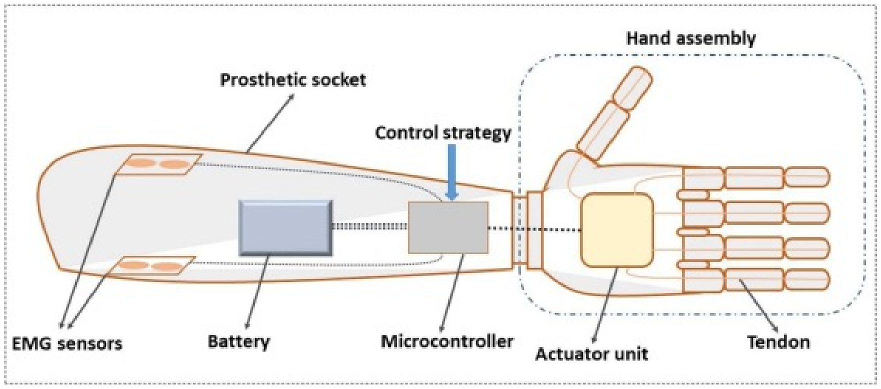
Using Gradient Descent to Optimize A Neural Network For Classification of Hand Gestures

Abstract

Electric prosthetic hands are typically controlled using two electromyography sensors. While they have been used for decades, they exhibit significant disadvantages that recent research has tried to solve using arrays of force sensors. To convert these signals into accurate predictions of the hand gesture the user is attempting to make, an algorithm such as an artificial neural network must be used. This paper details the process of training a neural network to accurately predict hand gestures based on sensor data using the method of steepest gradient descent. Overall, the method was successful, in correctly classifying 91% of signals, although further research would be beneficial in further improving the accuracy of the model.

Motivating problem

Most modern powered prosthetic hands are controlled by the user through EMG (electromyography) signals. Typically, two sensors are placed on the user's residual forearm, and by flexing one of the two main forearm muscles, the user can 'swipe through' a set of pre-programmed hand grips/gestures.



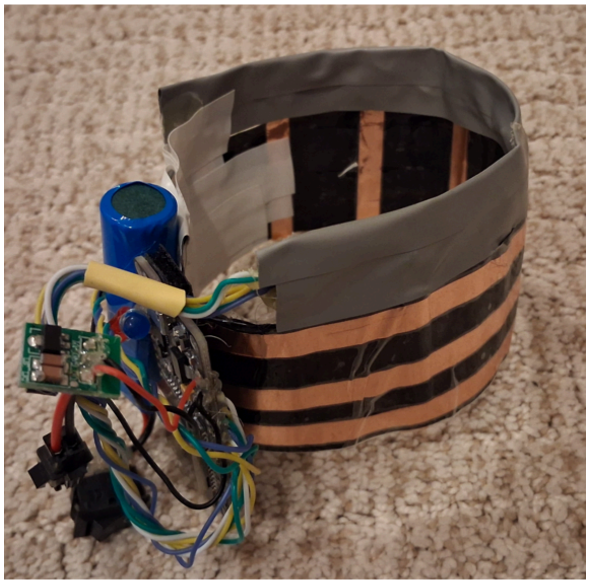
This has been a relatively reliable method of control for decades now, however it comes with its own drawbacks. Many users say that the complication of switching to the correct grip, and lack of fine control of their prosthetic are the main reason they dislike their prosthetic or avoid using it (1). Additionally, EMG signals typically have high noise, and become unreliable when misplaced or subject to moisture from sweat (2). This has led researchers to explore using force myography (FMG) (force sensors) as a replacement or supplement for EMG signals in prosthetic use. FMG sensors are cheaper, can be more densely packed in a small area, and are not affected by sweat or noise to the same degree. In order for this method to work, FMG data must be directly interpreted into hand grip selection (rather than cycling through grips).

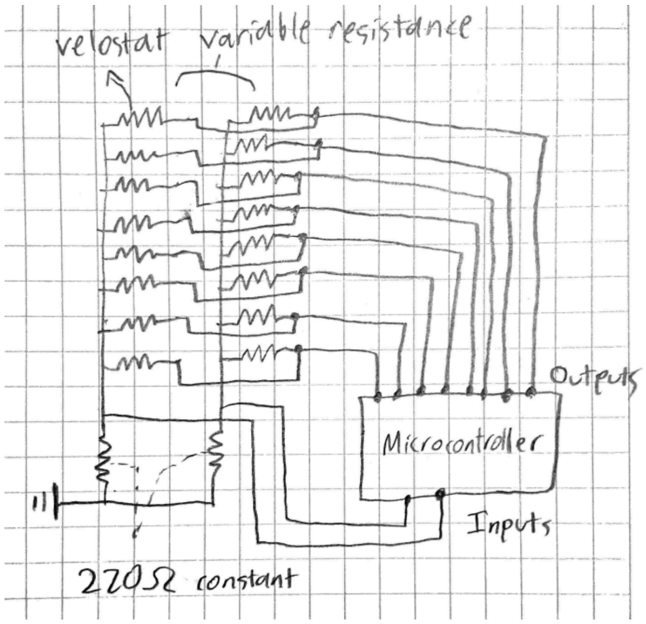
Literature Review

A 2023 study by Rehman et. al. found that using force sensors was a viable method to predict hand gestures. In their research, a 7 sensor band using a support vector machine algorithm to classify 5 gestures had an accuracy of 91.6% (2). A similar study from the University of Windsor used the same algorithm with a slightly higher accuracy of 93% (3). A 2025 study from Meta's Reality lab was able to achieve over 90% accuracy of gesture recognition using an EMG wristband and a neural network architecture (4). While the first two studies mentioned achieved high accuracy using the SVM algorithm, they only had a few (<10) participants, and it's unclear how these results would transfer to a new user, one with a different physiology. The Meta study had over 6,00 participants and was only able to achieve good results for the general population by training neural networks over thousands of participants. While the model they used (neural network with over 15 layers) is far beyond what will be discussed in this paper, it proves that neural networks are powerful and scalable enough to tackle the motivating problem.

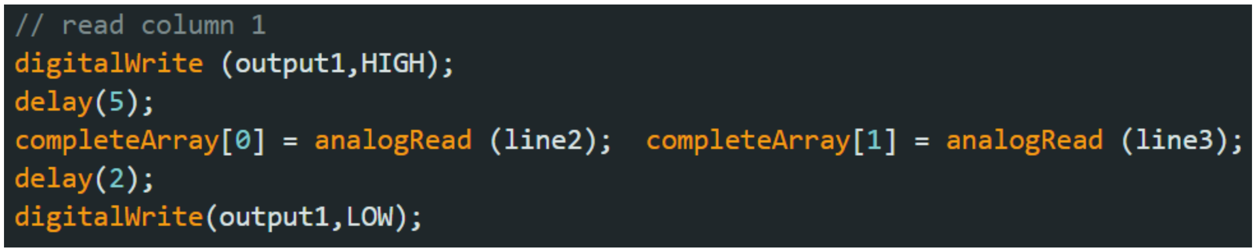
Background

In order to collect enough FMG data to predict hand gestures, an array of pressure sensors in a small area is needed. Previous research in this field has used commercially available individual force sensor resistors to accomplish this. While this has worked, a fully integrated design was chosen for its cost effectiveness and ability to increase sensor density in future iterations. The sensor mainly consists of a sheet of velostat sandwiched between two layers of rows of copper tape running perpendicularly. Velostat is a polymer impregnated with electrically conductive carbon black. As pressure is applied, the carbon black particles come closer to each other, reducing electrical resistance. Each of these 'resistors' is placed in series with a constant resistor forming a voltage divider circuit, as can be seen in the schematic below. Voltage is applied across both resistors, and the voltage across the constant resistor is measured by a microcontroller (ESP32), giving us the data we need to implement FMG control.



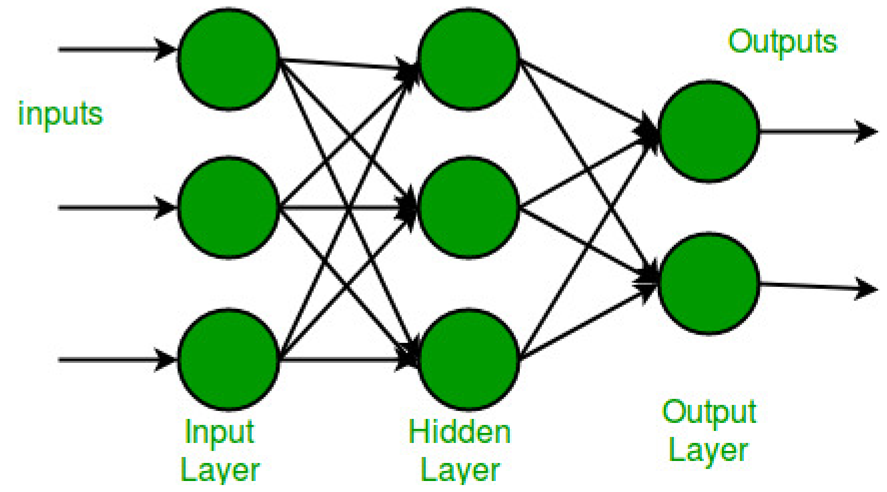


Below is a section of code written in the Arduino programming language that measures the voltage difference of one column of sensors (2 sensors).



```
// read column 1
digitalWrite (output1,HIGH);
delay(5);
completeArray[0] = analogRead (line2);  completeArray[1] = analogRead (line3);
delay(2);
digitalWrite(output1,LOW);
```

In order for this method to work, FMG data must be directly interpreted into hand grip selection (rather than cycling through grips). To do this, an artificial neural network (ANN) was used. An ANN connects a set of input nodes with a set of output nodes through an intermediary set of nodes often referred to as a ‘hidden layer’. Each node in a layer is taken as a weighted sum of the prior layer passed through an activation function. (each previous node is multiplied by a parameter ‘weight’, summed, added to another parameter ‘bias’ and passed through a non linear ‘activation’ function). The purpose of the activation function is to make the ANN non linear, which allows the network to better handle complex, high dimensional problems. The diagram below represents a simple neural network where the lines connecting the layers represent the weights.

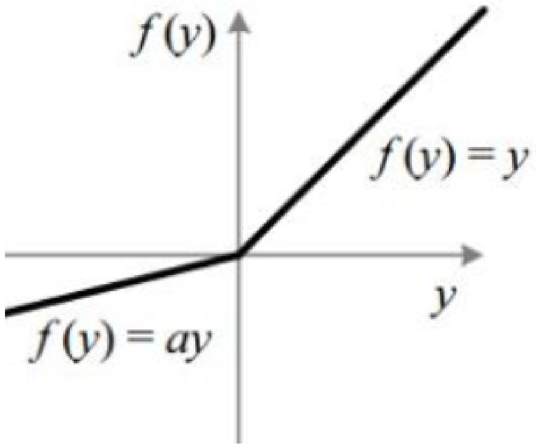


For the purposes of this paper, what is important to understand is that a neural network can be thought of as a very large non-linear function with many parameters (weights and biases). These parameters all must be adjusted based on training data in order to give the correct output for the given input. In concept, this can be thought of as similar to polynomial regression, where a polynomial can be fit to a set of data by tweaking the polynomial's coefficients.

The method to adjust the ANN weights and biases is called back propagation. To train the ANN, training data (in this case, about 200 examples), which contains both the inputs and outputs, is used. The first training example input will be plugged into the neural network, with the weights and biases initially randomized. The output of the ANN will then be compared to the true output, using a cost function. The cost is a function of all of the ANN's weights and biases. The more 'wrong' the ANN is in its prediction, the higher the cost will be. The gradient of the cost function can then be found, which will be explained in more detail in the approach section. The negative of the gradient shows the direction and magnitude of the changes to the weights and biases that will lead to the steepest decrease in the cost function, thus improving the accuracy of the ANN. This is a process known as steepest gradient descent. This process is then repeated for all (roughly 200) of the training examples, until the cost has reached a minimum.

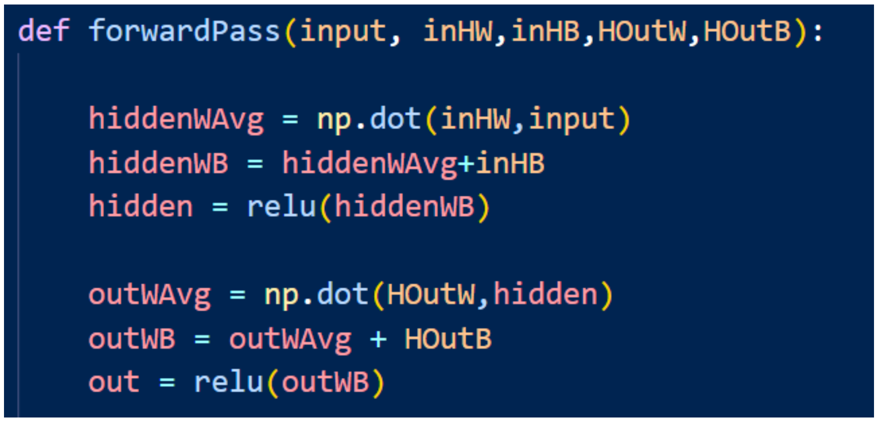
Approach

The input of the neural network is the 16 FMG sensor values, and the output is 4 nodes corresponding to hand gestures (open, close, point, pinch) respectively. For example, an output of [0,0,1,0] would correspond to the 'point' gesture. After testing several structures, an ANN with 1 hidden layer of 18 nodes (along with an input layer of 16 nodes and output of 4 nodes) using the 'leaky ReLu' activation function was chosen, as it was the least computationally expensive model that worked well.



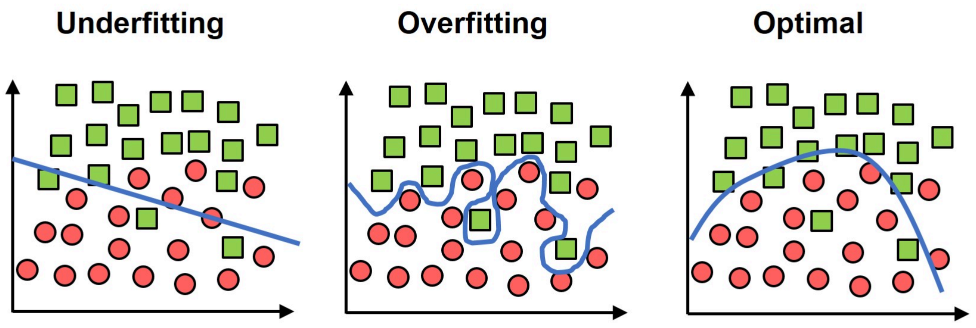
The graph above represents the leaky ReLu function. In this project, an (a) value of 0.1 was used.

This model has 382 weights and biases which are the design variables that must be adjusted. While the neural network can seem very complicated at first, it can be implemented relatively simply as seen by the few lines of python code below.



```
def forwardPass(input, inHW,inHB,HOutW,HOutB):

    hiddenWAvg = np.dot(inHW,input)
    hiddenWB = hiddenWAvg+inHB
    hidden = relu(hiddenWB)

    outWAvg = np.dot(HOutW,hidden)
    outWB = outWAvg + HOutB
    out = relu(outWB)
```

In the code, inHW and inHB are the weights and biases between the input and hidden layers, and HOutW and HOutB are the parameters between the hidden and output layers. The next step is to find the cost of the neural network, which is a function of all the parameters (weights and biases). This will be the objective function that we will seek to minimize. We will also add an inequality constraint to each of the weights to limit their absolute value. Research has shown that doing so helps to prevent the neural network from overfitting (5), which is represented in the diagram below.



The optimization problem can now be expressed in the standard form as seen below, where w1 and b1 are the parameters between the input and hidden layer, and w2 and b2 are the parameters between the hidden and output layer. w1 and w2 are matrices, and b1 and b2 are vectors. In the summation steps, 4 is representative of the 4 output neurons, 1 for each hand gesture, and T represents the total number of training examples.

$$\text{Min} \quad C\left([w1],[b1],[w2],[b2]\right) = \sum_{j=1}^{T} \sum_{i=1}^{4} \left(output_i - actual_i\right)$$

Subject to:

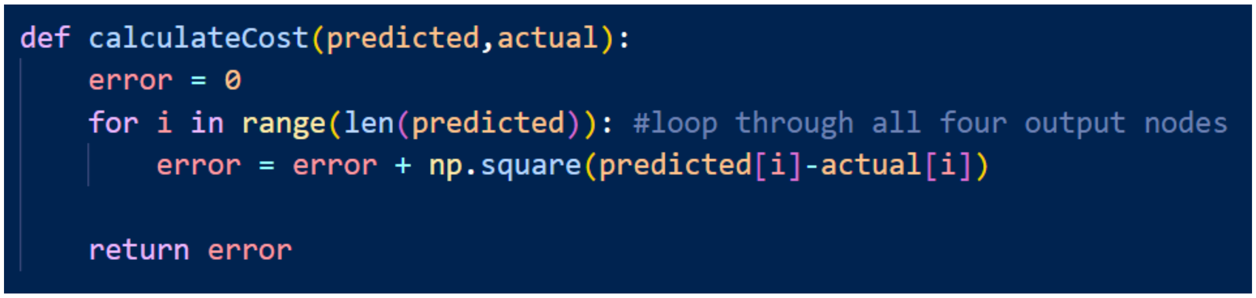
$$w1 - 10 < 0$$

$$w2 - 10 < 0$$

$$-w1 - 10 < 0$$

$$-w2 - 10 < 0$$

The code below calculated the cost function for a single training example, which can then be summed across the entire dataset.



```
def calculateCost(predicted,actual):
    error = 0
    for i in range(len(predicted)): #loop through all four output nodes
        error = error + np.square(predicted[i]-actual[i])

    return error
```

As previously mentioned, the approach to minimize the cost function was decided to be steepest gradient descent. The first step in this process is to determine the gradient of the cost function. The gradient in simpler terms can be thought of as the slope of the function. The gradient is a vector that for each independent variable (weights and biases), will represent the direction and magnitude of steepest ascent. For example, if a value in the gradient vector corresponding to a certain weight (Wa) is positive, that means that increasing Wa will increase the overall cost function. However, we want to minimize the cost function, so we will find the negative of the gradient. We must find the partial derivative of the cost with respect to each weight and bias.

To do this, the chain rule from calculus must be applied. The partial derivative of a weight between the hidden and output layer for example, is equal to the product of partial derivative of the cost with respect to the output node, multiplied by the partial derivative of the output node with respect to the activation function, multiplied by the partial derivative of the weighted sum (output node before it was passed through the activation function) with respect to the weight as is represented by the equation below.

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial Out_i} \cdot \frac{\partial A}{\partial Z_i} \cdot \frac{\partial Z_i}{\partial w_{ij}}$$

Similarly, the partial derivative of the cost with respect to a bias in the output layer is given below. The last term of dz/db is equal to 1 and thus can be dropped.

$$\frac{\partial C}{\partial b_i} = \frac{\partial C}{\partial Out_i} \cdot \frac{\partial A}{\partial Z_i}$$

The partial derivative of the cost with respect to a weight between the input and hidden layers is given below.

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial H_i} \cdot \frac{\partial A}{\partial Z_i} \cdot \frac{\partial Z}{\partial w_{ij}}$$

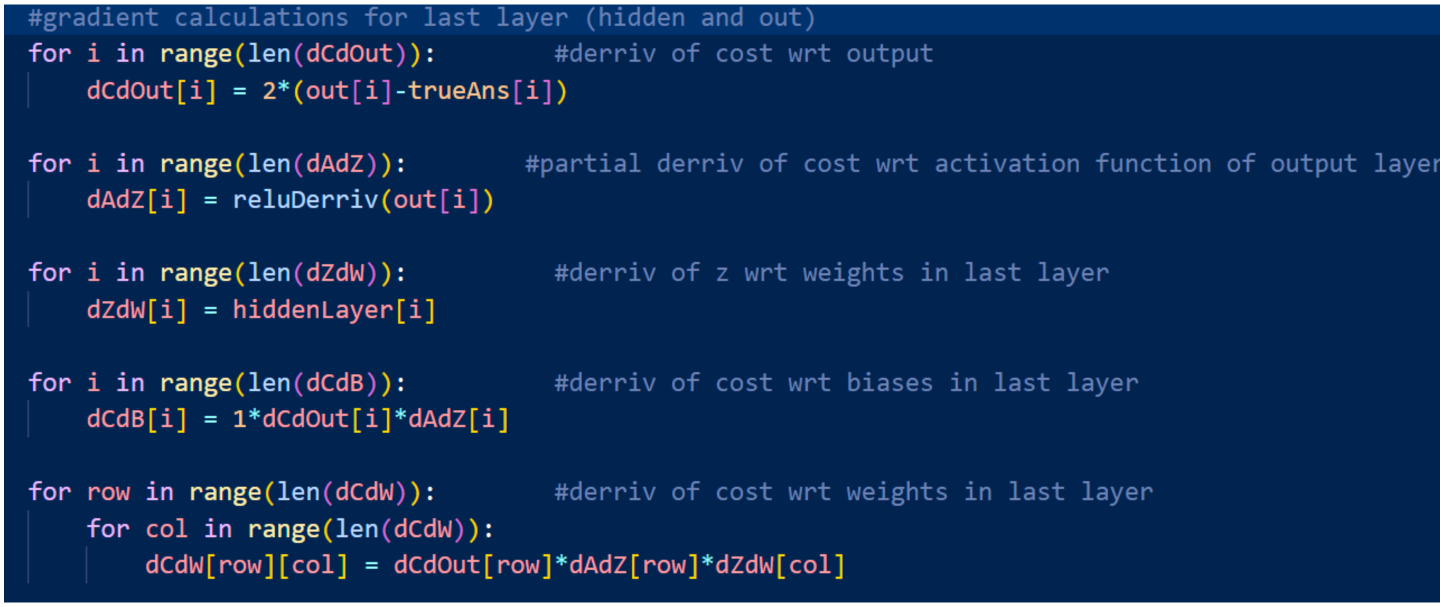
The partial derivative of the cost with respect to a bias in the hidden layer is given below.

$$\frac{\partial C}{\partial b_i} = \frac{\partial C}{\partial H_i} \cdot \frac{\partial A}{\partial Z_i} \cdot \frac{\partial Z}{\partial w_{ij}} \cdot 1$$

The partial derivative of the cost with respect to the hidden layer node (first term in the above equations) is given below.

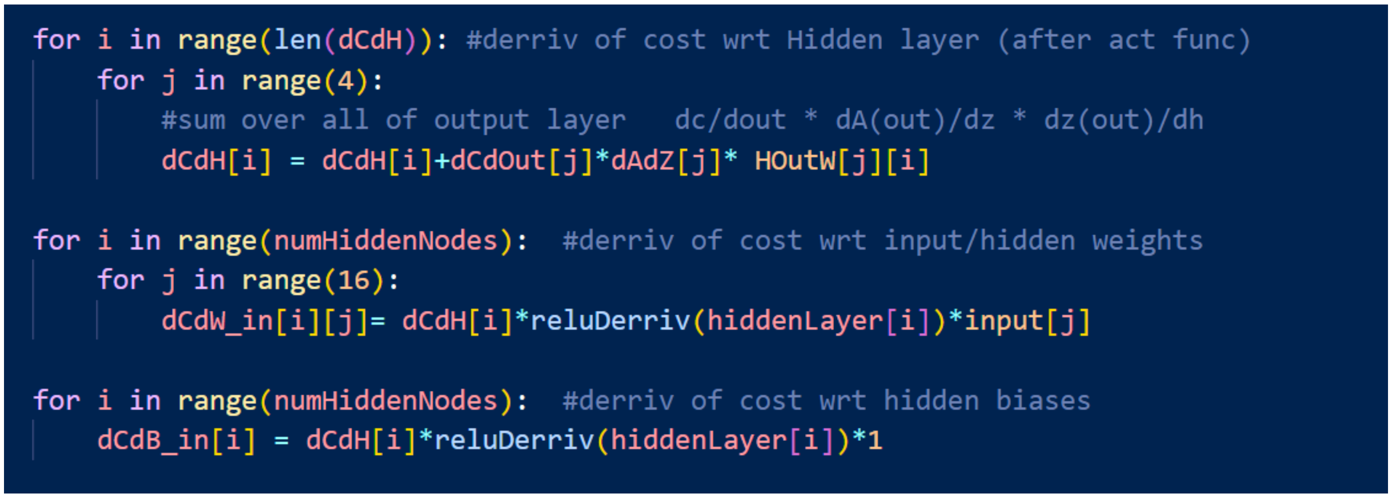
$$\frac{\partial C}{\partial H_i} = \sum_{j=1}^{4} \left(\frac{\partial C}{\partial Out_j} \cdot \frac{\partial A}{\partial Z_j} \cdot \frac{\partial Z_j}{\partial H_i} \right)$$

Below is the code that calculates the partial derivatives of the cost with respect to the parameters between the hidden and output layers.



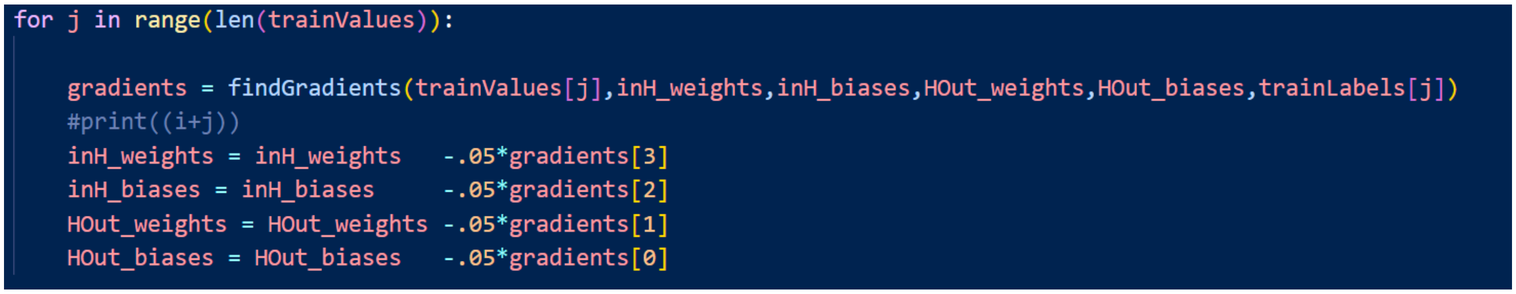
```
#gradient calculations for last layer (hidden and out)
for i in range(len(dCdOut)):        #derriv of cost wrt output
    dCdOut[i] = 2*(out[i]-trueAns[i])

for i in range(len(dAdZ)):        #partial derriv of cost wrt activation function of output layer
    dAdZ[i] = reluDerriv(out[i])

for i in range(len(dZdW)):          #derriv of z wrt weights in last layer
    dZdW[i] = hiddenLayer[i]

for i in range(len(dCdB)):          #derriv of cost wrt biases in last layer
    dCdB[i] = 1*dCdOut[i]*dAdZ[i]

for row in range(len(dCdW)):        #derriv of cost wrt weights in last layer
    for col in range(len(dCdW)):
        dCdW[row][col] = dCdOut[row]*dAdZ[row]*dZdW[col]
```

Likewise, the code for the partial derivatives of the cost with respect to the parameters between the input and hidden layer is given below.



```
for i in range(len(dCdH)): #derriv of cost wrt Hidden layer (after act func)
    for j in range(4):
        #sum over all of output layer   dc/dout * dA(out)/dz * dz(out)/dh
        dCdH[i] = dCdH[i]+dCdOut[j]*dAdZ[j]* HOutW[j][i]

for i in range(numHiddenNodes):  #derriv of cost wrt input/hidden weights
    for j in range(16):
        dCdW_in[i][j]= dCdH[i]*reluDerriv(hiddenLayer[i])*input[j]

for i in range(numHiddenNodes):  #derriv of cost wrt hidden biases
    dCdB_in[i] = dCdH[i]*reluDerriv(hiddenLayer[i])*1
```

Once the gradient has been calculated, the next step is to adjust all 382 design variables (weights and biases) to reduce the cost function, thereby making the model overall more accurate at correctly recognizing the 4 hand gestures. The way to do this is by taking the negative of the gradient, multiplying it by a scalar (fraction to reduce magnitude, prevents oscillations and overshooting), and adding the results to the design variables. This process is repeated over all training examples, until the cost function reaches a minimum. The code to accomplish this step of gradient descent is shown below.

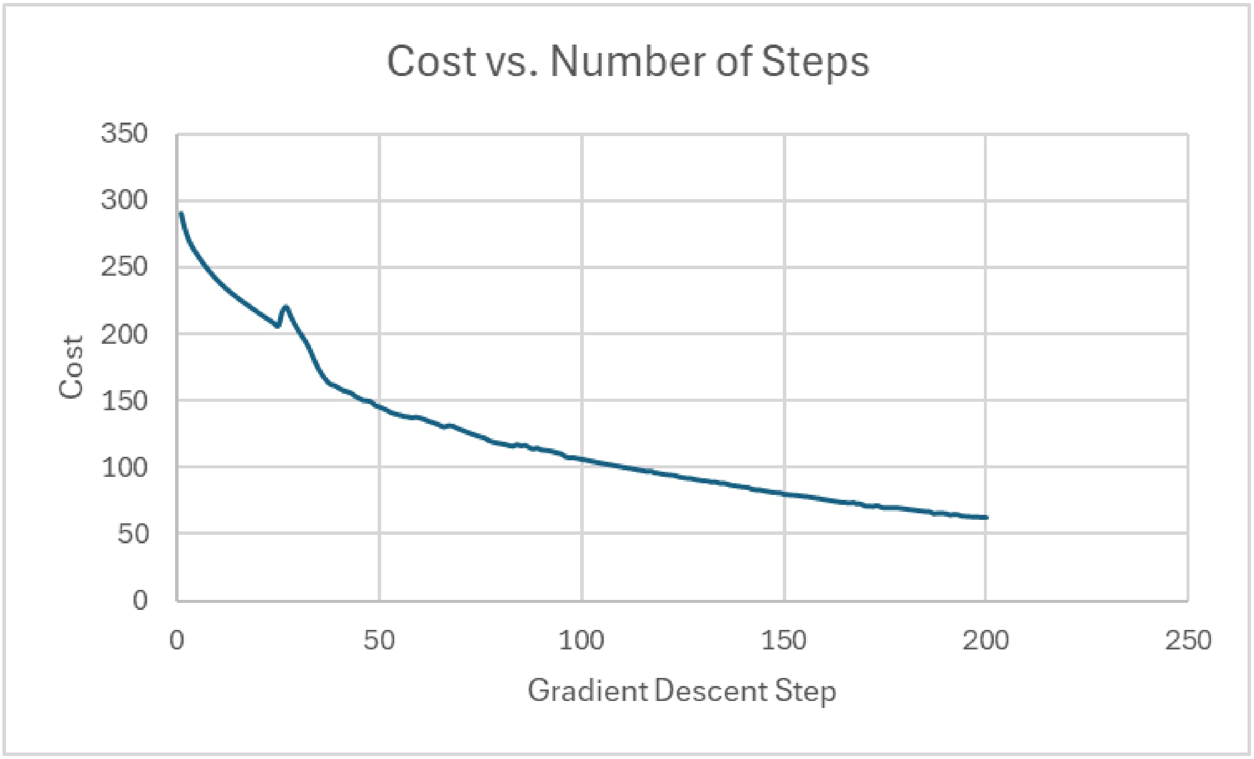


```
for j in range(len(trainValues)):

    gradients = findGradients(trainValues[j],inH_weights,inH_biases,HOut_weights,HOut_biases,trainLabels[j])
    #print((i+j))
    inH_weights = inH_weights   -.05*gradients[3]
    inH_biases = inH_biases     -.05*gradients[2]
    HOut_weights = HOut_weights -.05*gradients[1]
    HOut_biases = HOut_biases   -.05*gradients[0]
```

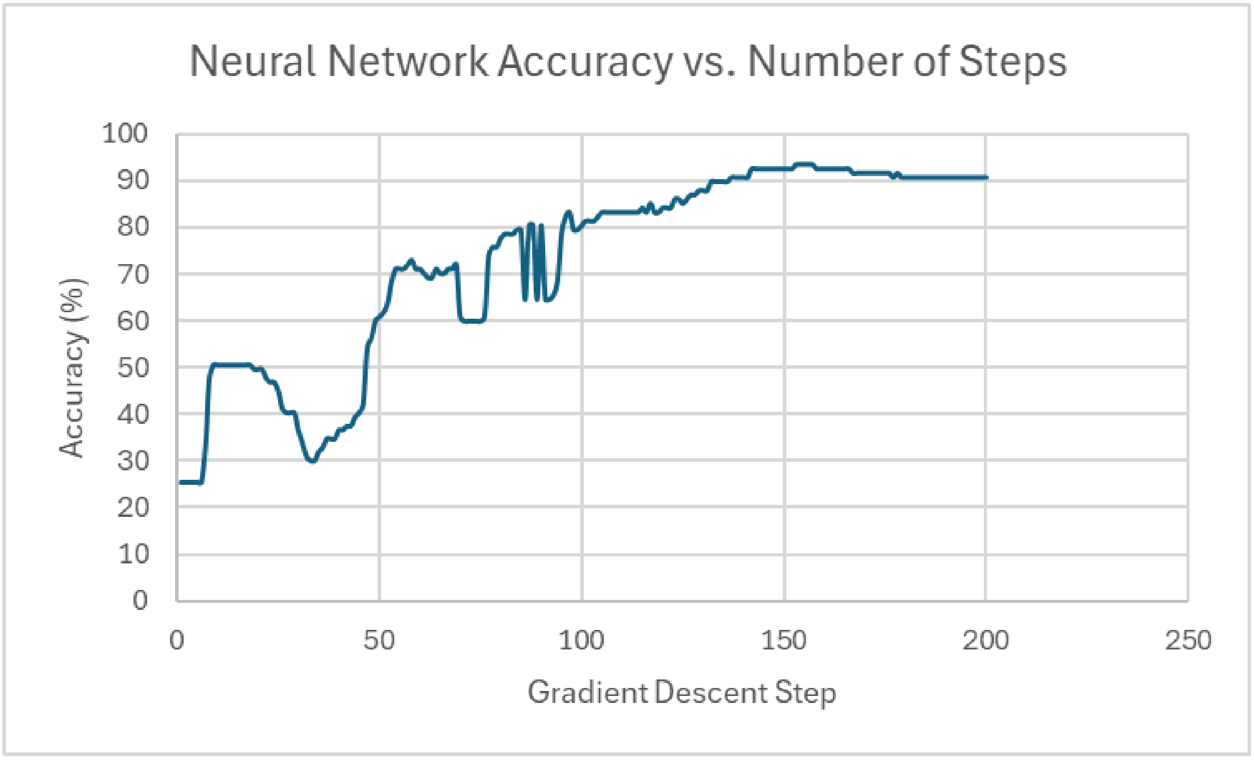
Here, the step size was decided to be 0.05 based on experimentation. A more analytical way to determine hyperparameters such as step size will be explored further in the results and discussion section.

Results

Overall, the method of gradient descent worked well. The graph below shows the cost of the neural network during the process of gradient descent.



After 200 gradient descent steps, the cost fell from 289.8 to 62.9, which shows that the gradient descent method was effective. However, the cost is somewhat of an arbitrary number that is relative to the size of the dataset, and doesn't directly tell us what we want to know, which is how accurate the neural network is at predicting hand gestures. To find the accuracy, a second dataset is then passed through the neural network. This 'testing' dataset of about 100 examples is completely separate from the training dataset, and the correct answer is not used to influence the output of the network. The accuracy of the neural network in regards to new data is shown in the graph below.

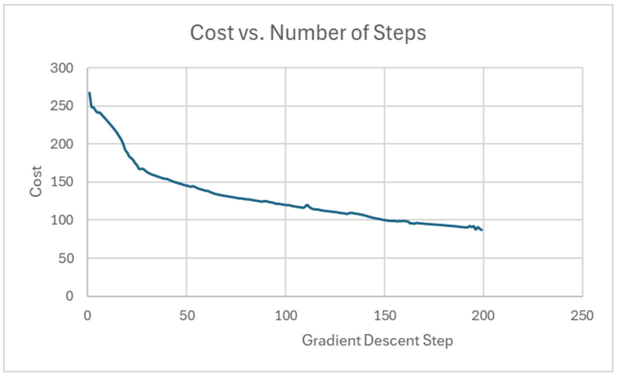


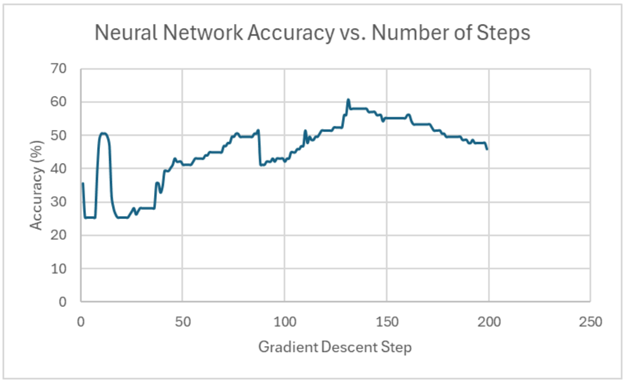
The accuracy increased over 200 gradient descent steps from 25.2% to 91.6%, which is in line with previous research in this field, albeit somewhat on the low side.

Discussion and Conclusion

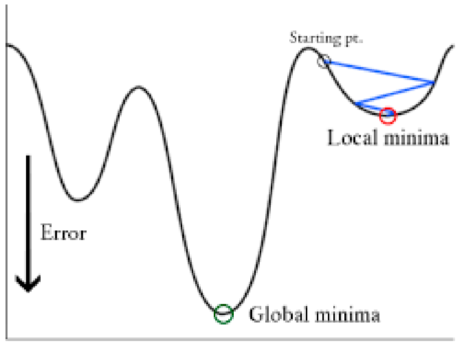
Several interesting observations can be made from the results. While the accuracy of the network increasing while the cost decreases is to be expected, the cost decreased in a mostly smooth manner, while the accuracy oscillates somewhat erratically while trending upwards. This could be due to several factors, including small training and testing datasets, or perhaps possibly due to the network overfitting in some instances.

Another observation during testing was that the results varied across different training runs. This is somewhat expected due to the parameters initially being randomized. However, the network in some cases was unable to improve accuracy in a meaningful way, as seen by the graphs below.

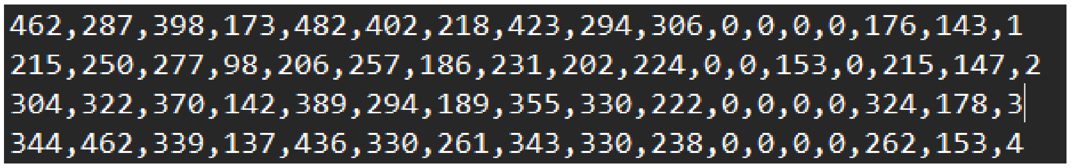




Here, the model plateaued at an accuracy of only 46%. This could be due to the model optimizing into a high local minimum, rather than a global minimum, which can be visualized by the image below.



A third observation can be made in regards to the actual parameters of the network. Neural networks are typically thought of as a 'black box' that are impossible to understand the inner workings of. This is especially true of deep learning models with dozens of layers and millions of parameters, but this model is relatively small, and interesting effects can be observed. From a small selection of input data (seen below), we can see that some sensors change more than others based on the grip pattern. Here, it can be noticed that the first sensor varies significantly, over 200 units between different grips, while the sixteenth sensor only varies a couple of units between grips.



```
462,287,398,173,482,402,218,423,294,306,0,0,0,0,176,143,1
215,250,277,98,206,257,186,231,202,224,0,0,153,0,215,147,2
304,322,370,142,389,294,189,355,330,222,0,0,0,0,324,178,3
344,462,339,137,436,330,261,343,330,238,0,0,0,0,262,153,4
```

After training the model, the difference in weight values of the two sensors was measured. The sum of changes to weights connected to the first sensor node was 39.2, while the sum of changes to the sixteenth sensor was only 20.6. If we observe individual weights, it becomes clear that the neural network 'decided' that the first sensor should be more influential than the sixteenth sensor. For example, the weight connecting the first sensor to the sixth hidden node changed from -.62 to 7.75. The corresponding weight for the sixteenth sensor only changed from 0.59 to 1.0.

While this paper demonstrated that a neural network could be used to accurately classify hand gestures, more research needs to be done to improve accuracy and make this solution commercially viable. Many of the ‘hyper’ parameters of the network such as number of layers, number of hidden nodes, activation function, and step size were chosen through simple trial and error, or randomly. These hyper parameters could be in themselves optimized, through methods such as genetic algorithms. Further optimization could also be done to the physical sensor arm band, which showed promising results for future research.

References

1. Pylatiuk C, Schulz S, Döderlein L. Results of an Internet survey of myoelectric prosthetic hand users. Prosthet Orthot Int. 2007 Dec;31(4):362-70. doi: 10.1080/03093640601061265. PMID: 18050007.\
2. Rehman, M. U., Shah, K., Haq, I. U., Iqbal, S., Ismail, M. A., & Selimefendigil, F. (2023). Assessment of Low-Density Force Myography Armband for Classification of Upper Limb Gestures. *Sensors*, *23*(5), 2716. https://doi.org/10.3390/s23052716
3. M. Anvaripour and M. Saif, "Hand gesture recognition using force myography of the forearm activities and optimized features," 2018 IEEE International Conference on Industrial Technology (ICIT), Lyon, France, 2018, pp. 187-192, doi: 10.1109/ICIT.2018.8352174. keywords: {Feature extraction;Sensors;Muscles;Optimization;Mathematical model;Gesture recognition;Support vector machines;Force Myography;Hand gestures;Feature extraction;Multiobjective optimization},
4. Kaifosh, P., Reardon, T.R. & CTRL-labs at Reality Labs. A generic non-invasive neuromotor interface for human-computer interaction. *Nature* 645, 702–711 (2025). https://doi.org/10.1038/s41586-025-09255-w
5. Brownlee, Jason. “A Gentle Introduction to Weight Constraints in Deep Learning.” *MachineLearningMastery.Com*, 6 Aug. 2019, machinelearningmastery.com/introduction-to-weight-constraints-to-reduce-generalization-error-in-deep-learning/.