**Connor Lambert**

**Neural Network to Recognize Handwritten Digits**

**CSC 492: Special Problems – Dr. Ali**

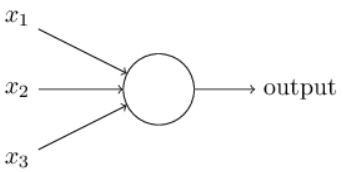
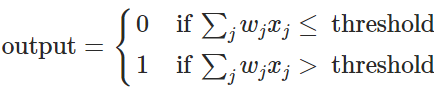
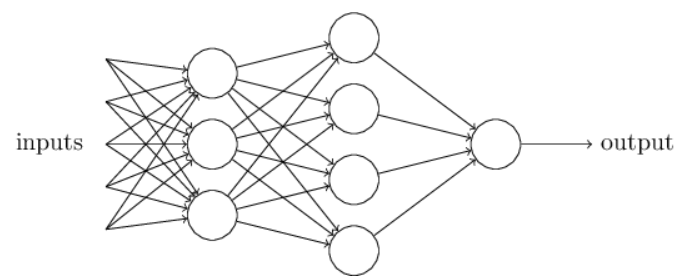
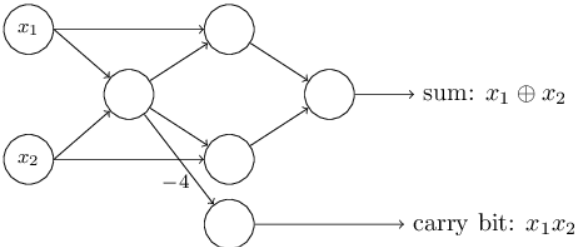
**Research Notes**

**Goal:** To create a neural network, that which can recognize handwritten digits. (Optical Character Recognition)

**Background of Problem:** Image recognition using our eyes is deceptively easy, as it has evolved across hundreds of millions of years into a system of visual cortices working in conjunction to perceive our surrounding environment. When we view a number written by a human, we can use visual patterns to almost instantaneously know what number we are looking at – “an 8 has two circles sitting on top of one another”. The concept of visual pattern recognition is easy when we do it but becomes extremely complicated when attempting to express in the form of an algorithm. Using precise rules to express this unconscious pattern recognition can result in becoming lost in a series of cases and exceptions.

**Solution:** Develop a system that which can take many handwritten digits (training data) and create rules to assist in digit recognition.

**Notes on Methods Etc.:**

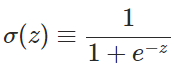
* **Perceptrons**
  + Developed by Frank Rosenblatt in the 50’s and 60’s.
  + Not commonly used in present day compared to Sigmoid Neurons
  + Perceptrons take several binary inputs and produce a single binary output:
  + 
  + Computation of the output is achieved by pairing each input with something called a *weight* – a real number that expresses the importance of the respective inputs to the output. Once paired, each input is multiplied by its weight and summed with the other products. If this summation is <= some threshold value, the Output is 0, otherwise if it is greater than this threshold value, the Output is 1.
  + Formula for Output: 
  + We can abstract our network of perceptrons into more layers, giving the network an ability to engage in more sophisticated decision-making
  + We can see from the above graph, layer 1 of perceptrons are making three “simple” decisions based off the weight of the given input. From that point, layer 2 perceptrons make decisions by weighing up the decisions from layer 1. The idea is, the latter layers can make decisions at more complex and abstract levels than perceptrons in previous layers.
  + **Simplifying the Formula**
    - The notation, **∑jwjxj > threshold**, is cumbersome and can be simplified.
    - First, we can rewrite “**∑jwjxj**” as a dot product, ***w* ∙ *x* ≡ ∑jwjxj**. *W* and *X* are vectors whose components are the weights and inputs respectively.
    - Second, we can move the **threshold** to the other side of the inequality, and to replace it with what is more popularly used, the perceptron’s ***bias, b* ≡ -threshold**. The bias is the measure of how easy it is for the perceptron to “fire”. If the bias is large, it’s extremely easy to receive an output of 1, the same is true for a small bias and an output of 0.
    - The final form of the new formula can be written as:
  + **Perceptrons can also be used to compute elementary logical functions (AND, OR, NAND)**
    - Perceptrons can be used as stated above, good examples located in text but too long for these notes.
    - Below is a network of perceptrons organized with weights and bias to function as a NAND gate for bitwise summation:

Starting from the left, we see the input layer of perceptrons, x1 and x2, both with weights of -2 and biases of 3. The middle layer is comprised of 1 perceptron, also with a weight of -2 and bias of 3. The middle layer is used as an input to the carry bit, originally as a double input w/ respective weights of -2 and biases of 3 but are, more logically, merged into a single input with a weight of -4 and bias of 6.

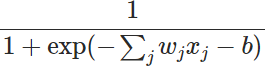
* + An issue with perceptrons is when adjusting their weights, they can have a drastic impact on the perceptrons’ outputs.
* **Sigmoid Neurons**
  + Ideally, we want our neural network to adjust its weights and biases on its own to correctly classify the target digit. We want to make a small change in the weights so that the network makes a small change in its decisions, moving the system closer to correct classification. Repeating this process over and over all the while reaching nearer the goal is the concept of learning within a neural network. Unfortunately, perceptrons are incapable of this, but in comes the Sigmoid Neuron. Unlike the perceptron, when making minor weight adjustments, they only have a minor effect on the output.
  + A sigmoid neuron can have many inputs, much like perceptrons, but instead of only taking the values 0 or 1, sigmoid neurons can take a value between 0 and 1 as well
  + Let’s Talk Formulas:
    - Sigmoid Neurons are comprised of an input value, weight, and bias, much like a perceptron, but the output is given as:

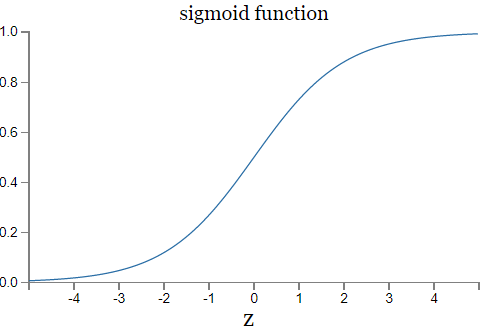
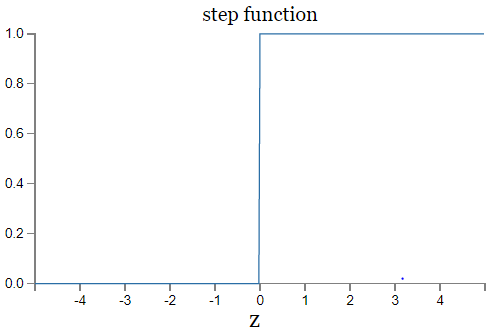
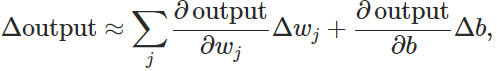


* Where σ is known as the **Sigmoid Function** (logistic function) and is as follow:



* A more expressive way to show the Sigmoid Function is defined as:



* Numbers are for nerds, I know, but this formula is amazing. It is the entire reason we can make Sigmoid Neurons learn. When the value of (w ∙ x + b) is extremely large and positive, it passes into the sigmoid function and the output for it becomes 1, as a perceptron would be. If the value of the previous statement is an “extremely negative” number, it passes into the sigmoid function giving it the value of 1. A little bit of knowledge on Euler’s number and how it works upon given inputs is needed to understand why the sigmoid function behaves as it does.
* We have seen what happens to the sigmoid function when subject to very large changes, but what about small, modest alterations? Well let’s begin by looking at the graphical representation of the Sigmoid Function: 
* We can see that the Sigmoid Function is a smoother version of a Step Function, which is actually the form the Perceptron Function takes: 
* The beauty of the smoothness in the graph of the Sigmoid Function is that unlike in the Step/Perceptron Function, small, deliberate changes made to the weights and biases yield small changes in the output, not just 0 or 1.
* We can calculate this small change in the output by the following function: 
* The weird squiggles on the right denote partial derivatives, on a numbers level, it’s okay to not completely understand, but it says that the change in output is a linear function based off the changes in the weight and bias.

**Architecture of Neural Networks:**

**Sources**

* Nielsen, Michael (2018). *Neural Networks and Deep Learning.* Retrieved from http://neuralnetworksanddeeplearning.com