**Connor Lambert**

**Neural Network to Recognize Handwritten Digits**

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

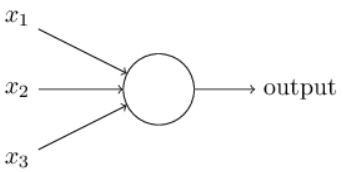
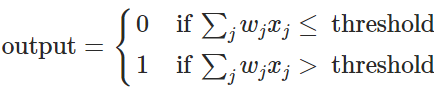
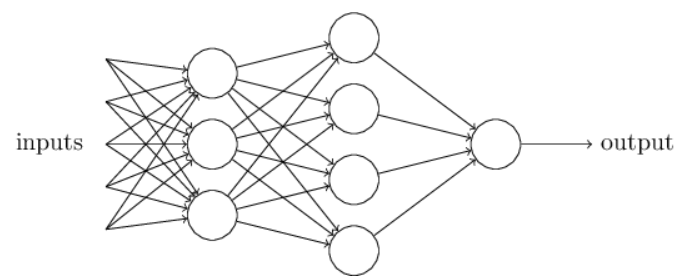
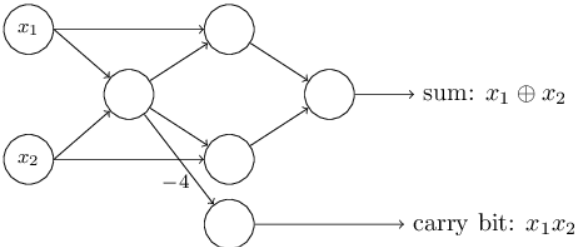
**Research Notes**

**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.:**

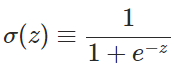
* **Perceptrons**
  + Developed by Frank Rosenblatt in the 50’s and 60’s.
  + Not commonly used in present day compared to Sigmoid Neurons
  + Perceptrons take several binary inputs and produce a single binary output:
  + 
  + Computation of the output is achieved by pairing each input with something 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.

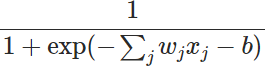
* + An issue with perceptrons is when adjusting their weights, they can have a drastic impact on the perceptrons’ outputs.
* **Sigmoid Neurons**
  + Ideally, we want our neural network to adjust its weights and biases on its own to correctly classify the target digit. We want to make a small change in the weights so that the network makes a small change in its calculations/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:

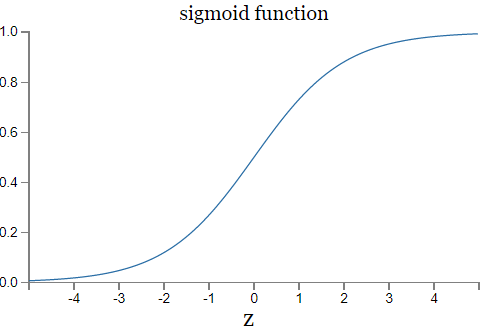
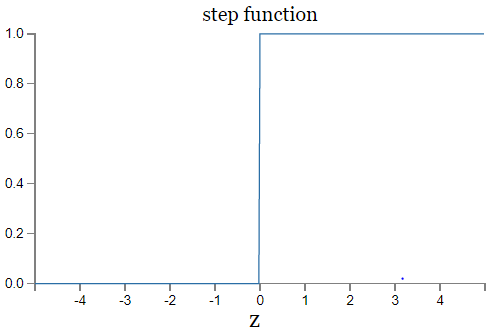
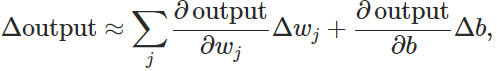


* Where σ is known as the **Sigmoid Function** (logistic function) and is as follow:

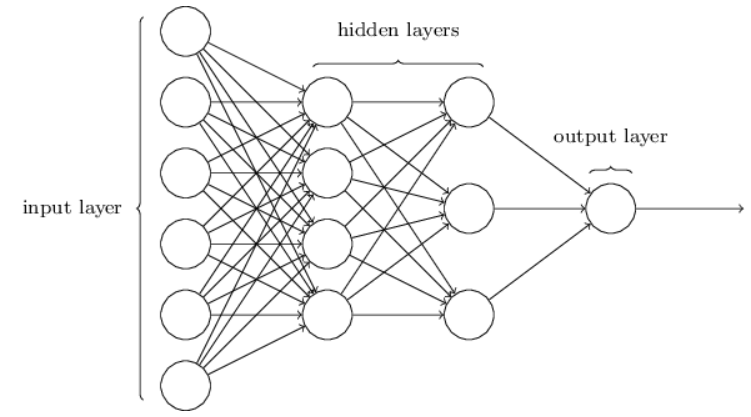


* A more expressive way to show the Sigmoid Function is defined as:

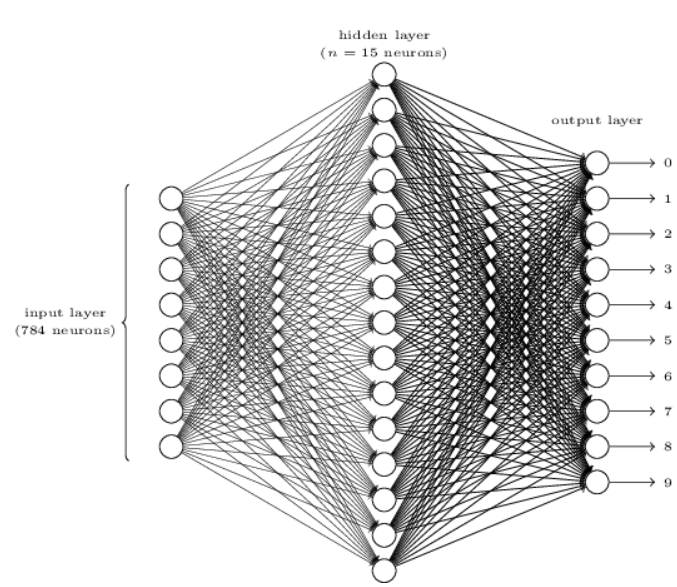


* Numbers are for nerds, I know, but this formula is amazing. It is the entire reason we can make Sigmoid Neurons learn. When the value of (w ∙ x + b) is extremely large and positive, it passes into the sigmoid function and the output for it becomes 1, as a perceptron would be. If the value of the previous statement is an “extremely negative” number, it passes into the sigmoid function giving it the value of 0. 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 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.

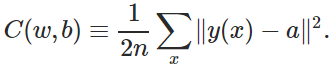
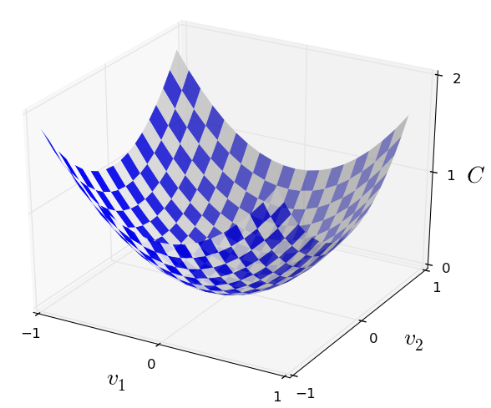
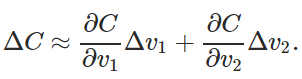
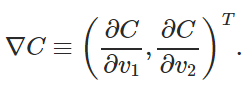
**Architecture of Neural Networks:**

* Neural networks are comprised of an Input Layer, Output Layer, and one or more Hidden Layers
  + The Input Layer is just as it sounds, our input being sent into the system.
  + The Hidden Layer(s) is just a fancy way of saying it’s neither an input nor an output layer.
  + The Output Layer is the endpoint of our network in which the calculated output is produced.
  + The above diagram shows a network of sigmoid neurons. It has an input layer, comprised of 6 neurons, two hidden layers comprised of 4 and 3 neurons respectively, and a single, output neuron within the output layer.
  + Types of Neural Networks:
    - **Feedforward Network** – This is the type of network we have discussed thus far. Feedforward networks are characterized by always “feeding information forward, never backward”. This means there are no loops within the network, which is good, as loops can cause problems within a network if not handled properly.
    - **Recurrent Network** – A type of neural network in which loops are possible. The idea is to have neurons fire for a limited time and then become dormant. After a time, another neuron fires in the same fashion and more until we have many neurons all firing briefly and going silent. Loops are allowed since everything isn’t fired immediately, a neuron in a loop’s output would not affect it immediately, its input would be altered some time later.
    - **Which One Is Better?**
      * Currently, Feedforward is more popularly used. Recurrent networks mimic human brain function more similarly but, at present, is supported by weaker algorithms in comparison to feedforward networks

**A Neural Network to Read Handwritten Digits**

* Achieving our goal will require us to break down our overall problem a little. To recognize a string of handwritten digits, we first need to break the string into segments rather than a jumble of digits, then we need to correctly classify these segments as the correct digits.
* The Segmentation Problem can be solved in many ways and is not what we are focused on in this project. We wish to correctly classify singular handwritten digits and we will (possibly) worry about segmentation later.
* **How will our network look?**
  + Our neural network will consist of 3 layers:
    - 784 neurons within the Input Layer. Why?
      * This layer will be used to encode pixel values. We are using 28x28 pixel digits for training data and 28x28 = 784.
      * The values for the input neurons will contain the greyscale value of the individual pixel, 0 for white, 1 for black, and, since we use sigmoid neurons, every value in-between.
    - 1 Hidden Layer comprised of *n* neurons. Why?
      * The Hidden Layer is something that will be experimented with to test for better results.
    - 10 neurons within the Output Layer. Why?
      * Well for the respective values of 0-9 of course! For example, if the 10th neuron fires, then the network believes the number it is processing is a 9.
      * We could achieve the same goal with just 4 output neurons, as 24 = 16 and is enough to hit all 10 desired values. Surprisingly, 10 output neurons work best for this particular problem.
        + When breaking down outputs from the hidden layer we could generally use 4 neurons per digit sought. Each neuron would be responsible for greyscale detection within a particular quarter of the image. When using 4 output neurons, the first output neuron would get stuck trying to decide which bit of the digit was the most significant, and that is not exactly an easy task when given nebulous shapes.
  + Below is a picture of what our network will look like (pictured with 15 neurons within the hidden layer):

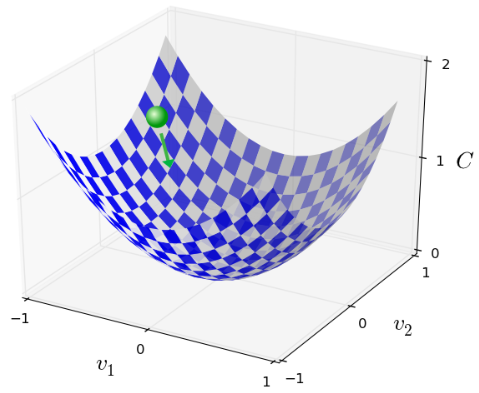
**Teaching Our Neural Network: What is Gradient Descent?**

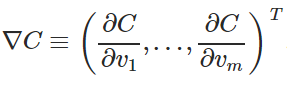
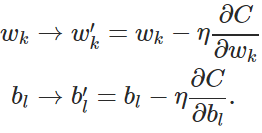
* After our code is complete, its execution will be done in two main parts: Training and Evaluation.
  + For training, we will be taking 60,000 images from the United States’ National Institute of Standards and Technology. These images consist of 28x28 pixel, greyscale images of handwritten digits taken from 250 people split between employees of the Census Bureau as well as high school students.
  + Evaluating our network will be done using 10,000 images. These images are also 28x28 pixel, greyscale images of handwritten digits. This time taken from an entirely different set of 250 individuals but still either an employee of the Census Bureau or a high school student.
* We will denote *x* to be a training input, more specifically, *x* is a 28x28 = 784-dimensional vector. Each entry within the vector represents the grey value of a particular pixel within the image.
* Desired output will be characterized by *y=y(x)* where *y* is a 10-dimensional vector.
  + For example, if a particular training image, *x*, shows a 4, then *y(x)=(0,0,0,0,1,0,0,0,0,0)T*
    - In this example, the superscript, T, is the transpose operation, turning a row vector into a column vector.
* As mentioned previously, to increase our accuracy, we need a way to adjust weights and biases so we can reach our goal. We need an algorithm that can help us do this, so we define a *Cost Function*:
* In the above function, *w* and *b* are the collections of all the weights and biases in the network, *n* is the total number of training inputs, *a* is the vector of outputs from the network when *x* is input. The sum is over all training inputs, *x*.
  + This function is also called the *Quadratic Cost Function* as well as the *Mean Squared Error*.
  + A few deductions can be made when looking at this function:
    - This is a non-negative function, since every term in the sum is non-negative.
    - More importantly, the function itself becomes small (*C(w,b) ≈ 0*) when *y(x)* is approximately equal to the output, *a*, for all training inputs, *x*. In other words, our training algorithm is doing well when *C(w,b) ≈ 0*.
      * Vice versa, the function is doing quite bad when the its output is large, meaning *y(x)* is not close to the output, *a*, for many inputs.
  + From the above information, we can see now that the goal of training will be to minimize this Cost Function by adjusting our weights and biases.
    - We can do this using an algorithm known as *Gradient Descent.*
    - But first, you may be asking, “Well why are we trying to minimize a cost function? Why are we not just focused on number of correctly classified digits?”. Good question!
      * The issue with solely worrying about number of classified digits is that it doesn’t turn out to be a *smooth function* based off the weights and biases of our network, meaning, making small changes to the weights and biases won’t have much of an effect on numbers correctly classified at all. Using a smooth cost function, we see that it’s much easier to deliver the desired results.
      * So, we begin with minimizing our quadratic cost and then look towards correct classification.
* Now we can begin our talk on **Gradient Descent**:
  + Our goal now is to minimize the Cost Function or find its global minimum.
    - For simplicity, let’s think of our cost function, graphically, as a two-variable function*, v1 and v2*. It could look something like this: 
    - Logically, to find the minimum, we could think of our function as a valley, in everyday life we could roll a ball down this valley and it would naturally, at its stopping point, be resting at the lowest portion. This is essentially how we’re going to find the minimum of our cost-function, but maybe not quite so literally.
      * So, we start by thinking about what happens when a ball is rolled a small amount, *Δv1*, in the *v1* direction and a small amount, *Δv2*, in the *v2* direction. C would change as follows: 
      * Using this, we want to find values for *Δv1* and *Δv2* such that *ΔC* is negative, or in the ball scenario, having the ball roll downhill.
      * To accomplish the negative value for *ΔC* it helps to think of *Δv* to be the vector of changes in *v*, *Δv ≡ (Δv1, Δv2)T*, T is again the Transpose operation
      * Let’s also define the *Gradient of C* to be the vector of partial derivatives: 
      * Taking the previous two expressions into consideration, the equation for *ΔC* can be rewritten in terms of *Δv* as: 
        + This equation assists us in understanding what a gradient vector is. The gradient vector allows us to relate changes in *v* to changes in *C*, but even more interesting is it allows us to choose *Δv* to make *ΔC* negative.

Suppose we choose:  where *n* is a small, positive parameter (known as the learning rate). Then the previous equation tells us:  because  this guarantees that *ΔC <= 0* or *C will always decrease, never increase.*

Because of these facts, we can change the *v* within the initial equation we chose for it and use it to define the laws of motion for our ball! So, we’ll be using the reformed equation to compute a value for *Δv*, then move the ball’s position *v* by that amount: 

We’ll continue using this update rule to make moves, decreasing *C* until we reach our global minimum. So, essentially, the way this works is we continually compute the gradient, , and then move in the ***opposite*** direction, going down into our ”valley”. We can visually represent it in this manner:



* Our ball doesn’t behave in the manner a real ball would if this experiment is done. We disregard friction, etc. and tell it to “Just go down, now.”
* We want our Gradient Descent to work correctly, and to do that we must choose a *learning rate, n,* to be small enough that the equation, , is a good approximation. Blindly choosing a learning rate can lead us to having a result of *ΔC > 0* which is not good. On the flip side, if the learning rate is too small, the changes to *Δv* will be too tiny, making our Gradient Descent algorithm work too slowly. In practice, we vary the learning rate so that it has a good balance between a solid approximation as well as having speed.
* Up until now, we’ve talked about *C* being a function with two variables, but our network will have much more than two variables. It turns out this can be represented fairly simply going off our previously defined formulas concerning two variable networks.
  + Suppose that *C* is a function of *m* variables, *v1, ….., vm*. Then the change in *ΔC* in *C* produced by a small changed *Δv = (Δv1,…, Δvm)T is:*  *where the gradient C* is the vector:
* Just like with two variables, we can choose , and we’ll be guaranteed that our dot product equation above for *ΔC* will be negative. So we can follow our gradient to a minimum, even when *C* is a function of many, many variables, by just applying our update rule repeatedly: 
  + We can think of the above Update Rule as *defining* our gradient descent algorithm. It gives us a way of repeatedly changing the position *v* in order to find a minimum of *C*. The algorithm is not perfect, many things can go wrong, but it can also work very well in many cases and is a great way of minimizing cost functions within neural networks, helping the neural network to learn.
* We use gradient descent to find the weights and biases which minimize the Cost Equation.
  + Let’s look at how this works:
    - We can rewrite out the Update Rule in terms of components, as we know, we can first replace *vj* with the components it stands for, weights (*wk) and biases* (*bl)* and the gradient vector C has the corresponding components: 
    - We now write the Update Rule in terms of components as: 
    - By repeatedly applying this rule we can roll the ball down the hill and find a minimum of our cost function, i.e. this is a rule that can be used to learn in a neural network.

**Implementing Our Network to Classify Digits:**

* Our network will consist of 3 core files:
  + A network python file, which houses our neural network and its learning algorithm (Stochastic Gradient Descent)
  + An mnist\_loader python file, which loads up the training data for us to use from the mnist database.
  + A test python file which will be the main driver program of our network.
* More detail on what each file specifically does:
  + Our network file uses the built-in random library to be allowed the use of them random function and the numpy library, a 3rd-Party Python library, which allows many different scientific calculations to be handled easily and efficiently.
    - Within the Network class, we define our constructor to create a list called *sizes* which contains the number of neurons for the respective layers of the network, i.e. if the list was [2, 3, 1] then the network would be 3 layers with the input layer containing 2 neurons, 3 neurons in the hidden layer, and 1 neuron in the output layer.
    - We define a feedforward function which keeps us from going into any loops.
    - We use an SGD function (Stochastic Gradient Descent) to train our neural network.
      * Stochastic Gradient Descent, more specifically using mini-batch SDG allows us to seek the global minimum of our cost function without calculating the entire graph, we calculate a few and choose the best option to minimize our cost.
        + Our SDG function works as follows. In each epoch, it starts by randomly shuffling the training data, and then partitions it into mini-batches of the appropriate size. This allows us to easily sample the training data randomly. For each mini\_batch we apply a single step of gradient descent using the command “self.update\_mini\_batch(mini\_batch, eta)”. This update the network weights and biases according to a single iteration of gradient descent, using just the training data in mini\_batch
    - After we define our SDG, we define update\_mini\_batch function which, as previously mentioned, updates the weights and biases according to a single iteration of gradient descent.
      * Most of the update\_mini\_batch function’s work is done by the line “delta\_nabla\_b, delta\_nabla\_w = self.backprop(x, y)”, which invokes the backpropagation algorithm, which is a fast way for us to the gradient descent of the cost function.
      * It works, “simply”, by computing gradients for every training example in the mini\_batch, and then updating self.weights and self.biases appropriately. (Note: I used a recommended backpropagation function and am not entirely sure how it works, supposed to be covered in another chapter of text). Looking at other sources, Backpropagation is an algorithm that can tell the network when it has made a mistake in making a certain prediction.

**Sources**

* Nielsen, Michael (2018). *Neural Networks and Deep Learning.* Retrieved from http://neuralnetworksanddeeplearning.com
* LeCun, Y., Cortes, C., & Burges, C. (unknown). The MNIST Database of Handwritten Digits. Retrieved from http://yann.lecun.com/exdb/mnist/
* Dobrazanski, M.D., Neural Network for Handwritten Digits, (2016), DeepLearningPython35, https://github.com/MichalDanielDobrzanski/DeepLearningPython35
* A Beginner’s Guide to Backpropagation in Neural Networks, (Unknown), Retrieved from https://skymind.ai/wiki/backpropagation