Design of a Smart American Sign Language Translator

# Introduction

Physical disabilities can prevent the people from immersing themselves in everything society has to offer. Deafness in particular has a significant impact on a person’s life in that it prevents one from communicating through speech. Studies show that deaf person are less likely to make use of medical services and are more prone to psychological disorders – disorders stemming from the difficulty of forming meaningful connections with hearing counterparts [1], [2]. There have also been recorded instances of discrimination of the disabled in employment due to their physical disabilities, including deafness [3]. In the case of deafness, communication with others is the overarching issue, bringing with it a surge of inevitable side-effects. This paper seeks to develop a translation tool which relies on the training of a neural network for the purpose of recognizing different ASL (or American Sign Language) gestures.

The following project, tentatively called the “Sign AI,” seeks to design an ASL (American Sign Language) translation tool having deaf persons as the target demographic. The translation tool will take the form of a Windows-based GUI program with which users can interact for various purposes. A deep learning model will be trained to recognize ASL through pre-processed EMG data and orientation data in the form of quaternions calculated, all acquired from the Myo armband developed by Thalmic Labs. The final product is a gesture-based translator, providing translation for each gesture performed for a total of 100 gestures. In addition, the final product will have gesture-to-speech (text-to-speech) and speech-to-text capabilities. As stated before, Sign AI aims to satisfy the communication needs of deaf persons. Therefore, it will not only translate gestures, but also establish a method for allowing deaf users to understand spoken word.

Although design constraints are few in number due to the laissez-faire or “hands-off” nature of an AI project, undoubtedly, they do exist. One constraint lies in the ASL language itself, while others in the software used to develop the GUI. For one, some gestures are not arm-based or hand-based, which means that they cannot be reliably measured by the EMG sensors on the Myo armband. This is due to the linguistic nuances of ASL. For example, in ASL, pronouns are expressed through pointing a finger. Even with a sensing device, there is no way to determine if a gesture meant for a pronoun is “I,” “you,” “he,” “she,” or “they.” It is up to the signers to determine this for themselves. Since the neural network will be hosted locally, it must run on a device that possesses sufficient storage or RAM as well as processing power. The libraries required for running a neural network (such as Tensorflow and Keras) are large, reaching gigabytes. In addition, there are no pre-made datasets involving the Myo armband that can be used to train the neural network. Therefore, the dataset must be produced by me. Finally, as a consequence of software dependencies, wireless internet connection will be necessary to access Google’s text-to-speech and speech-to-text APIs.

# Background

In essence, the neural network is a mathematical model of the human brain. Every kind of neural network developed thus far has taken inspiration from studies of the natural world such as biology and chemistry. To understand the neural network, one must first understand the innerworkings of the human brain. The brain is the vital organ which oversees the body’s actors and actions. To be more specific, this biological machination takes in the stimulus it receives from the body’s sensory units, processes it, and prompts a response from the body. For example, when the human eye sees an image of a cat and a dog, it is able to pick out which image is depicting the cat and which the dog. The brain receives the images from sensing organs and interprets them through a series of chemical reactions. The key word here is “interpret.” Without the brain acting as an interpreter, any and all stimulus would lose their meaning. The images of the cat and the dog would simply come across as indistinguishable blotches of color on printed paper.

When discussing the origins of the neural network, one cannot overstate the influence of the biological studies from which inspiration for neural networks came. One such study was the 1962 study by Hubel and Wiesel. The Hubel and Wiesel experiment involved observations of the visual cortex. Using cats as a test group, Hubel and Wiesel determined that the neural response to a visual stimulus can vary depending on its presentation [4]. When the pair shined a light over an empty region of paper, the cats did not respond, and their neurons registered little to no activity. However, when the light was instead shined on a bunch of vertical lines or horizontal lines, the cats displayed significant brain activity. Specific sets of neurons were activated each time the light was shined on the lines. This study of the visual cortex served as one of many inspirations for the design of the ConvNet (or convolutional neural network), a neural network specializing in image processing.

Although the ConvNet will not see any use in the paper, the biological inspiration behind it does provide some light on the general mechanisms of a neural network. Similar to the brain, the neural network is nothing but an interconnection of neurons which respond in specific ways to specific stimulus (except, in the case of neural networks, neurons are called “nodes”). Unlike the brain, however, a neural network can only process numerical stimuli, packaged in array or vector form. Aside from the nodes, there are several other elements in a neural network which mimic elements in the brain. There are weights, biases, and activation functions – all of which determine the network’s response to incoming stimuli. The layout of a neural network can be broken down into large units of nodes called “layers.” In many software that allow for designing neural networks, this is how a neural network is defined (that is, by specifying a layer and the number of nodes in it). There are connection between each node, and values called “weights” modify the input going through each node. These weights help determine which parts of the input are important for producing an output. Training a neural network means finetuning these weights for better performance. Following the weights are activation functions, which are “bouncers” of a neural network. The activation functions determine which parts of the input proceeds to the other layers of the neural network. In the brain, weights are akin to the conductivity of the connective tissue in between neurons while the activation functions are akin to the rate of action potential.

# Project Approach

## Data Visualization & Acquisition

Naturally, the start of any deep learning project involves determining a means of acquiring reliable data. While the meaning of “reliable data” is different from project to project, its function remains the same for every project: to capture qualities of *something*. The job of data is to capture desired qualities while the job of the neural network is to make sense of them. Data collection was expected to be a major bottleneck in the development of Sign AI, as it is a time-consuming endeavor. The task of collecting data and the task of training the neural network cannot be pipelined because the neural network takes in the collected data as an input.

Once data is acquired, one must then design a neural network capable of processing the collected data. Architectures for neural networks are plenty, and some even specialize in solving niche problems. For example, there exists the recurrent neural network (or RNN, for short), which is frequently used for pattern recognition in text and speech. In solving image-based problems, the convolutional neural network was developed to fulfill the need. The design of the neural network should be congruent with the type(s) of data it will be handling.

In the case of Sign AI, the training data originates from the Myo armband, which, for all intents and purposes, is a glorified sensing device. The Myo armband acquires EMG readings from 8 different sensor modules and transmits it to the host computer via Bluetooth. The sensor modules are also equipped with an Invensense IMU (inertial measurement unit), which the Myo armband accounts for when calculating quaternions. The Myo armband outputs EMG readings for gauging muscle activation and quaternions for representing orientation in 3D space. For reference, the EMG readings produced by the Myo armband are pre-processed in that they are normalized to a range of values between -128 and 127. Since EMG can vary from person to person, the normalization of the raw EMG signals will help to center all readings to a common scale.

There are 2 methods through which a user can view data (EMG, orientation) from the Myo. The first method is through the Myo diagnostic page located at the following link: http://diagnostics.myo.com. The second is through the Myo Python library created by Nikolas Rosenstein. All documentation related to the Myo Python library can be found on GitHub and PyPi. As the Myo diagnostics page cannot be web-scraped, the second option was chosen instead. Thanks to the Bluetooth module and dongle, allowing Myo to communicate data to other devices is simple and hassle-free. The remaining hurdle is writing a script to continuously acquire said data.

The Myo Python library has a class known as the Listener class, which borrows from an already-existing class, the Device Listener from the Myo SDK (Standard Development Kit). The Myo SDK was put together by Thalmic Labs and was initially written in the C language. It was later adapted to the Python language by Nikolas Rosenstein in the form of the Myo Python library. The Myo Python library acquires data from the Myo armband through the hub instance. The hub instance establishes communication with the Myo armband until the program in which it was created is terminated. After the hub instance is created, one can now access the attributes of the Listener class such “orientation,” which provides a 4-element quaternion vector calculated from the Myo armband’s IMU), and “EMG,” which provides an 8-element vector containing EMG readings from the armband’s 8 different EMG sensor modules.

The data-logging will be implemented in Python through a step-by-step algorithm. First, all unique sensor readings will be documented in their own separate text files. Readings from the first EMG sensor module will be placed in a different text file than readings from the eighth EMG sensor module. Similarly, different elements (w, x, y, and z) in the quaternion vector will be stored in their own text files as well. 50 samples of each reading will be collected, as 50 samples has proved to be sufficient in capturing the entirety of many gestures. Finally, an indicator that signifies which group of readings were taken from the same instance was implemented in the form of an integer number that is stored in a text file and continuously checked by the program. Acting as a label, this number will be appended to the names of all text files created in a given instance. The number will be used to group them up when it comes time to format all the files for use in the neural network.

Keep in mind that, for a neural network to be able to process the data given to it, said data must be in the form of an array, the dimension of which depends on the type of layers used in the network. First, the dataset was split up into two groups: one dataset (called the training dataset) will used for training the neural network while the other dataset (called the validation dataset) will be used to test the neural network after during each training cycle. The neural network will never see the solutions for the validation dataset while the opposite is true for the training dataset.

To transform the text files containing the sensor readings, several steps must be observed. Obviously, since all text files are expressed in string or character form, they must be type-casted into their numeric counterparts. This can be easily done in Python. Additionally, since the type of data acquired from the Myo armband is time-series data, the way in which said data is packaged should reflect this as well. Fortunately, arrays are automatically suited for this, as its indices of each of the elements contained in them can substitute for time. One can assume that the indices represent the time difference between the readings. This is not a problem since, like the indices which are spaced out by a constant of 1. EMG is also sampled at a constant 200 Hz by the Myo armband.

The key to formatting the sensor data lies in stacking the readings element-wise and storing them in a master file where the neural network can process them all at once. The NumPy library will be used to perform the stacking. Stacking the readings element-wise involves taking an array containing sensor readings from the same instances and storing said array as an element. Every interval of time, the Myo armband acquires many different readings. To represent them properly, these readings must all be stored as a single element. For example, since 50 data points were collected from the Myo armband from different sources of data, each data point must correspond with each other in accordance to their order. For example, the first data point from one source must correspond with the first datapoint taken from another source. A label is appended to the stacked array once all readings are processed. This label represents the gesture performed when the now-stacked readings were taken. All stacked arrays will be stored in a pickle file for ease of access. The benefit of storing the arrays in a pickle file is that they do not lose their format. If the arrays were to be stored in a text file, they would be turned into strings. Accessing them again would require a conversion back into array form, which overcomplicates things. Storage in a pickle file is beneficial in that it maintains a content’s integrity and simplifies access.

## The LSTM Model

The LSTM model falls under the umbrella of the recurrent neural network (RNN). For context, the RNN is a type of neural network that excels in processing sequential data. It is able to do this through transmitting prior information to later parts of a neural network, essentially forming a memory. Most neural networks are feed-forward networks, meaning that information flows in a single direction. Every layer in a feed-forward neural network produces outputs that other layers are not able to access. The other layers are not told how these outputs are produced. This is where the RNN begins to diverge from the traditional feed-forward neural network. The RNN interconnects outputs of layers by passing said outputs from one layer to the next through a loop. Prior information is known as the “hidden state” in the realm of deep learning. The hidden state relays more than just the outputs of a previous layer. It also describes how previous outputs have changed over time while in previous layers. RNNs have layers with feedback loops, and this is where the hidden state is passed on and updated. In an RNN neural network, the hidden state of a previous RNN layer can affect the outputs of the RNN following layers because it is processed in conjunction with the actual inputs.

The specialty of the LSTM model is memorizing long-term dependencies or patterns in data. There are mechanisms within an LSTM layer that allows it to keep track of dependencies. In an LSTM layer, two elements are responsible for memorization: the cell state and the gates. For reference, the terms “cell state” and “hidden state” can be used interchangeably. The cell state is an additional output produced by an LSTM layer. It is where changes in the input are documented. The cell state and gates go hand-in-hand in that the gates are responsible for updating the cell state. In short, the “gates” are activation functions which have a binary range, meaning their outputs lie between 0 and 1. The activation functions determine if the change to the cell state is significant.

Is the LSTM model suitable for gesture classification? The answer is a resounding “yes.” When one is considering the architecture of a neural network, one must also consider the data that will be processed by said architecture. There is a correspondence between the architecture and the processed data. The processed data will relay important information to the network, while the network interprets relayed data and outputs a value that represents it. The factor that determines if a network is suitable for solving a problem is its ability to interpret the input data. What information is hidden in the data? Are there patterns or dependencies? What is the size of the data? Does the data vary with time? Is the proposed network capable of understanding subtext in the data? The following questions bring up some important considerations. In all considerations, the bottom line is that the neural network must be able to “read between the lines” and decipher desired chunks of information.

What makes the LSTM model suitable for gesture classification? The answer lies in its ability to memorize long-term patterns and dependencies. Recall that the LSTM model contains layers which update themselves based on the changes that the input data undergoes in the network. These “updates” are representative of the patterns and dependencies that the network has extracted after processing the input data. The input data will come in the form of readings from EMG sensors and a 9-axis IMU. These readings vary with time and correspond to a gesture. The job of the neural network is to determine the performed ASL gesture based on the aforementioned readings. To be successful, it will need to keep track of changes in the sensor readings. In other words, it will need to memorize patterns and dependencies. This need is fully met by the LSTM model, due in large part to its design and intended function.

In the paper “How to Construct Deep Recurrent Neural Network” written by Bengio et al., much discussion was devoted to depth of RNNs and RNN variants like the LSTM (Long Short-Term Memory) RNN. Bengio et al. state that, for feed-forward neural network, a deeper model often leads to a “more expressive model,” meaning that the model is able to understand data at a higher level [5]. They suggest that the same proposition could lead the same outcomes for RNNs. Their discussion referenced another paper written by Hermans and Schrauwen. This paper is essential in understanding the real impact of stacking recurrent layers.

In the Hermans and Schrauwen paper (titled “Training and Analyzing Deep Recurrent Neural Networks”), some background was explored regarding the default depth of a single layer in an RNN: “when folded out in time, it [RNN] can be considered as a DNN with indefinitely many layers” [6]. A DNN is an abbreviation used to describe deep neural networks. Hermans and Schrauwen claim that an RNN is, in their opinion, a DNN that possesses infinite layers. The “infinite layers” commentary refers to the basic structure of an RNN. A layer of an RNN has an internal memory that enables it to process data sequences. This internal memory is known as the “cell state.” Variants of the RNN like the LSTM RNN also have cell states. The cell state is updated through feedback loops in an RNN that propagates new and old information. If the input data is infinitely long, then data is infinitely propagated through a layer, making the RNN infinitely deep. Hermans and Schrauwen were alluding to the persistent nature of the RNN by saying it is “infinitely deep.”

Later in the paper, Hermans and Schrauwen delve into the design of an RNN, particularly the logic behind it: “For RNNs, the primary function of the layers is to introduce memory, not hierarchical processing.” Hierarchical processing in neural networks is the idea that new knowledge is extracted at each layer. Hermans and Schrauwen believe that hierarchical processing is not the aim of an RNN. They also believe that shallow hierarchical processing can limit an RNN’s performance, even listing several performance-affecting limitations:

“One potential weakness of a common RNN is that we may need complex, hierarchical processing of the current network input, but this information only passes through one layer of processing before going to the output… Secondly, we may need to process the time series at several time scales. Common RNNs do not explicitly support multiple time scales, and any temporal hierarchy that is present in the input signal needs to be embedded implicitly in the network dynamics” [6].

Hermans and Schrauwen are making the following assertions: that the RNN does not excel in hierarchical processing, which can affect its understanding of the input, and that the RNN is not by default capable of analyzing data piece-wise with respect to time. To address these concerns, the following proposition was made and later verified:

“The architecture we study in this paper is essentially a common DNN (a multilayer perceptron) with temporal feedback loops in each layer, which we call a deep recurrent neural network (DRNN). Each network update, new information travels up the hierarchy, and temporal context is added in each layer. This basically combines the concept of DNNs with RNNs. Each layer in the hierarchy is a recurrent neural network, and each subsequent layer receives the hidden state of the previous layer as input time series. As we will show, stacking RNNs automatically creates different time scales at different levels, and therefore a temporal hierarchy” [6].

Essentially, Hermans and Schrauwen are proposing a fusion of the DNN and RNN. Dense layers will be used to increase hierarchical processing, while RNNs will be stacked for the purpose of giving temporal context to the input data at different instances of time.

Due to the benefits of increased hierarchical processing and temporal context, the paper will be adopting Hermans and Schrauwen’s architecture. To recap, the Myo armband will be providing multiple sensor readings to the neural network, some different types. Specifically, EMG and orientation data will be used by the network to classify ASL gestures. If the network lacks hierarchical processing, it will not be able to understand the spatial context between the readings. Here, “spatial context” refers to the knowledge of which sensor produces which reading. In addition, without the memory of an RNN layer, temporal context will be lost. This means that the network will be unable to recognized time-based patterns. Hermans and Schrauwen’s architecture combines the hierarchical processing of DNNs and the temporal context from RNNs to process data from multiple perspectives.

## Regularization Methods

### Dropout

By all accounts, a stacked LSTM model is a large neural network. This can be problematic if the dataset used for training is small. Having a large neural network but small dataset is one indicator of overfitting. The Sign AI project has 100 labels, with each label corresponding to an ASL gesture. Each label contains 100 samples, resulting in 10,000 overall samples. The label-to-sample ratio is 1:100. While there is no standard that defines a “small” or “large” dataset, it is preferred to have more samples. More samples suggest that the dataset is diversified, covering data containing differentiations variations with which to train the network. The dropout layer is one age-old defense against overfitting. Its merits are discussed in depth in “Dropout: A Simple Way to Prevent Neural Networks from Overfitting” written by Hinton et al.

Deep neural networks are beneficial in that they are able to learn complicated relationships. This is corroborated by Hinton et al., who state that having multiple layers help the network recognize underlying connections between inputs and outputs [7]. In addition to its architecture, a neural network also relies on training data, which it dissects for hidden information. Ideal circumstances mean that training data is abundance. However, this is not always the case. A limited dataset can cause a neural network to overfit without proper regularization, which is a means of increasing a network’s ability to “generalize.” If a network is capable of “generalizing,” then it is able to handle data that it has not been trained with. A network not capable of generalizing is considered to be “overfit.” Training a neural network involves micro-managing weights and biases, both of which are parameters that dictate the network’s response to inputs. At every epoch or training cycle, weights and biases are constantly overwritten in an attempt to find the most ideal values. There are many reasons behind an overfit model. One obvious culprit is the number of epochs. If the neural network is exposed to the training data for a long time, it will start to develop weights which respond well to the training data but fail with other data. In other words, the neural network will perform poorly when processing data it has not been trained with.

An often-used form of regularization, the dropout method is a method which randomly disables nodes in a layer (or layers) of a neural network. In the Keras API, one can specify the percentage of nodes to be disabled by passing it as an argument to the dropout layer. As discussed in Hinton et al.’s paper, there is a biological basis behind the dropout method, one rooted in the topic of genetics and the passing of genes.

Hinton et al. explains the role of dropout in a neural network through the concept of natural selection. The key words are “reduction” and “co-adaptations.” In biology, evolution forces individuals to develop new genes. These new genes are supposed to help humans develop advantageous characteristics. The end goal is to develop new *and* advantageous genes. This is done through minimizing interplay between genes, known as biological co-adaptations. Co-adaptation is a phenomenon in which a person’s characteristics are determined by gene pair. Each gene relies on its counterpart to remain functional. These dependencies between genes are not desirable as they create multiple points of failure. Co-adaptation also exist in neural networks in the form of dependencies between layers. Co-adaptations between layers in a neural network can be minimized through dropout. Each layer feeds information into future layers, which can create unwanted or unintended dependencies between layers and cause the neural network to overfit. One way to reduce these dependencies is by preventing small chunks of information from reaching future layers, which the dropout method accomplishes through shutting down nodes. The logic behind the dropout layer is to force layers to be more self-reliant and infer the missing information themselves [7].

### Batch Normalization

In the context of neural networks, batch normalization is a method of standardizing inputs. Batch normalization addresses several problems related to the performance and training of a neural network. A paper written by Sergey Ioffe and Christian Szegedy (titled “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”) discusses the merits of batch normalization in explicit detail. The internal covariate shift is touted by Ioffe and Szegedy as something that is negative, having an adverse effect on a neural network. Ioffe and Szegedy justify their claim by bringing up the distribution of inputs in layers: “The change in the distributions of layers’ inputs presents a problem because the layers need to continuously adapt to the new distribution. When the input distribution to a learning system changes, it is said to experience covariate shift” [8].

Before one can understand the impact of batch normalization on a neural network, one must first understand how its weights are updated. In a neural network, data flows forwards and backwards. The backwards flow of data is called “backpropagation.” The job of a neural network is to map inputs to appropriate outputs. During training, outputs is pre-determined. The important parameter is the network’s prediction. Weights are updated depending on the exactness of the prediction. How is this “exactness” quantified? The answer lies in the concept of error. Error needs to be calculated for every node in a network and sent back for processing. This “sending back” is backpropagation at work. It is an algorithm which calculates the gradient or partial derivative of the cost function with respect to the network’s weights. This gradient represents the error of each weight in the neural network.

Several intricacies still need to be addressed before the effect of batch normalization can be fully explained. First, what is the cost function? Second, how is the cost function calculated? The cost function is a function which describes the range of performance of a neural network based on the number of training samples, number of layers, and the output of every node. The desire of every network is to reach the global minimum of a loss function, as this would mean that the network is performing at its best. Now, the question needs to be asked: how is backpropagation used to update weights? To reiterate, error is the parameter that flows backwards through the network. Every node has an error value associated with it, which will used to determine if the network is navigating the cost function towards the global minimum, the ideal case. New costs are compared with old costs, and, if the new costs are lesser, then the neural network updates itself by overwriting previous weights with the weights determined in the most recent epoch or training cycle.

How does batch normalization benefit the neural network? Batch normalization is, in essence, a rescaling of the weights to a localized range. Ioffe and Szegedy mention in their paper a phenomenon called “internal covariance shift,” explaining that the purpose of batch normalization is to minimize the influence of such a shift [8]. First, what is “internal covariance shift”? Ioffe and Szegedy explain it as the variations in the outputs of each node in a layer, which are received as inputs by other nodes from another layer. These variations are thought to worsen as parameters such as weights and biases are updated for every layer in a network. The particular term Ioffe and Szegedy use for the outputs of the nodes is “hidden unit values.” In addition, rather than using the word “variation,” the more accurate word would be “distribution.” As weights and biases of previous layer change, so do the distribution of inputs that enter the subsequent layers in a neural network. Changes in input distributions can cause the network’s performance to shift wildly, as the network has to continuously accommodate for new input distributions. With Batch Normalization, the stability of input distributions are increased as all hidden unit values in the affected layers are made to have zero mean and unit variance. To be more specific, the hidden unit values are forced to conform to a standard normal distribution all throughout the neural network.

There have been papers written that conflict with Ioffe and Szegedy’s ideas, however. One paper written by MIT PhD students argue with proof that the purpose of batch normalization is not to reduce the internal covariance shift of a neural network in any way. In fact, an experiment was conducted in the paper that proves this is not the case. One of the primary objectives of the paper was to verify the connection between reduction in the ICS (abbreviation for “internal covariate shift”) and batch normalization. The experiment conducted by the paper involved training networks where noise is injected to disturb input distributions. The reason for injecting noise is rather clear from the authors’ perspective: to create the effect of skewed input distributions. With the implementation of batch normalization layers in their network, the authors (Ilyas et al.) sought to prove or disprove the supposed “power” of batch normalization to fix the uneven distributions and thereby reduce the ICS (which is expected to increase the network’s performance as a result). Ilyas et al. discovered that, while the normalized network performed better than the non-normalized networks, it was not due to reduction in the ICS that this was so. In fact, the batch normalization layers did little in addressing input distributions:

“Observe that the performance difference between models with BatchNorm layers, and ‘noisy’ BatchNorm layers is almost non-existent. Also, both these networks perform much better than standard networks. Moreover, the ‘noisy’ BatchNorm network has qualitatively less stable distributions than even the standard, non-BatchNorm network, yet it still performs better in terms of training” [9].

Ilyas et al. state that the reason why batch normalization is so effective is because of its reparameterization of “the underlying optimization problem” being solved by the neural network during training [9]. One might ask: what is the optimization problem that the neural network is solving? The answer lies in the gradient of the cost function. Recall that the goal of training a neural network is to reduce error in predictions, which means navigating the cost function towards its global minima. How does the neural network determine that the error has been reduced? It does so by looking at the gradient, which is an expression for the rate of change of something. After all, a gradient is simply a partial derivative. The gradient tells the neural network which direction it is traveling the cost function. Since Ilyas et al. have stated that batch normalization makes the gradient “more predictive and well-behaved,” the network will display more stable performance during training as it knows with more certainty where it is going. “Well-behaved” in this case reflects the gradient’s “Lipschitzness,” as mentioned by Ilyas et al. The mathematical concept of Lipschitzness deals with gauging a function’s continuity, the criteria used by Ilyas et al. to measure “smoothness.” More continuous functions tend to have less regions where the change from one value to another value is too significant and abrupt. A continuous function looks more connected, requires less interpolation between points, and is therefore more “smooth.” In short, smoother gradients are beneficial because of their continuity.

## GUI Development

When developing a graphical user interface (or GUI, for short), both form and function go hand-in-hand. In terms of form, a GUI should be easy-to-navigate; in terms of function, it should require as little heavy lifting or accommodation from the user as possible. Firstly, a design of a GUI should not be littered with features. There is such a thing as sensory overload, after all. Having more functions usually translates to having more UI elements. This can end up cluttering the user’s screen and make going through the GUI troublesome. Although a GUI that has countless functions sounds like an ideal to strive for, its layout will take careful planning. Secondly, the GUI should be targeted to fulfill a specific purpose. What is the GUI supposed to allow the user to do? In the case of Sign AI, the purpose of the GUI is to acquire and display gesture information. The GUI will package the neural network and the communication features (speech-to-text and text-to-speech) into one. It also will lay the groundwork for establishing the means of receiving and processing data from the end user.

It was decided that the GUI for Sign AI will be simple and fulfill 3 basic functions: acquiring sensor data from the Myo armband, sending sensor data to neural network for processing, and acquiring speech data to be converted to text (and vice-versa) by Google APIs. Since the GUI will be written using the Python language, the Tkinter library will be used to develop it. The Tkinter library compiles and adapts the Tk GIU toolkit written in its own language to Python. One main benefit of using Tkinter for GUI development is that it is a cross-platform library, meaning that GUIs spawned from it can be run on Microsoft, Linux, and Apple operating systems. It is also equipped with all the traditional UI elements found in most GUIs such as buttons, drop-downs, and scrolled lists.

Typically, buttons are used for executing something at the behest of the user. If the user clicks a button, the user wants to run the function that said button performs. Buttons will be implemented to give the user a choice to perform gestures or to listen to speech from another party. There will be 2 buttons to encompass these functions, and the buttons will be aptly labeled the “Perform Gestures” button and “Listen to Others” button, respectively. The “Perform Gestures” button talks to the neural network and fires up Google’s Text-to-Speech API once a translation is received. It enables the user to perform as many gestures as possible until the “Stop” button is pressed. The users need to know *when* the GUI is doing *what*, so a status message field was created which describes the GUI’s current action. If the GUI is interfacing with the neural network, the status message field will say so.

In Python, making GUIs that call long-running processes through button clicks is not as straightforward as it seems. This is because Tkinter GUIs are not capable of performing asynchronous tasks without outside help. The Tkinter GUI updates itself as well as executes all other user-defined functions or methods in a single thread. The GUI updating itself is last in the pecking order. To make matters worse, if the GUI does not receive the request to update itself within a short period of time, it will become unresponsive, which will prompt the operating system to shut it down and diagnose it. If you call a long-running process, the program will inevitably branch from the main loop where the GUI updates itself to the long-running process that was just called. If your program branches from the main loop for too long, the GUI glitches out and turns unresponsive to any stimulus (e.g. button presses). Although there is a chance for the GUI to become responsive again, letting it become unresponsive in the first place is a sign of bad GUI programming.

To allow my GUI to call a long-running process, I decided to use the multiprocessing library in conjunction with implementing a queue where outputs from long-running processes are polled periodically. It should be noted that the queue is a FIFO (first in, first out) structure, the opposite of a stack, which is a LIFO (last in, first out) structure. This means that its contents are fetched in the order they were put in. The multiprocessing library will allow my GUI to bypass Python’s global interpreter lock (GIL) and spawn concurrent processes. Recall that the GIL prevents Python from running multiple threads at once. This is effectively dealt with by spawning sub-processes instead. The multiprocessing library is compatible with Windows and Unix systems, but not Apple systems. The queue (which was imported from the Queue library) is made to be a global variable, meaning that it can be accessed and modified by anything in the code space. Methods were written that allow the GUI to put content and retrieve content from the queue. On button click, the multiprocessing library spawns the desired subprocesses. At millisecond intervals, the program checks is the subprocess is still executing. If the subprocess is indeed still executing, the program is instructed to wait until it finishes execution, allowing the GUI to update UI elements without interruption. Once executed, the output of the subprocess is appended to the queue, where it can be fetched. The reason behind the success of this implementation lies in the fact that the GUI is not running the subprocess inside the loop where it is supposed to update UI elements. In this implementation, the only thing that the GUI is responsible for is fetching the output of the subprocesses from the queue. This fetching process takes but a fraction of a second, which means it does not disturb the flow of the main loop and will not render the GUI unresponsive.

# Results and Discussion

## Evaluating Neural Network’s Performance

The final version of the neural network consisted of two stacked LSTM layers containing 128 nodes and a dense layer with 100 nodes for the output layer. Recurrent dropout was applied to the LSTM layers. A batch normalization layer followed by a dropout layer was placed after the stacked LSTM layers. The decision to stack 2 LSTM layers was discovered through trial-and-error. Stacking 3 or more LSTMs generally led to worse performance, while not stacking at all made the network too shallow. The best performance was observed when 2 LSTMs were stacked.

When evaluating the performance of a neural network, one must examine 2 parameters: loss and accuracy. The loss of a neural network is a number indicative of the network’s tendency to make prediction mistakes, while its accuracy describes in percentage form its success in making predictions. There is a relationship between loss and accuracy that can tip off signs of overfitting. Recall that there are 2 datasets that the neural network must be able to solve: the training dataset and the validation dataset. The neural network sees the solutions to the training dataset, so it is able to compare its prediction to said solutions. However, the solutions to the validation dataset are not visible to the neural network, which means that the loss and accuracy of the validation dataset are more telling of the network’s true performance. For reference, the training dataset is 8000 samples large while the validation set is 2000 samples large. When there are significant differences between the training outcomes and the validation outcomes, this should raise flags that the neural network is overfitting.

Thusly, several methods of regularization were used to prevent the neural network from overfitting. Among these methods, dropout and weight constraints were used. Dropout regularization was employed through adding dropout layers in the neural network, while weights constraints required the use of the max norm function from the Keras library. As discussed before, the benefit of dropout lies in the ridding of dependencies between layers and forcing layers to “think for themselves.” This is done through disabling nodes in a layer, ensuring that not all information will flow to the subsequent layers. On the other hand, max norm regularization involves ensuring that weights fall within a specified range at all times. Keras checks each weight and rescales the weights which exceed the constraint put on them. This combats the problem of exploding gradients.

Keras offers many functions which assist in the creation of better-performing neural networks. One of these functions is the “early stopping” function, which, as its name suggests, cuts short the training of a neural network once it has stopped showing signs of improvement. Early stopping takes in similar arguments as the “reduce LR on plateau” function; it has a “monitor” argument and “patience” argument. Depending on the user’s specifications, the training of the neural network can be long or short. In addition to the “early stopping” function, the “save best model” function was used to capture improvements of the neural network. Simply put, whenever the neural network exhibits improvement in the monitored metric, its current state is saved as a file that can be loaded in the future. This file contains all the information necessary (architecture, weights, etc.) to reconstruct the neural network.

A famous gradient descent optimizer was used to improve the training process: the “Nadam” optimizer. “Nadam” is shorthand for the Nesterov-accelerated version of the adaptive moment estimation algorithm. Recall that the job of the gradient is to let the neural know where and how it is navigating the loss function. Optimizers like Nadam aid in this endeavor by affecting the updates made to the weights of a neural network. Like the Adam (adaptive moment estimation) optimizer from which it is an offshoot, the Nadam optimizer combines the concepts of momentum and adaptive learning rates to achieve better performance and speed.

To understand the concepts of momentum and adaptive learning rates, the basics of stochastic gradient descent – yet another popular optimization method – must first be explained. As its name suggests, the calculation of the gradient (of the cost function, with respect to each weight) plays a major part in stochastic gradient descent. SGD (stochastic gradient descent) uses the computed gradient as a compass to tell it where to go. For example, let us assume that the loss function is described by a simple quadratic . The derivative of such a function is . This derivative can be used to determine the behavior of the loss function. If , the derivative will resolve to 4. Since the derivative resolves to 4 at , this means that the gradient moving in a direction that will take us away from the global minimum, which is . Such a thing is obviously undesirable, so SGD will force the loss function to move elsewhere by updating the weights and biases accordingly. Although the example is rather simple, it describes in simple terms what SGD seeks to accomplish: to re-calibrate weights and biases based on the gradient of the loss function. SGD is rather slow to arrive at ideal solutions, so momentum was developed for SGD to speed it up as well as make the gradients more useful. Momentum applies the concept of moving averages to the gradients. Instead of computing the gradient per batch of data and re-calibrating based just on said gradient, momentum forces SGD to keep a moving average of all the calculated gradients. This averaged gradient intuitively leads to better updates, as the neural network has a better idea on how it should move.

Compared to momentum, adaptive learning rates are not as hard to get a grasp of. The Nadam optimizer uses the mean and variance of the gradient to compute learning rates for each weight in the neural network, supposedly helping the network to reach ideal weights. Here, moving averages of the gradient are used for the computation. In Nadam optimization, the name of the game is weight decay, where updates made to weights are affected through manipulating the learning rate. Nadam optimization begins with a sizable learning rate, which only get smaller and smaller as the performance of the neural network improves. The intuition behind this is similar to shooting a basketball: the closer you are to the net, the less force you have to exert in order to land the ball in the net. In this case, the net represents the ideal weights for the neural network, while the force required to shoot the ball is the learning rate.

A mixture of regularization methods as well as help from the optimizer and Keras helped the neural network reach a performance boasting an accuracy and loss of 0.9455 and 0.1819 in the training dataset. In the validation dataset, a comparable performance was reached, with the accuracy and loss plateauing at 0.9405 and 0.2766. Thanks to the Nadam optimizer, an exponential decay in loss and a logarithmic growth in accuracy can be observed as training progressed, indicating the speed at which the network learned. The best weights were reached at epoch 32 and were saved by Keras. Early stopping maintained the training process until epoch 40, where the neural network displayed prolonged periods of stagnation and was promptly prevented from training further.

## Evaluating Real-Time Performance

Once the neural network was trained and the GUI put together, real-time translation work was done to gauge the accuracy of the final product. Table 1 splits up all 100 gestures in 3 categories: first 3 attempts, between 5 to 10 attempts, and beyond 10 attempts. Each category describes the number of attempts that was needed to get a correct translation of a gesture. Gestures belonging in the first category were gestures which the neural network was easily able to recognize. In fact, most if not all the gestures in said category were recognized on first attempt. Meanwhile, gestures belonging in the second category were gestures which the neural network was capable of recognizing, but with a little more difficulty. Detection of these gestures relied on accurately performing the versions of the gestures that the neural network was trained with. Putting personal variations on these gestures – no matter how slight – can cause occasional mispredictions. On the other hand, gestures belonging in the final category were gestures which the neural network consistently mis-predicted. Many of these gestures overlap with other gestures in terms of the types of motion performed, which make them tricky for the neural network to recognize.

Table 1. Real-Time Testing: Assessment of Performance through Measuring Number of Translation Attempts

|  |  |  |
| --- | --- | --- |
| **First 3 Attempts** | **Between 5 to 10 Attempts** | **Beyond 10 Attempts** |
| Sister, brother, aunt, uncle, separate, day, hot, cold, hamburger, egg, fork, hungry, socks, shoes, coat, hurt, bathroom, sleep, sad, sorry, bad, excuse, help, who, where, big, tall, full, more, blue, yellow, red, brown, orange, cost, bird, horse, sheep, pig, bug | Gold, silver, drink, spoon, angry, single, home, work, school, store, church, come, go, in, out, with, night, week, year, today, finish, hotdog, apple, cheese, cup, cereal, water, candy, cookie, cat | Mom, dad, marriage, grandma, grandpa, baby, pants, underwear, brush, nice, cry, like, good, love, please, thank you, what, when, why, how, stop, green, dollars, dog, cow, children, future, here, milk |

There are potential reasons why the real-time performance of the translator pales in comparison to the performance projected in Keras. One main reason lies in the inevitable variations in EMG readings. In theory, EMG readings can be used to measure the activation of muscle from an electrical domain. However, the reality is that, despite the normalization done by the Myo armband on the raw EMG signals, there are certain factors that can without fail throw off one’s EMG readings, especially in non-invasive EMG like surface EMG. Raez, Hussain, and Mohd-Yasin state that a person’s physical makeup can very well influence their electrical state and affect their EMG readings: “Anatomical, biochemical and physiological factors take place due to the number of muscle fibers per unit, depth and location of active fibers, and amount of tissue. The amount of the tissue between contracting muscles and electrodes, along with their thickness, affect the amplitude of the EMG signal” [10]. In addition, muscular diseases can also lead to abnormal EMG readings, as can slight changes in skin conductance (which can be caused even by changes in one’s mood) [11], [12]. The point is that there is no surefire catch-all when it comes to EMG. Such is likely the reason why there is a performance gap between the performance projected by Keras and real-time performance.

# Acknowledgements

I would like to thank Dr. Jason Brownlee for his tutorials on machine learning and deep learning algorithms. In addition, I would also like to thank my mentor Dr. Lei Zhang and the UMES Department of Business and Technology for providing me with the materials necessary for me to finish my project.

# References

[1] J. Fileccia, “Sensitive care for the deaf: a cultural challenge,” *Creat. Nurs.*, vol. 17, no. 4, pp. 174–179, 2011.

[2] F. G. Bowe, B. T. McMahon, T. Chang, and I. Louvi, “Workplace discrimination, deafness and hearing impairment: The national EEOC ADA research project,” *Work*, vol. 25, no. 1, pp. 19–25, 2005.

[3] C. Lou Garberoglio, S. Cawthon, and M. Bond, “Deaf People and Employment in the United States: 2016,” 2016.

[4] D. H. Hubel and T. N. Wiesel, “Shape and arrangement of columns in cat’s striate cortex,” *J. Physiol.*, vol. 165, no. 3, p. 559, 1963.

[5] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, “How to construct deep recurrent neural networks,” *arXiv Prepr. arXiv1312.6026*, 2013.

[6] M. Hermans and B. Schrauwen, “Training and Analysing Deep Recurrent Neural Networks,” in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 190–198.

[7] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[8] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *arXiv Prepr. arXiv1502.03167*, 2015.

[9] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, “How does batch normalization help optimization?,” in *Advances in Neural Information Processing Systems*, 2018, pp. 2483–2493.

[10] R. H. Chowdhury, M. B. I. Reaz, M. A. B. M. Ali, A. A. A. Bakar, K. Chellappan, and T. G. Chang, “Surface electromyography signal processing and classification techniques,” *Sensors (Basel).*, vol. 13, no. 9, pp. 12431–12466, Sep. 2013, doi: 10.3390/s130912431.

[11] C. Larivière, D. Gagnon, and P. Loisel, “The comparison of trunk muscles EMG activation between subjects with and without chronic low back pain during flexion–extension and lateral bending tasks,” *J. Electromyogr. Kinesiol.*, vol. 10, no. 2, pp. 79–91, 2000.

[12] E. Gatti, E. Calzolari, E. Maggioni, and M. Obrist, “Emotional ratings and skin conductance response to visual, auditory and haptic stimuli,” *Sci. data*, vol. 5, p. 180120, 2018.