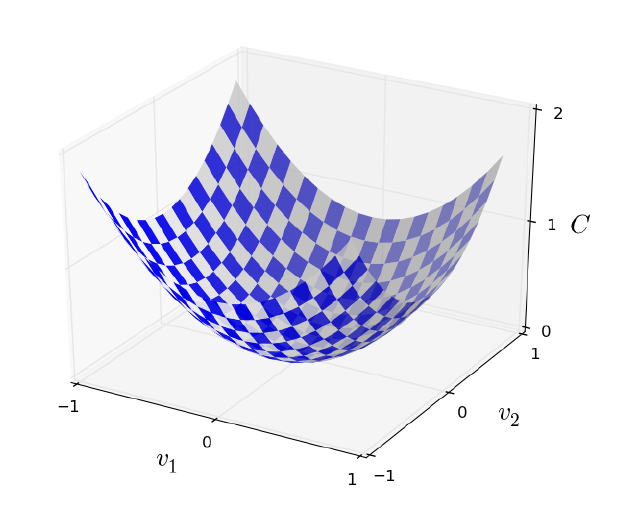


Fig 1.6 – An oversimplified visualization of our cost function plotted in 3 dimensions

### Steepest Ascent

To determine what direction to shift our weights and biases, let’s assume they are v1 and v2 respectively in figure 1.6. Using calculus, we can determine which direction is the steepest ascent of our cost. Although this may seem trivial, computer resources are not cheap. If we apply calculus to a very large neural network, calculating partial derivatives can become very pricy in terms of performance.

### Backpropagation Algorithm

To solve the issue of calculating partial derivatives, we will use a 4-step back propagation algorithm. Using 4 equations, we can determine the error at each layer and the gradient of our cost function in terms of the weights and biases for each layer. The process begins by feeding our batch forward and calculating our error, , at the final layer using our cost function. Next, we shift our output layer to minimize our cost function by multiplying a small value to our weights and biases by Hadamard product, .

Functions 1.5-8 – The backpropagation functions

1. Although figure 2.2 has multiple connections for outputs from many of the neurons in the hidden layers, the value that is output from the neuron is the same across all outputs for that neuron. [↑](#footnote-ref-1)
2. Process of accepting inputs, calculating the output, and feeding the result to the next layer. [↑](#footnote-ref-2)