University of Arkansas – CSCE Department
Capstone II – Final Report – Spring 2021

Sign Language Interpreter

David Clairmont, Johnny Doan, Jack Gaither, Nick Hester, Sam Witucki

Abstract

The problem that we have discovered is that learning American Sign Language (ASL) is difficult, as it is with learning any language. Many different platforms help with teaching ASL, but through experience, online sign language teachers lack the ability to keep the student fully engaged and interested in the subject. Our objective is to develop a sign language teacher to improve the learning process by actually engaging with the student, with hopes of maintaining the desire to learn the language to its fullest extent. Overall, our objective is to make learning ASL easier and more fun.

Our initial approach is to train an AI with an Xbox Kinect by having a reference sign each letter. We now decided to go with the same approach but use our web camera instead. The significance of the sign language interpreter is that it would help teach people how to better communicate with deaf/hard of hearing individuals. It would allow a person to expand their knowledge of the language and give them a special benefit in most situations.

1.0 Problem

The problem that many people face when they start to learn sign language is that it is difficult to do, and even more difficult to do correctly. Learning sign language is unique because it's a completely visual language. This means that to practice sign language, you must either sign with someone else who knows sign language or record yourself signing and replay it.

In the U.S. alone, around 250,000 – 500,000 people rely on sign language for their day-to-day communication because of their hearing disability [12]. When you factor in the number of people who then rely on sign language to communicate with people with a hearing disability, the number grows even larger. An application that provides a feasible and interactive way to learn sign language would allow an entire group of the U.S. population to effectively communicate in public settings.

Without this application, sign language would remain a very difficult language to learn individually. To accurately learn sign language without forming bad habits, you would most likely have to join a class where a sign language educator could differentiate correct and incorrect gestures and provide feedback. Without something like this application to interpret sign

language, people who rely on it will inherently have a more difficult time doing everyday things that are specifically catered to people who do not have a hearing disability (i.e., audio-only drive-through option at restaurants).

2.0 Objective

The objective of this project is to write software capable of correctly interpreting American Sign Language (ASL) using a web camera.

3.0 Background

3.1 Key Concepts

Digital image processing is using a computer to analyze an image using algorithms [1][2]. This kind of processing can range from compression to enhancement and restoration. Some special image processing will likely be necessary to correctly pick out human bodies and their structure from the live image feeds.

Once an image has been processed, we will have to use computer vision concepts to correctly interpret the image. Computer vision is a broad field that deals with how to get computers to interpret images the way humans do [3][4]. This would be considered a high-level understanding of the image. Various algorithms are used to try to achieve the broad range of image interpretation that humans currently possess. Often these algorithms are complicated and require AI to function correctly.

AI Action recognition is the process by which an AI can recognize a specific action taking place in an image. It's also considered a subset of computer vision. Usually, the actions the computer is trying to recognize are different human actions, like walking, running, talking, etc. Often this requires deep learning to work effectively along with a large amount of input data from many contexts to train the AI [5].

Once the computer vision process has been completed for a given image, it must be compared to a library of sign language gestures. We'll be using American Sign Language specifically, although there are other sign languages [7]. It has unique signs for letters, numbers, and words all of which are formed using the hands, face, and other body language. While it is commonly used in English speaking countries, it has its own grammatical rules different from English grammar [6].

3.2 Related Work

Dr. Lora Streeter worked on a very similar project for her dissertation [8]. She used Kinect along with several different algorithms to try to identify user gestures and translate them into instructions for a gesture-based programming language. Early in the paper, she mentions that an application of her project could be helping people with certain disabilities, even mentioning a theoretical program that could recognize sign language gestures, which is a major part of what this project is trying to achieve. Our project will involve building a system that will likely interpret images similar to how hers did, but instead of using it for programming, we will use it for sign language.

We were able to find a project on GitHub created by Harsh Gupta, Siddharth Oza, Ashish Sharma, and Manish Shukla that is essentially a working implementation of what this project hopes to achieve. They created a sign language interpreter using deep learning in a day for the University of North Texas's 2019 Hackathon with Python and several other packages. The interpreter was capable of recognizing 44 different signs with high accuracy [9]. Our implementation will probably be similar, but we aim to have it recognize a larger number of signs.

Another similar project is the Kinect Sign Language project created by Svilen Popov on GitHub. It's a sign language translator that uses Kinect to get user input [10]. The project appears to be completed and working, but the description doesn't explicitly state that it works. Whether it's working or not, a Kinect-based sign language interpreter would be very useful to work off of for our project.

PyKinect2, a project that enables its users to create applications for Kinect using Python [11], is more general than the others listed but will be helpful. Having a Python library build specifically for Kinect will help us make progress much faster than if we didn't have it.

4.0 Design

4.1 Requirements, Use Cases, and Design Goals

4.1.1 Requirements

1. Connect the user's web camera through Python
2. Able to convert sensory data into Numpy datasets
3. Must allow for extensive training of datasets in order to improve AI accuracy
4. Must be able to predefine different sign language letters, words, and phrases into identifiable data for AI
5. Allow for the user to sign to the web camera and receive back the identified sign

4.1.2 Use Cases

1. User runs the application, allowing access to the user's web camera
2. User can provide a sign and receive the correct interpreted letter, word, or phrase

4.1.3 Design Goals

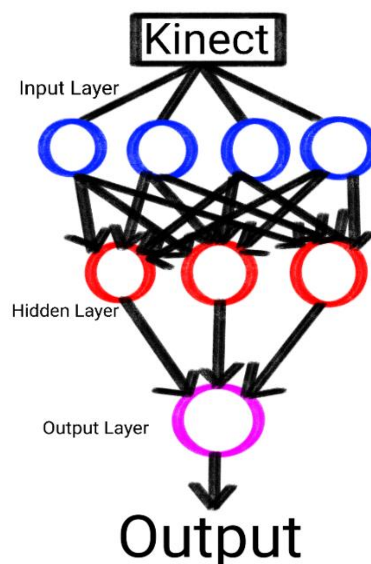
1. Fast, efficient code
2. Easy access for modifying set data and loading train model

4.2 Detailed Architecture

4.2.1 Architecture

We initially planned to use the Xbox Kinect to implement this project. To implement the sign language recognizer, we would start by creating a Python file that is able to connect to the Kinect API. By connecting to the Kinect API, we can access features such as Kinect body tracking and coordinate mapping. This will allow us to capture the left and right-hand joint coordinates from the device. From here, we will be able to convert the data into usable inputs for the neural network. At this point, we will have to implement the neural network using Python libraries such as TensorFlow and Numpy. The network will be trained with the relevant signs made in different positions and distances. The picture shows how the Kinect can be used to identify different hand symbols [13]:

Live video feed is received from the Kinect and converted into input for the input layer. The input goes through hidden layers to distinguish identifiable traits and determine which possible alphabet sign it might be. The final layer is the output layer, where all possible outputs will reside. Upon receiving output in the output layer, a result will be sent back to the program representing the neural network's determined alphabet letter. Below shows a diagram of how the data gets processed:



After encountering issues with the Kinect, we decided to switch to using our web camera. The structure of our program design would include the input from the web camera, the model that interprets the input, and the output to the screen. The input is a subsection of the total area of the webcam input cut into a 200x200 pixel image which is then manipulated to be ready to be interpreted. Without these changes to the image, it would not be able to be interpreted by the model. After the interpretation, the image to be displayed to the screen needs to have the output box inserted into the current frame, and then the interpreter's prediction is printed onto that output box. Once this is complete, this frame is displayed to the user and then the process is repeated with each new frame from the webcam.

4.2.2 Technologies Used

For the hardware, we used our own basic web cameras to provide the images for the model to interpret. We ran this program on our own computers as opposed to a remote processing center. Some of us were also able to use GPUs that we owned, which greatly sped up the development process. For the software, we decided to write our project entirely in Python. We picked Python because of its ease of use and a large portion of developers who attempted projects like ours used this language. There are also a wide variety of packages to choose from, many of which are designed to handle specific tasks. The main two that we were interested in were PyTorch and Tensorflow, both of which are popular machine learning libraries in Python.

Another one of the technologies we used was the OpenCV library for Python. The CV in OpenCV stands for "Computer Vision," and the developers have explicitly designed it to be a library used for computer vision projects like ours. We used it primarily for capturing images from our web cameras and for making various changes to those images. There are also a few scripts in the program used for making large scale changes to our databases that took advantage of OpenCV's ability to read and write images to the hard drive and used its image manipulation capabilities much more than what we would use for the interpreter. OpenCV was very easy to use and seemed well optimized for the role it filled in our project, which makes sense because it's designed to be used almost exactly like we were using it.

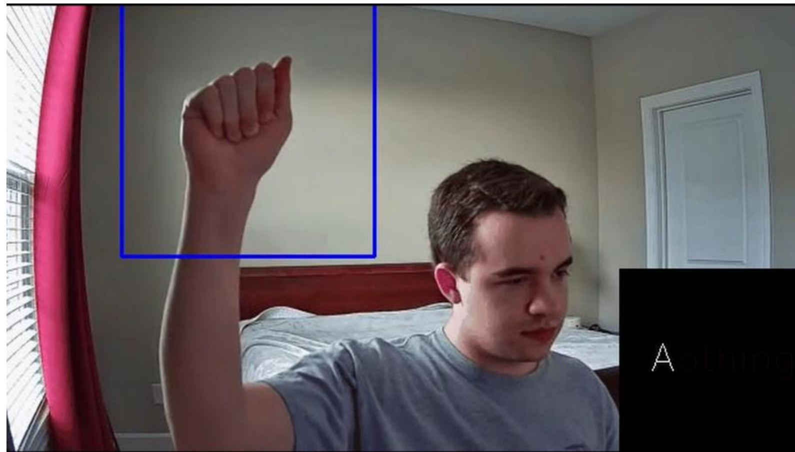
NumPy was another popular package we used. We initially installed it because it was required for OpenCV, although it is an important package used by many other libraries as well. It was essential for debugging our program, as many of the issues we ran into had to do with the size and shape of the tensors we were using, which would have been much more difficult to determine without this library.

PyTorch was the library we settled on using for the machine learning part of our program. The main reason why we picked this over Tensorflow was that David and Jack had prior experience with PyTorch and had negative experiences with Tensorflow. It was used to both train and test the images, save our model after it had been trained, load it when used by the interpreter, interpret the images from our webcam in real time, and allowed us to tweak our model during development.

Pillow was another library we used for this project. It was used exclusively for image manipulation. The only place it's been used is during the interpretation of a live image, where the image needs to be converted to the correct format before it can be interpreted.

4.2.3 Interface Design

Our interface was inspired by another sign language interpreter on Github [9] and showed us that we didn't have to have a very complicated user interface to accomplish our task. The interface consists of a stationary input box outlined in blue towards the top left of the screen and a stationary black output box in the lower right corner of the screen. The prediction from the model is printed in white text inside of the output box. These elements are all layered on top of the frames coming from the user's webcam. The input box itself is 200x200 pixels and is the only part of the webcam image used by the interpreter. The rest of the image is just so that the user can see him/herself and so that there is room for the output box. 200x200 pixels was chosen as the size for the input box based solely on the fact that the images used to train the model are also 200x200 pixels, and using an image of any other size would be less accurate than if we stuck to the original resolution. There were a few times we tried a different resolution, and it was always less accurate. Below shows our program interface:



4.2.4 Initial Implementation with the Xbox Kinects & Virtual Environments

The first thing we worked on was the Xbox Kinects. Only Samuel and Johnny were ever able to get ahold of a Kinect from the University of Arkansas, and the other group members had to have theirs shipped to them as they were living out of town. They never had a chance to work on the Kinects because we all decided to abandon them by the time their Kinects arrived. The first problem we ran into when working with the Kinects was one where it would flash a green light and refuse to give any video input. After some searching, we learned that one reason might be due to a lack of adequate power; however, this seemed unlikely since the adapter the Kinects used had a USB type B connector and another that plugged directly into an electrical outlet.

After that, we started looking for special drivers for the Kinect as we thought this might be the problem. Initially, we tried an open-source driver that wasn't created by Microsoft. The reason why we wanted to try this driver was due to its compatibility with Windows, Mac, and Linux.

This driver might have worked fine, but we had to compile it from its source code, which became too difficult as it required libraries with extremely complex installation instructions.

We moved on to finding Microsoft's original drivers for the device and were able to install them, which fixed the blinking green light but didn't give us any input. After that, we installed Microsoft's Kinect for Windows SDK, but the newest version was designed for Kinect version 2 for the Xbox One, which we did not have. So, we found an older version and installed that and were finally able to get some input from the Kinects; however, it was inconsistent and seemed to require a restart if you wanted to start and stop a program that used the Kinects more than once. It was at this point that we decided to abandon the Kinects.

Next, we started developing the Python virtual environment that we would use to encapsulate our project and make it more portable. We initially struggled with the creation and management of the virtual environment, eventually turning to Anaconda, a software used to help create Python virtual environments and manage their packages. This seemed to work well at first, but when we discovered that it stored environment packages in a location outside of where we had created the virtual environment, we decided to abandon this software and keep trying with the basic version that comes with Python.

After some work, Samuel was able to get the required packages installed in a virtual environment created using the basic version that was correctly able to differentiate between the packages installed in the virtual environment and the base Python environment. He moved the virtual environment to another computer, and it didn't work at all. He then asked his teammates to test this virtual environment to see if they had the same problem and they did. After some troubleshooting with no success, we decided to abandon Python virtual environments.

4.2.5 Final Implementation with the Web Cameras

In place of the Kinects, we opted to use basic web cameras as this would likely be easier for both the developers and the users to set up and use. However, we lost the extra information provided by the Kinect's special sensors. In place of the Python virtual environment, we opted to use the Python base environment. This did not provide any extra portability or encapsulation; however, it would be easier on both the developers and users by avoiding any unexpected complications that might come with using a virtual environment.

It was around this time that we started developing the app as it is now, as everything that we had worked on before had to be discarded. The first part of this new development was writing code to handle our webcam input, which we were quickly able to get running with OpenCV's library and this tutorial [14]. With the basics of the webcam working, we were able to move on to creating a model. We found the Kaggle dataset [15], which contained 3,000 pictures for each letter in ASL as well as for special gestures "space" and "del." It also contained an entry for "nothing," which was the only extra entry that we would keep in the final model as "space" and "del" were removed. The "nothing" entry was not one that we had considered having in our dataset before,

but it made sense to include it as there would have to be an entry for the model when it determined that it wasn't being shown any signs.

After finding the Kaggle dataset, we looked online for tutorials to help us get started with training out model and found this one [16]. Using this tutorial, we were quickly able to get the code to start training a model using the Kaggle dataset. Initially, we included code to test the model to try to determine its accuracy; however, we would later remove this code as the only test that really mattered for this project was the interpretation of live video feed. As far as the structure of the model goes, we decided to use a pre-trained RESNET model as it would reduce the amount of time spent on designing the model and we would likely get better accuracy. After troubleshooting some issues with our training code, mostly having to do with making sure there were the correct number of classes (possible predictions for the model) and that the image tensors were the correct dimensions, we were able to produce our first model. Very soon after, we managed to get the model to interpret the images coming from our webcam; however, we hadn't written the code to pick out a subsection for interpretation yet, so the model was interpreting the entire webcam frame. The accuracy was terrible at this point, but that was to be expected since the images coming from the webcam were nothing like the images the model had been trained for.

It was at this time that another version of our model training software was created using Jupyter notebooks and remote GPUs for the training; however, the data it was using to train was a very different format than the data we used for our first version of the training software. We found that it was significantly more difficult to convert input from our webcam into a format the new model produced by this code could process, so we quickly abandoned this version of the training software.

Work on cutting a subsection out of the webcam frame for processing by the model began immediately after we abandoned the new training software. Since we could arbitrarily pick any size image out of the webcam frame (even one bigger than the frame itself), we had a lot of freedom to make design decisions, but we decided to go the safe route and pick a subsection that was exactly the same size as the images used to train the model. We didn't want to risk the model making any mistakes due to us feeding it images that were the wrong size. So, a 200x200 subsection was picked towards the top left part of the webcam, which is a spot we thought would likely be less cluttered than the rest of the frame and allow for more accurate interpretation. This was quickly finished and for the first time we were able to start testing our model's accuracy. We were disappointed with the results as the model was not very accurate, but it was interpreting a few signs correctly. Since our dataset seemed to have enough variation to produce a decent model, we decided to start making changes to our training code.

While we had initially decided to use the RESNET model, we tried making our own custom model structure using this tutorial [17]. There were various issues that came up during the development of this model, mostly centered around having correct dimensions for the tensors being handled by the model, so after a lot of troubleshooting, we gave up and moved on, not wanting to waste time like we had on the Kinects and virtual environments. RESNET has

different versions of their RESNET models with higher and lower complexity, so we decided to try changing that. We switched to versions of RESNET with lower complexity due to the size of the first model produced from our training being around 92 MB. Lower complexity RESNET models are smaller than higher complexity models, and we also thought that our model was too sensitive and that having less detail would help produce better results. The models produced from training with lower complexity RESNETs were not any more accurate than our first model and were sometimes less accurate. At this point, we decided that we should stick to our original RESNET and try changing something else.

The images in the Kaggle dataset were normal RGB color images, so we thought that reducing the complexity via reducing the number of colors might produce better results and eliminate inaccuracies based on color. We had used some functions for converting RGB images to monochrome earlier in development, but we hadn't had any luck with conversions that picked a cutoff based on the sum of the RGB values as they never did a very good job differentiating between the background and the user's hand. However, we found the Canny function, which would find high contrast areas of an image and draw white pixels on that region while leaving the rest of the image black. This did a much better job of picking out the user's hand from the image; however, it would still outline other parts of the image. Even with these imperfections, we wrote a script to convert the entire dataset into a new dataset after passing the images through the Canny function. We trained a model with this dataset and added the Canny function to the subsection code, but we found that this new model was less accurate than our original model. The Canny function was abandoned at this point.

We still thought that reducing the complexity of the Kaggle dataset images was the way to go, so instead we tried reducing the overall size of the images in the dataset to 100x100, making the images a quarter of the size they used to be. Just like before, we wrote a script to convert the entire Kaggle dataset to 100x100 versions of the original dataset; however, it was easier to write this script as it was essentially a small modification of the script we had written for the Canny conversion. Once this was done, we modified the subsection code to reduce its 200x200 image to 100x100 and fed it into the new model we had trained with our low-resolution dataset. The accuracy was worse than with our original model. We didn't use the low-resolution model after this point.

A second attempt was made at creating a custom model, and we had more success with this one. We wanted to try making one that was more complicated than our first attempt at a custom model since it seemed like reducing complexity was not helping us when we tried it in any other parts of the project, so we followed this tutorial [18] to create our new custom model. There were some issues getting it running, once again relating to tensor dimensions, but this time we were able to get those fixed and were finally able to start training a new model. Probably about a second after we started running the training code with our new model framework, anyone who was running the model had their computer soft-locked due to extremely high memory and CPU usage. Everyone who tried running this model had to shut off their computer to stop the training. Our new model was abandoned at this point due to being unusable.

Finally, we decided that we should try training a model based exclusively on images of ourselves as the accuracy would probably be higher if the people testing the model were the same people in the training images. However, there was a problem: we would need a very large number of images for each sign if we were going to have decent results. We decided at least 1000 images was the best option. We wrote a script that allowed us to take many 200x200 images in a very short period of time. This script was more advanced than the previous two, which allowed the user to create the entire training dataset without having to restart the program, so a simple menu was created that worked by having the user press keys corresponding to the letter they wanted to create images for. The program also outputs the path the images are being saved to as well as the name and number of the image being saved. When the program would begin the process of writing images, it would first check to see if there were any images already written to that directory and would change the number in the image's name accordingly if it found any images already there. If the directory for the new custom dataset had not yet been created, the program also created those directories. With this script, we were able to quickly create our own datasets. At first, we tried having one person train a model using only their images and only testing that model on themselves to be sure that this method was going to work. We didn't get higher accuracy this way, so we initially thought that it would be pointless to combine our datasets as it would decrease accuracy by adding extra images that were not similar to the user or the user's room.

However, we remembered that the only time we had really had any success with training an accurate model was with the Kaggle dataset, so we thought that maybe combining our custom datasets with this one would yield a more accurate model. Combining the datasets was not as simple as copying both sets of images into a new dataset, many of the images had the same names and so would overwrite each other. Also, since there would be around 4,000 – 5,000 images per letter once the datasets were combined, renaming the images manually was not feasible. We then created another script to combine the datasets. It worked by creating a new directory for the combined datasets and then copying one dataset into the new directory, counting the number of images for each letter as it wrote the images. Once the first dataset had been copied, the second dataset would also be copied into the combined dataset directory, however the images for each letter would be renamed using the number of images counted when the first dataset had been copied. Soon we had our combined dataset and used it to train a new model. We should also note here that we simplified our training code using the official documentation from PyTorch [19] to skip any automated testing as the only functionality we cared about was how well it performed when interpreting live footage. Our new model was significantly more accurate than any of our previous models, and it's the model we used when creating our demo. A few more things were tried to make our model more accurate, like training for a longer or shorter amount of time, but for the most part additional development stopped after the creation of our demo model. The letters I, J, R, and V were not able to be correctly interpreted, but there wasn't enough time left for us to keep working on the model at that point.

Overall, the implementation of this project took about 3 months during which we tried and abandoned a few different technologies, and once we had a working model, we tried several different things to increase its accuracy.

4.2.6 Lessons Learned

Using old and unsupported proprietary software or hardware has a higher chance in running into issues. It majorly affected our progress, and there were not enough helpful and relevant resources to help solve our issue. Also, developing an app on something that is essentially dying out is not good for the longevity of your program, as users will have to acquire something that is limited in numbers and probably hard to find. Using hardware or software that is proprietary, old, or unsupported will also likely create more security vulnerabilities than if you had used something that had none of these qualities.

Focusing on peripheral features before critical ones isn't always a bad thing as long as you don't run into many problems and development goes well for both features; however, if you start having problems with either, you'll find that you quickly start to run out of time. Getting tunnel vision like we did with the virtual environments just takes away time that we could have used to create a more accurate model. To help understand the impact that this can have on a project, we would probably have another week and a half to try to further improve the accuracy of our model if we had skipped the virtual environment entirely.

4.2.7 Potential Impact

Creating a successful sign language interpreter via web camera will provide better accessibility to sign language education. To efficiently learn sign language, it's best to do it in-person. These opportunities are usually offered through universities and community colleges or hosted by deaf communities; however, these usually cost a fee up to $100 per session, which not everyone can afford or travel to these locations. Especially with current COVID restrictions, it limits accessibility even further.

In a broader sense, this project has an impact on the capabilities of computer vision, bringing more opportunities for businesses and the tech industry.

4.2.8 Future Works

This project mainly focuses on gesture recognition, and it can definitely extend to motion recognition, allowing an endless possibility for ASL words and phrases. Being able to implement other words and phrases can help the user better carry a conversation. Even extending this to other types of sign language such as British, Chinese, French, etc. would target a larger audience.

Sign language can be difficult to learn because each gesture has a specific hand position that can make an individual confused between words, phrases, and letters (i.e., signing A, E, and S). We can potentially add a tutorial section into our program so the user can know the specific finger position and hand posture. The tutorial can include a graphic hand model demonstrating the signed letter or word, and the user can manipulate the hand model, ensuring better accuracy and adjustments.

4.3 Risks

Risk	Risk Reduction
Inaccurate Sign Language	Extensive research of ASL
Inaccurate Neural Network	Train with many datasets in order to improve accuracy
Slow code	Write out algorithms and optimize for the best complexity time

4.4 Tasks

Our tasks are broken down into phases based on engineering design principles: Preparation, Design, Implementation, Testing, and Documentation Phases. Below is the list of tasks based on our final design approach:

4.4.1 Preparation Phase

1. Understand how to collect data from our web camera. To have a general basis for our approach, we will reference some of the related works mentioned before.
2. Install the necessary software(s), which mainly consist of Python libraries.

4.4.2 Design Phase

1. Develop the user interface where the user starts signing to the camera, and the letter gets interpreted.

1.1 The main goal of this application is to get most of the alphabet interpreted, and if we have enough time, then we can start interpreting other categories of ASL words/phrases.

2. Develop a dataset for each letter in the alphabet.

4.4.3 Implementation Phase

1. Implement the first three letters of the alphabet (A, B, C) along with the letters J and Z.

1.1 This will help build a foundation for the remaining letters. Letters J and Z are included because, unlike the other letters, they require motion.

2. Complete the remaining letters of the alphabet.

2.1 The alphabet will either be split to each team member or be split to 2-3 members while the remaining will work on the application. A sample distribution is shown below:

Team Member	Set of Alphabet Letters
David Clairmont	D – G
Johnny Doan	H – L
Jack Gaither	M – P
Nick Hester	Q – U
Sam Witucki	V – Y

3. Once we complete the alphabet, we can then extend to other categories of sign language (i.e., numbers, time, people, questions, etc.) if we have enough time.

4.4.4 Testing Phase

1. Run the application and check if it fulfills our objective and expectations.
2. Test to see how accurate the web camera can depict each letter of the alphabet.

2.1 Testing will include different angles of the person's hand with respect to the web camera.

4.4.5 Documentation Phase

1. Report the results of our application. Screenshots and a video walking through the application will be provided so the user understands how the program works.
2. Record a demonstration of the program being able to read the person signing.

4.5 Schedule

Below indicates our proposed timeline for each task listed in section 4.4:

Tasks	Dates	Person(s)
1. Software preparation & Determining necessary equipment	1/11 – 1/22	All
2. Requesting equipment & Waiting for responses/updates	1/25 – 2/5	All
3. Setting up Python application and virtual environment.	1/25 – 2/5	Samuel
4. Testing Python virtual environment and compatibility with the Kinect (for those who had the Kinects).	2/8 – 2/12	Samuel, Johnny
5. Reevaluating project and change approach	2/15 – 2/19	All
6. Work on Python application that collects data from MP4s or web cameras. Create a user interface.	2/22 – 3/5	All
7. Work together on interpreting the letters A, B, C, J, & Z	3/8 – 3/12	All
8.1. Interpret letters D – G & Test the letters	3/15 – 3/19	David
8.2. Interpret letters H – L & Test the letters	3/15 – 3/19	Johnny
8.3. Interpret letters M – P & Test the letters	3/15 – 3/19	Jack
8.4. Interpret letters Q – U & Test the letters	3/15 – 3/19	Nicholas
8.5. Interpret letters V – Y & Test the letters	3/15 – 3/19	Samuel
9. Improve the model to increase accuracy.	3/22 – 4/2	Samuel
10.1. Record video demonstrating the project.	4/5 – 4/16	Samuel
10.2. Provide instructions to run the project.	4/5 – 4/16	Samuel
10.3. Documentation of project results (website, slides, poster).	4/5 – 4/16	All
11. Final adjustments before submitting project.	4/19 – 4/29	All

4.6 Deliverables

- Design Document: Contains a listing of each major software component.

- Dataset: The dataset with all of the signed letters.
- Python code for training the AI, gathering and storing optical data, digital image processing, etc.
- Video demonstrating the program interpreting the working letters
- Final Report

5.0 Key Personnel

David Clairmont – Clairmont is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed relevant courses in Programming Foundations I & II, Programming Paradigms, Software Engineering, and Database Management Systems. He is currently an undergraduate research assistant for Dr. Qinghua Li. Clairmont will be responsible for creating the Python files to connect to the Kinect API and programming some letters of the alphabet.

Johnny Doan – Doan is a senior Computer Engineering major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed relevant courses in Programming Foundations I & II, Programming Paradigms, Database Management Systems, Software Engineering, and Algorithms. Doan will utilize his leadership experience from student organizations to keep track of each member's progress. He will also use his class experiences to work on the set of alphabets and the application that connects the program to the Kinect.

Jack Gaither – Gaither is a senior Computer Science major in the Computer Science and Department at the University of Arkansas. He has completed relevant courses in Programming Foundations I & II, Programming Paradigms, Database Management, Software Engineering, Cryptography. He has experience with running various HPC applications, image processing, action recognition/segmentation, and teaching python tutorials. Gaither has also interned for the past three years with the Texas Advanced Computing Center, starting as an REU student and then mentoring for the past two years for various student programs. He now works with Dr. Luu in the CVIU lab on campus doing action recognition/segmentation. He will be responsible for working on the set of alphabets and testing each of the letters.

Nick Hester – Hester is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed relevant courses which include Programming Foundations I & II, Programming Paradigms, Database Management Systems, and Software Engineering. He has experience creating a website that will come in handy when creating the website for this class. Hester will be the main reference for the initial signing training. Hester is capable of signing the full alphabet along with the simple phrases that will be used to train the AI and will work on the testing portion of the project along with programming the set of alphabets.

Sam Witucki – Witucki is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed Programming Foundations I & II, Programming Paradigms, Software Engineering, and Database Management Systems. He is currently employed as an IT/Engineering Intern, writing software at J.B. Hunt. He will be responsible for writing some of the Python code for this project as well as providing an extra reference for the sign language interpreter.

6.0 Facilities and Equipment

For this project, the main equipment we initially used was the Xbox Kinect. This would've acted as our main medium for our image capture. We ended up utilizing a GPU machine that can be accessed via SSH for our initial image processing and model training.

7.0 References

[1] Digital image processing, https://en.wikipedia.org/wiki/Digital_image_processing

[2] Digital Image Processing Introduction, https://www.tutorialspoint.com/dip/image_processing_introduction.htm

[3] Computer vision, https://en.wikipedia.org/wiki/Computer_vision

[4] What Is Computer Vision?, https://www.pcmag.com/news/what-is-computer-vision

[5] Introduction to action recognition, https://www.coursera.org/lecture/deep-learning-in-computer-vision/introduction-to-action-recognition-Ift3j

[6] American Sign Language, https://www.nidcd.nih.gov/health/american-sign-language

[7] American Sign Language (Wikipedia), https://en.wikipedia.org/wiki/American_Sign_Language

[8] Streeter, L., "Teaching Introductory Programming Concepts through a Gesture-Based Interface" (2019). *Theses and Dissertations*. 3240. https://scholarworks.uark.edu/etd/3240

[9] Sign Language Interpreter using Deep Learning, https://github.com/harshbg/Sign-Language-Interpreter-using-Deep-Learning

[10] Kinect Sign Language, https://github.com/sgpopov/kinect-sign-language

[11] PyKinect2, https://github.com/Kinect/PyKinect2

[12] Mitchell, R., "How Many People Use ASL in the United States? Why Estimates Need UPdating," 2005. https://www.gallaudet.edu/documents/Research-Support-and-International-Affairs/ASL_Users.pdf

[13] Mustafa, E., Dimopoulos, K., "Sign Language Recognition using Kinect," 2014. https://www.researchgate.net/profile/Konstantinos_Dimopoulos2/publication/266144236_Sign_Language_Recognition_using_Kinect/links/5447bc8b0cf2f14fb81228ca/Sign-Language-Recognition-using-Kinect.pdf

[14] Hand Detection and Finger Counting Using OpenCV-Python, https://medium.com/analytics-vidhya/hand-detection-and-finger-counting-using-opencv-python-5b594704eb08

[15] ASL Alphabet, https://www.kaggle.com/grassknoted/asl-alphabet

[16] How to Train an Image Classifier in PyTorch and use it to Perform Basic Inference on Single Images, https://towardsdatascience.com/how-to-train-an-image-classifier-in-pytorch-and-use-it-to-perform-basic-inference-on-single-images-99465a1e9bf5

[17] PyTorch Image Recognition with Convolutional Networks, https://nestedsoftware.com/2019/09/09/pytorch-image-recognition-with-convolutional-networks-4k17.159805.html

[18] Deep Learning for image classification w/ implementation in PyTorch, https://towardsdatascience.com/convolutional-neural-network-for-image-classification-with-implementation-on-python-using-pytorch-7b88342c9ca9

[19] Training a Classifier, https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html