Claudia Schmidt
Sussman Lab
April 28, 2023

Spring 2023 Semester Summary

Neural Network Background

During my first several weeks working with Professor Sussman, I focused on acquiring a base understanding of neural networks and how they work. I started by understanding the purpose of these networks and why they are so important and then moved on to learning about how these networks produce and optimize outputs. Below is a summary of my knowledge thus far.

Our brains can recognize visual patterns in almost everything we do. The difficult part is teaching computers how to recognize these same patterns. Handwritten digits are the most common example used in teaching computers how to learn and function like a human brain. Handwritten digits contain many variations and flavors that are specific to each individual person. The goal is to train computers to recognize commonalities among these digits and to infer rules about each digit in order to correctly classify them. As you feed in more and more examples, the computer learns more and more and improves its ability at correctly classify each digit.

To teach computer to learn, programmers use neural networks. Sigmoid neurons are the most useful type of artificial neuron for neural networks. Sigmoid neurons use weights and biases. Weights determine how much a piece of input data influences the output while biases are additional constant inputs for the next layer of neurons. Each neuron in a network has a specific bias, and each edge or path from one neuron to another has a weight.

The important part about sigmoid neurons is that small changes in their weights/biases cause only small changes in the output. As such, it is possible to finely tune these weights and biases to get a specific, desired output. Because of this, the output values of sigmoid neurons are not binary but instead can be any real number between 0 and 1. Specifically, the output produced by a sigmoid neuron is the sigmoid function described by the equation $\sigma(z) = \frac{1}{1+e^{-z}}$. The output for a sigmoid neuron with multiple inputs is $\sigma(z) = \frac{1}{1+e^{-\Sigma z}}$. The ‘z’ value in this equation is often

thought of as a message that neurons send to each other. This value is calculated using the equation $z = w * a + b$. In this equation, 'w' is the sum of all the weights inputted into the neuron. 'b' is the bias of the neuron, and 'a' is the activation value of the neuron (any value between 0 and 1). These messages, 'z', are used to calculate the activation value of each neuron in the following layer. A single neuron often receives many different messages or 'z' values. It sums all these message values and multiplies that sum by the sigmoid function to determine its activation value ($a = \sigma[\sum z]$). These activation values determine if that neuron is important in the output of the network.

Sigmoid neurons are often involved in feedforward neural networks in which the output of one layer is used as the input for the next later. In these networks, the input layer is the leftmost layer, the middle layer(s) is the hidden layer, and the last layer is the output layer. For example, when classifying handwritten digits, the output layer contains 10 neurons labeled 0 through 9. The neuron with the highest activation value (closest to 1) determines which number the network is classifying the input as. For example, if the first neuron has the highest activation value, the network thinks the input digit is a 0.

Many of these neural networks use stochastic gradient descent (SGD). The cost function is a necessary part of the SGD process because it describes how well we are achieving our goal. The cost function is defined by the equation $C(w, b) = \frac{1}{2n}\sum||y(x) - a||^2$. In this equation, 'w' is the collection of all the weights in the network. 'b' is all the biases. 'n' is the total number of training inputs, and 'a' is the output value. Y(x) is the desired output. The goal is the minimize the cost function by finding the weights and biases that produce an output close to the desired output. The gradient provides a way to minimize the function. To understand this, it is helpful to think about a ball rolling down a hill. You want to continuously change the position of the ball in the downward direction until you reach the lowest point. This lowest point is where C=0. You reach this point by making small changes to the weights and biases throughout the network until you reach an optimal output.

To do this, SGD uses backpropagation. Qualitatively, backpropagation describes how fast the cost function is changing as the weights and biases change. The process of backpropagation begins with the output layer. The error at the output layer is calculated by multiplying the rate of change of the cost function by the rate of change of the activation. This is calculated at each neuron in the output layer. If the cost function does not rely much on a specific neuron, then the

error at that neuron will be small. The process then works backwards through the network. The error at a certain neuron is layer ‘l’ is calculated using the weights and biases in the layer ‘l’+1. The process moves backwards to understand how the cost is affected by earlier weights and biases. After moving through the entire network, it is then possible to calculate the gradient of the cost function at each neuron. To calculate the gradient with respect to the weight, you multiply the activation of the neuron in layer ‘l’-1 by the error of the neuron in layer ‘l’. The gradient with respect to the bias is simply the error at that neuron. The weights and biases of each neuron in the network are then updated in an attempt to optimize the output.

Glassy Physics

After I had thoroughly read about, learned, and traced these neural networks, I went on the explore some of the principles behind the glassy physics part of the Sussman Lab. I did not spend enough time on these glassy concepts to gain a thorough understanding, but I was able to learn about the basic concepts. There were two main areas I explored with regards to the glassy physics: the basic concepts behind glassy physics and the Tomi’s previous research/data.

Basic Concepts: Glasses are interesting because they demonstrate characteristics of both crystals and liquids. They have elasticity like crystalline solids but can also flow like liquids. Additionally, glasses are not naturally occurring and only form under certain circumstances. There are a couple different ways they can form. One way is by cooling a liquid below its freezing point until it reaches a rigid condition. Another is by compressing a liquid and finding the pressure at which the transition to glass occurs. For example, glass glucose can be formed via undercooling. When glucose (in its melted state) is cooled below room temperature, the transition to glass occurs at a specific point. At this point, the expansivity and heat capacity are similar to crystalline glucose, but its volume and heat content are like liquid glucose.

Glassy Data: Throughout the semester, I worked with data generated in Tomi’s previous simulations. As such, I spent some my time understanding where this data came from and what each data point represented. Tomi ran simulations of glassy particles at different temperatures ranging between 0.45 and 2.0 (LJ units). At each temperature, there were 4,096 particles. He generated the trajectory of each particle and analyzed it. For example, consider a single particle with a single trajectory. He looked at a small part of that trajectory, split it in half, and determined how far each point in the first half was from the average of the second half (and visa

versa). That window was then slid across the entire trajectory, and the same process was repeated. When there was a change in the trajectory, the Phop function (from Tomi's paper) was very big. Otherwise, it was small. This process was used to determine if the particle was rearranging (output = +1) or not rearranging (output = 0). For each particle in the data set, there is a corresponding list of 100 features describing the local radial distribution around that particle. Each particle also has a corresponding label, either a 0 or a 1, that indicates whether the particle is rearranging or not rearranging.

Python Implementation

Tomi used Support Vector Machines to explore these fluid structures. However, instead, I generated and applied the neural networks (as described in the first section of this document) to explore these data sets. My main goal was to train a network against the lowest temperature data set (0.45), individually feed the other data sets into this trained network, and explore the results. I then planned to repeat this process for the other data sets, training separate networks against each data set and feeding the other sets into each network. In total, there would be 6 different trained networks (for each of the 6 data sets), and all 6 data sets would be inputted to each network. In doing so, I hoped to compare the classification accuracy of each network at each data set. We hypothesized that as the temperature increased, the classification accuracies would decrease. However, we expected each neural network to classify best at the data set it was trained on.

To do this, I used the sklearn.neural_network package, imported the MLP (multi-layer perceptron) Classifier to my local environment, and began to understand how to customize it. The following parameters are the ones I focused on.

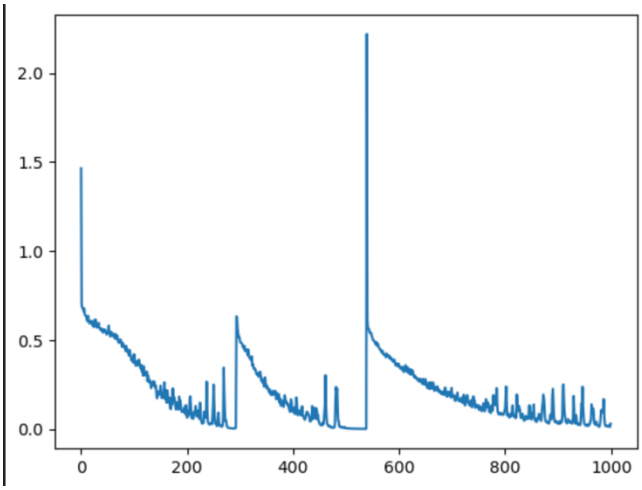
- solver determines the type of weight optimization solve. I mostly used the 'adam' or 'sgd' solvers.
- hidden_layer_sizes constructs the architecture. For example, (20,20,20) represents 3 layers each of size 20
- max_iter sets the number of steps the network iterates through. The solver stops when it either hits convergence set up 'tol' or when it reaches this number of iterations.
- tol sets the tolerance for the optimization. When the accuracy score (described in more detail later on) is not improving by at least 'tol' for a set number of iterations, the solver stops. This set number of iterations is determined by the parameters

'n_iter_no_change'. I wanted the solver to iterate through all the iterations, so I set the 'tol' to 0 and the 'n_iter_no_chnage' equal to 'max_iter'.

- To calculate how well the network was predicting/learning, I imported 'accuracy_score' from sklearn.metrics.

I first attempted to find to optimize an architecture for the network trained on the 0.45 data that produced the most accuratly predicted the labels of the 0.45 data. The accuracy score produces values between 0 and 1. As such, a value close to 1 meant the network was correcting predicting the data set labels (0's and 1's). I first tried to do this in a methodical fashion, by exploring each parameter one at a time and progressively adding the 'optimized' parameters to the network. For example, I started with a network only characterized by its solver ('adam') and the number of hidden layers. At this point in the process, I thought the syntax for defining the number of hidden layers was 'hidden_layer_sizes = (layers, width)' where layers was the number of layers and width was the number of neurons in each layer. As such, I iterated through various numbers of layers and widths and found that 1000 layers each of width 60 produced optimal results. However, as I would later find out, this architecture was actually only two layers, one of width 1000 and the other of width 60. I repeated this same iterative process with the learning rate and number of iterations. I found that the optimal number of iterations was 280 and learning rate was 0.0001. This architecture produced accuracy scores between 0.68 and 0.7 which were not bad but definitely could be better. I also experimented with using the 'adam' and 'sgd' solvers and found that the 'adam' solver produced better accuracy scores. In one of our next meetings, Daniel helpfully corrected the syntax for the hidden layer sizes.

The next big challenge I faced was trying to understand how many iterations the solver needed run through in order to arrive at its best accuracy score. This number could have been anywhere between 10 to 10^7. I chose a somewhat random architecture (5 layers of size 100) and set the max number of iterations to 1,000. I then looked at the loss curve (pictured to the right) to get an understanding of how the cost function changed over time. From this curve, I looked at where the curve reached minimum values. As such, I was able to determine that the number of iterations I needed to run through was most likely between 10^2.5 and 10^3.5.



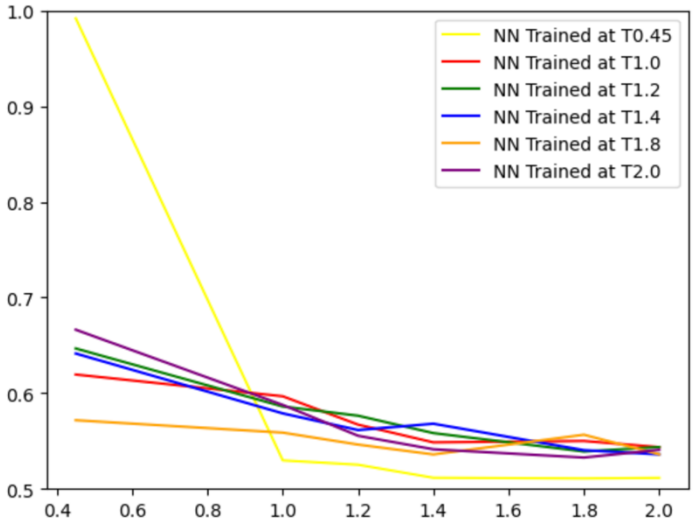
To further optimize the network parameters, I used logspace to create a set of logarithmically spaced time/iteration values in the range 10^2.5 to 10^3.5. I chose a set of networks with number of layers [2,4,8,10] and widths [20,40,60,80]. I chose these smaller networks because I believed I could still get good results while also cutting down the number of parameters in the network (weights and biases). This translated to a more manageable running time. I iterated through these three variables (times, widths, and layers) and calculated the accuracy score at each variable triplet. There were a couple different combinations that produced accuracy scores of 1. However, I chose the combination with the least number of iterations in the interest of cutting down running time. This combination was 10^2.5 iterations, 8 layers, and widths of 80. I then tried to find the smallest network that would still produce an accuracy score of 1 while maintaining the same number of iterations. I found this architecture to be 4 layers each with 100 neurons.

At this point, I was confident I had found an optimized network architecture, so I applied this architecture to each of the 6 networks, fed all the data sets into each network, and compared the accuracy curves. We expected:

- The accuracy of the 0.45 network on the 0.45 data to be the highest classification accuracy on the entire graph (close to 1).
- Each network to best predict the 0.45 data (no matter which data set it was trained on).
- Each network's curve to slope downward as the temperature of the inputted data sets increased.

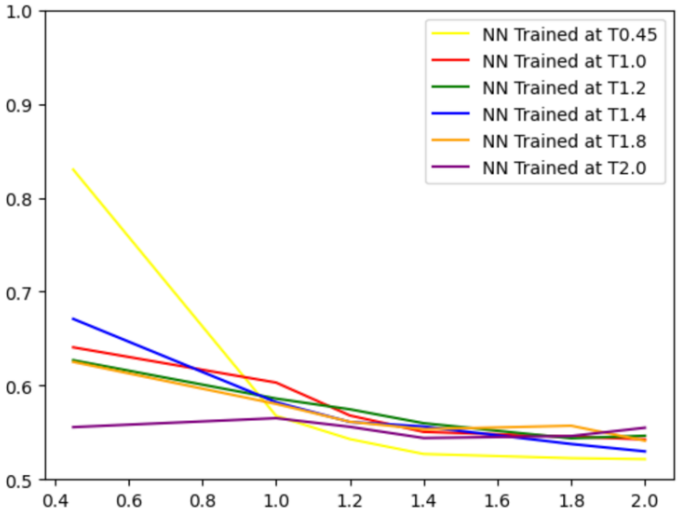
- Each network's curve to peak slightly at the temperature of the data set for which it was trained and then continue its downward slope.
- The curves to be stacked on top of each other, with the 0.45 network curve on top and the 2.0 network curve on the bottom. For example, we expected the classification accuracies of the 0.45 network to be higher than all the other networks at each of the data sets. Then, we expected the classification accuracies of the 0.1 network to be second highest (compared to all the other curves) at each of the data sets (slightly below the 0.45 curve). The rest of the curves would follow in the same fashion in order of increasing temperature.

This is the graph of my results (x-axis = temperature, y-axis = classification accuracy). These were not the results I was hoping for. I expected the accuracies of the other networks to be higher overall (especially at the 0.45 data set). I also expected the accuracy of the 0.45 network to be higher for the other data sets. Instead, the curve plumets to a value close to 0.5 for temperatures 1.0-2.0. Additionally, the order of the curves is mixed (not stacked correctly). They all have roughly the same accuracies at each of the data sets.

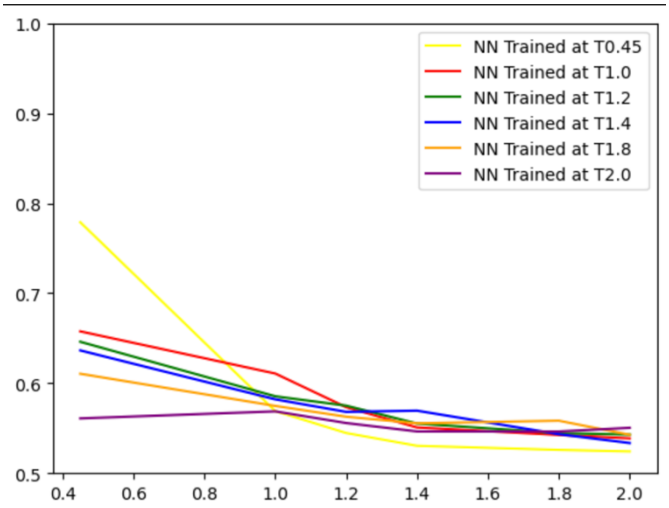


After sharing my findings, Daniel hypothesized that the networks were not learning despite how well the 0.45 network was able to predict the 0.45 data set. Instead, he suggested that the network was memorizing the data set and corresponding labels instead of predicting them. This is why the other accuracy scores were so low across the board. The network was bad at learning. To prevent this, we discussed a few different approaches. The first was use a network architecture that produced an accuracy score close to 90% (when the 0.45 data was applied to the

0.45 network). I identified this architecture from my previous trials and applied it to the 6 networks. These are the results I produced. There were some small improvements but not much. There was more separation between the curves as they were not as jumbled together. Additionally, the order in which they were stacked was almost what we expected but not quite.



The other approach Daniel suggested was using cross validation. Cross validation involves splitting the data sets into two parts, training the network on the first half, and testing the network on the second half. This helps to assess how accurately the network actually learns. When doing this, the loss curves for both halves of the data set should be about the same. As such, I attempted to find a network that produced a good cross validation score. To do this, I used the 'cross_val_score' from sklearn. I started by finding the cross validation score for the network architecture I previously believed to be optimized, 4 layers each with width 100 (number of iterations = 10^2.5). The score was 0.56 (not very good). I then tried increasing and decreasing the number of layers while maintaining widths of 100. When I did this, the cross validation score only varied by 2-3%. I looked at the loss curves for these architectures and realized the loss curves were still decreasing when they reached 10^2.5 iterations. As such, I decided to increase the number of iterations to 10^3 (if I increased it any more, the running time was too long). However, this produced the same score. I then tried increasing the number of layers in increments of 2. With each increment, the score increased by about 1%. I stopped incrementing when I reached 10 layers because the running time was too long. I then decided to decrease the widths to 40 while maintaining 10 layers. The score increased to 0.61. I then decreased the widths to 20, and this increased the score to 0.63. I then decreased the widths to 10, and the score increased to 0.67. However, further decreasing the widths did not



increase the score. As such, the best cross validation score I was able to generate was 0.67. I applied this architecture to the six different networks and plotted my results. This is the graph that I produced. The 0.45 network accurately predicts the 0.45 data 77% of the time. This accuracy score is less than the that of the previous two graphs. However, the overall graph is a little closer to what we expected. All the curves have peeks at the data sets they were trained on. Additionally, the curves are stacked in the correct order at the 0.45 data set. For example, at the 0.45 data, the 0.45 network produces the best classification accuracy, then the 0.1 network, then the 1.2 network, and so on. However, the curves get a bit jumbled as they slope downwards. Going forward, I think I can optimize this architecture a bit further, but I would need to discuss with Daniel on how to best move forward. I think I may need to increase the number of iterations to a number larger than 10^3.