

SYNTHETIC HAND GESTURE DATASET GENERATION

An Undergraduate Research Scholars Thesis

by

SAMUEL ONCKEN¹ AND STEVEN CLAYPOOL²

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor:

Prof. Stavros Kalafatis

May 2023

Majors:

Electrical Engineering^{1,2}

Copyright © 2023. Samuel Oncken¹ and Steven Claypool².

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

We, Samuel Oncken¹, and Steven Claypool², certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with our Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

	Page
ABSTRACT.....	1
DEDICATION.....	3
ACKNOWLEDGEMENTS.....	4
1. INTRODUCTION	5
1.1 Background.....	6
2. METHODS	8
2.1 Requirements	9
2.2 Software Interface.....	14
2.3 Gesture Data Collection Subsystem	17
2.4 Human Model Generation Subsystem.....	22
2.5 Model Animation Subsystem	25
2.6 Data Capturing/Collection Subsystem.....	31
2.7 Dataset Generation Full System	36
3. RESULTS	41
3.1 Synthetic Dataset Generation/Replication.....	41
3.2 Dataset Training and Testing Validation.....	51
4. CONCLUSION.....	60
4.1 Interpreting Results.....	60
4.2 Applications.....	60
4.3 Future Analysis.....	61
REFERENCES	62

ABSTRACT

Synthetic Hand Gesture Dataset Generation

Samuel Oncken¹ and Steven Claypool²
Department of Electrical & Computer Engineering^{1,2}
Texas A&M University

Faculty Research Advisor: Prof. Stavros Kalafatis
Department of Electrical & Computer Engineering
Texas A&M University

The development of human-computer interaction has led to an increasing need for large and diverse hand gesture datasets for image classification in machine learning. However, generating such datasets leads to many issues with cost, privacy, and time. This research aims to streamline the dataset generation process using synthetic data. A Leap Motion Controller (LMC) hand tracking device is used to record hand gestures to replicate existing or create new datasets. A virtual environment is simulated in the Unity game engine, where the newly recorded hand gestures are applied to virtual human models generated with MakeHuman software and images are captured of the model's gesture performance to generate a synthetic dataset of any specified size. In the virtual environment, diverse backgrounds and human models are used to add complexity to the data to further improve the robustness when training machine learning models. Tests will be run using these datasets on various machine learning models to determine the viability of augmenting or replacing existing or new real hand gesture datasets. The models will be trained and tested with both real and synthetic data and the recognition accuracy obtained from each will be compared in a future section of this paper.

With a virtual environment, adjustments to the system for different datasets, human models, and gesture types become far simpler and require less resources than when generating real datasets. The use of virtual human models also eliminates the legal privacy concerns that arise when using real human participants as models. This method can be modified for gestures from any part of the body and potentially for non-human subjects. The application of this approach has significant potential to improve other areas of machine learning and computer vision.

DEDICATION

To our friends and families for continuously supporting and motivating us throughout the research process. We dedicate this paper additionally to our faculty advisor and graduate student TA for consistently offering advice and assistance to support our research in any way possible.

ACKNOWLEDGEMENTS

Contributors

We would like to thank our faculty advisor, Professor Stavros Kalafatis, and graduate student Pranav Vaidik Dhulipala, for their guidance and support throughout the course of this research.

Thanks also go to our friends, family, and colleagues and the department faculty and staff for making our time at Texas A&M University a great experience.

The materials used in Synthetic Hand Gesture Dataset Generation were provided by the Department of Electrical & Computer Engineering and specifically by our graduate student faculty advisor Pranav Vaidik Dhulipala. The analyses depicted in Synthetic Hand Gesture Dataset Generation were conducted in part by Pranav Vaidik Dhulipala and these data will be included in a future section of this document.

All other work conducted for the thesis was completed by both student thesis authors.

Funding Sources

Undergraduate research was supported by the Department of Electrical & Computer Engineering at Texas A&M University.

This undergraduate research received no other form of funding or support.

1. INTRODUCTION

Building the datasets required to train hand gesture recognition neural networks takes significant time and personnel to gather all necessary data. Not only do the training sets require thousands of images for each gesture, but the images themselves must also be diverse in terms of participant features and gesture performance randomness. Recording this data manually with camera and lighting equipment, real human participants, and human organizers/dataset creators can take months of planning and large sums of money. To avoid this, we have created a virtual environment within the Unity game engine that uses recorded gesture data from one individual using a Leap Motion Controller (LMC) and applies the recorded gesture onto unique virtual human models. After each iteration of spawning a random human model and applying a random gesture to it, our virtual environment captures an image of the character's hand and stores/organizes the data into gesture specific folders designated by the user. After a full training set is created, the usefulness of the virtual data will be evaluated by training and testing in gesture recognition neural networks. The goal of this research is to prove whether we can attain similar accuracy in hand gesture recognition when training with virtual datasets in comparison to real, benchmark datasets. Ultimately, this research will prove whether virtualized training sets are viable options for dataset creators on a budget or without access to highly specialized equipment who still require diverse data that can train hand gesture recognition neural networks to highly rated standards of accuracy.

1.1 Background

Many scholarly articles covering machine learning and artificial intelligence have been written in the last decade, proving its increasing importance across various industries in society. Our research focuses on a distinct category of machine learning and AI: computer vision. More specifically, we seek to uncover how well virtual (synthetic) data can train a neural network to recognize hand gestures. The first stage of this project is to create a user-friendly Unity environment where a dataset organizer can produce comprehensive and scalable hand gesture datasets capable of training a hand gesture recognition neural network. The next stage of this project involves understanding and training existing image classification Convolutional Neural Networks (CNNs) to compare the recognition accuracy when training with various compositions of real and virtual data [1,2,3].

Many of the datasets we have found online, summarized by [4], require a handful of real subjects to perform similar actions in front of a motion sensor (Kinect, Wii remote, etc.) which is recording from a set location. By hiring several different humans to perform the same gesture, some variance between gestures is collected which is required given that a gesture recognition system must not only recognize one size or shape of hand for it to function properly. Other datasets like the American Sign Language Lexicon Video Dataset and those derived from it [5] record human subjects using synchronized cameras all recording from different angles. Once again, this is beneficial to the machine learning process because in practice, a system will not always be looking at a human from a single angle, therefore it is required that a machine learning model be trained on hand gesture data recorded from many different perspectives.

Our research is aimed at bringing these important considerations together in the generation of a “virtual” dataset. In a virtual dataset, instead of having several human subjects

record their movements manually, we will be recording the movements of one human (under various conditions such as lighting, distance from sensor, etc.) and applying the motion to randomly generated 3-D virtual human models within the Unity game engine environment. In addition, within Unity, every iteration of a gesture performance will be recorded using multiple virtual cameras to improve gesture recognition from numerous angles.

We have uncovered a handful of research papers pursuing the application of synthetic data in computer vision algorithms from a wide variety of topics including but not limited to satellite imagery [6], robotics [7], and random object detection [8]. Similar research has been done by a Texas A&M alumnus in which a virtual training set of hand gestures was used to increase the recognition accuracy of gesture recognition neural networks [9]. We are looking to expand upon this student's research because we believe that this method can prove to be a cost-effective solution that can produce high recognition accuracy results, which we will be evaluating when testing our virtual datasets within hand gesture recognition neural networks in a future portion of this document.

2. METHODS

Collecting the gesture data required to extensively train a hand gesture recognition neural network is time consuming and resource intensive. Our aim is to establish a new method of training set generation using virtual data that takes significantly less time, human participants, and money than the traditional routine. Through our research, we shall develop a platform that takes as input user recorded gestures (through an LMC) and outputs a full training set, consisting of thousands of images of each gesture (taken from random camera angles) on unique virtual human models. Additionally, we seek to understand whether virtual data can be used to train a hand gesture recognition neural network to similar if not improved performance when compared to a model trained using real gesture data. We shall do this by building at least two virtual training sets that include the exact gestures of existing, real training sets, then comparing the recognition accuracy achieved when used in the same two hand gesture recognition neural networks. Figure 1 displays a conceptual image of this project.

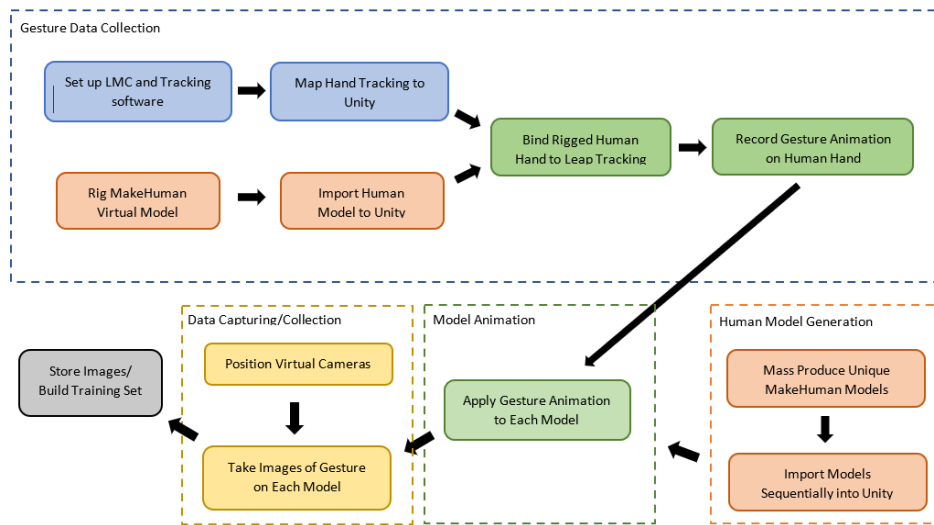


Figure 1: Project Conceptual Image

2.1 Requirements

2.1.1 System Definition

The hand gesture dataset generation system allows users to easily create extensive and diverse hand gesture training sets using a virtual environment. Using a virtual environment, dozens of human participants are no longer required to generate gesture images while still enabling the training of neural networks to equivalent recognition accuracy. The system has four subsystems: Gesture Data Collection, Human Model Generation, Model Animation, and Data Capturing/Collection.

The dataset generation Unity environment shall consist of 2 separate scenes. The first scene will house the Gesture Data Collection subsystem, where hand gestures will be recorded using the LMC (mapped to an example MakeHuman model) in Unity and saved as animation clips. Each of these gesture recordings will be accessible to the second scene components and stored to form an animation dataset.

The second scene will house the remaining subsystems. First, the Human Model Generation subsystem will randomly create tens/hundreds of diverse human models and import each sequentially to the Unity environment. Next, the Model Animation subsystem takes a random hand gesture animation clip that was just recorded by the user and applies it to the spawned human model. As this model performs the gesture, the Data Capturing/Collection subsystem takes an image from a randomized virtual camera angle (centered around the palm of hand) and organizes the images into complete virtual training set folders by name.

2.1.2 Functional / Performance Requirements

2.1.2.1 Gesture Recognition Accuracy

The metric of performance for the neural network trained with a virtual training set shall be equivalent to the performance metric when using the benchmark training set. We have decided that the gesture recognition accuracy when training using our synthetic data (or combinations of real and synthetic data) shall be no more than 5% below the result when using the benchmark real training set.

Rationale: For a comparison to be valid, the performance metric of each test must be the same. The gesture recognition accuracy with the virtual training set must be roughly equivalent or better than the accuracy with the benchmark training set to validate the usage of virtual training sets.

2.1.2.2 Computer Requirements

The computer running the environment shall use Windows 7+ or Mac OS X 10.7+ with X86 or X64 architecture CPU, a dedicated GPU, 2 GB of RAM, and a USB 2.0 port.

Rationale: Specifications provided by LMC datasheet [10] and Unity 2021.3 manual [11].

2.1.3 Physical Requirements

2.1.3.1 LMC Mounting Position

The LMC will operate in a head mounted position to record hand gesture motions.

Rationale: We tested various LMC mounting positions and found that hand motion was most accurately recorded when the LMC was mounted on the head and directed at the hands.

2.1.3.2 Head Mounting Apparatus

The head mounting apparatus shall be able to hold 32g of weight.

Rationale: The LMC weighs 32g, which is the only item being mounted.

2.1.4 Electrical Requirements

2.1.4.1 LMC Power Input

The LMC requires a 5V DC input with a minimum of 0.5A via USB.

Rationale: Specification provided by LMC datasheet.

2.1.4.2 Data Output

The system shall output a complete virtual hand gesture training set, composed of images in the file format of .jpg. Images can also be stored as .png files, but that must be changed in the code of a “SceneController” script within the Unity virtual data collection platform. The input file format of the used gesture recognition neural network must be equivalent to the data output produced by our system.

2.1.4.3 Connectors

The Hand Gesture Recognition system shall use a USB-A to Micro-b USB 3.0 cable to connect the LMC to the computer.

2.1.5 Software Requirements

2.1.5.1 Neural Networks with TensorFlow 2.8.0

The neural networks used shall work with TensorFlow.

Rationale: TensorFlow is a free, open-source software library commonly used in the training of neural networks. TensorFlow is used because of its high accessibility, ease of use, and large support network.

2.1.5.2 Virtual Environment - Unity

The virtual environment used shall be any version from Unity 2021.3 and newer.

Rationale: Unity is a free game engine software that allows for easy use of scripting within the created environments. Additionally, Ultraleap provides Unity specific plug-ins and documentation/support for the use of the LMC hand tracking software.

2.1.5.3 MakeHuman

The MakeHuman software shall be the latest stable version (1.2.0).

Rationale: MakeHuman offers a mass produce function that allows us to randomly generated human models. We will be importing models with a Default rig, which includes all hand and finger bones for the hand gesture animation to run. We will be using the latest stable version to avoid any potential incompatibility.

2.1.5.4 Leap Motion Controller

The Leap Motion Controller shall use Gemini - Ultraleap's 5th Generation Hand Tracking software.

Rationale: This is the latest version of hand tracking software offered by Ultraleap. It offers the highest quality hand tracking and has large amounts of usage/support documentation.

2.1.5.5 Unity Plugins

The Ultraleap, Animation Rigging and Unity Recorder Unity plugins are needed to create and run the Unity virtual environment.

Rationale: Ultraleap Unity Plugin includes required scripts for hand tracking/binding as well as useful prefabs of fully rigged hand models for testing. The Unity Recorder is used in the gesture recording process to track bone transform data and export the animation as a usable animation clip file for application on future models. The Animation Rigging package offers a Bone Rendering option which displays bone position on the character model, allowing for easy manipulation of transform data and access to specific bones.

2.1.5.6 Image Classifiers Python Package

The latest version (1.0.0b1) of PyPI's image-classifiers python package is needed for loading pre trained neural networks for use.

Rationale: The Keras API in TensorFlow is missing some useful image classification models that PyPI's package has, including models we used such as ResNet18.

2.1.6 Environmental Requirements

2.1.6.1 Lighting

There shall be adequate lighting when recording hand gestures with the LMC.

Rationale: The LMC loses hand tracking accuracy in darker conditions, reducing virtual hand gesture quality and therefore decreasing gesture recognition neural network accuracy.

In the virtual environment, lighting shall be manipulated and randomized while performing and recording animations.

Rationale: By varying lighting conditions, we can achieve higher gesture recognition accuracy for more edge cases in testing where images might be darker or brighter than expected.

2.1.6.2 LMC Operating Conditions – Temperature, Humidity, Altitude

The LMC Shall be operated in consideration of the following constraints. The LMC shall be operated from 32° to 113° F. The LMC shall be operated at a humidity between 5% to 85%.

The LMC shall be operated at altitudes between 0 to 10,000 feet.

Rationale: Operating conditions are provided by the LMC datasheet.

2.2 Software Interface

2.2.1 LMC Interface

2.2.1.1 LMC Tracking in Unity

The Leap Motion Controller can interface with Unity using the “Ultraleap Plugin for Unity” downloaded from the Ultraleap website. This plugin includes a Service Provider (XR) prefab which enables hand tracking and displays transform data relative to the head mounted LMC as shown in Figure 2 below. Additionally, this plugin includes several rigged hand prefabs as well as scripts that enable the tracking of each individual bone.

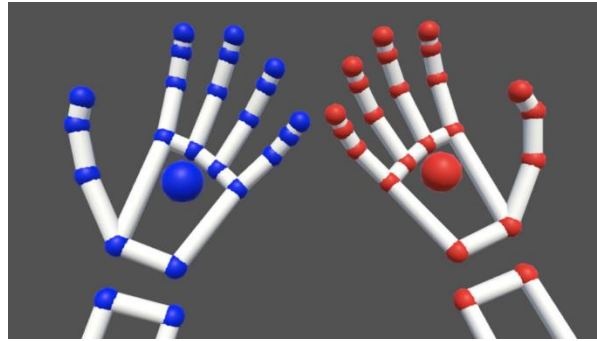


Figure 2: Ultraleap Rigged Hand Model in Unity

2.2.1.2 LMC Tracking on MakeHuman Model

The LMC must interface with an imported MakeHuman model within Unity for the Gesture Data Collection process. This can be carried out inside of Unity using the Service Provider (XR) prefab as well as the “Hand Binder” script offered in the Ultraleap Unity Plugin. Using this script, hand and finger bones of the rigged MakeHuman model are directly mapped to the LMC hand tracking software, which enables user hand motion to directly affect the movement of the virtual human hands as shown in Figure 3.

Rationale: Through our research thus far, we have attempted to record animation clips using the Ultraleap provided rigged hand models then apply the animation to the human model. This worked to some extent, however, the x, y, and z orientations for some bones on the provided hand models were different than the bone orientations of the MakeHuman models, which resulted in incorrect direction of motion on the virtual human. As a result, we found that directly mapping the LMC to the MakeHuman model was more consistent, thus requiring an interface between the two.



Figure 3: LMC Tracking Directly to MakeHuman model

2.2.2 Human Model Interface

2.2.2.1 MakeHuman Input Interface

To use the human models created in the Human Model Generation subsystem in the dataset generation Unity environment, all models must be set to the default rig and exported as .fbx files. Figure 4 displays the bone structure of a character with a default rig. After generating and exporting over 50 unique virtual models to a “Models” folder within the “Assets” folder of the virtual environment, the Model Animation Subsystem can access the human model files for spawning and animation performance within the Unity environment.

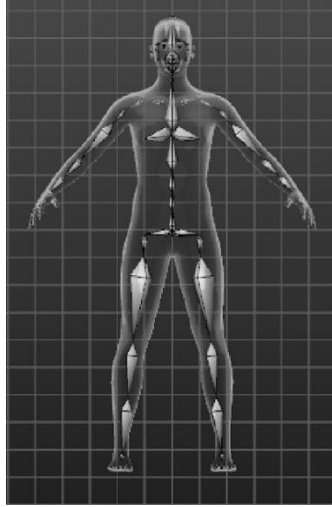


Figure 4: MakeHuman default rig for a human model

2.2.3 Human Model Animation Interface

After gesture recording is completed and a finalized animation clip dataset is created, it is necessary for the animations to be applied to randomly generated MakeHuman models. This shall be done by adding an “Animator” component to the lower arm bone of the MakeHuman rigged model. The behavior of the animator is determined by an animation controller, where each gesture animation clip is placed. When run, the model will perform the gesture recorded in the animation clip. This will be randomized to apply any given gesture on any given MakeHuman model.

2.2.4 Unity Camera Interface

The Data Capturing/Collection subsystem works concurrently with the Model Animation subsystem. As human models are animated, multiple cameras in front of the human model take images or videos at various angles. All images are taken as .jpg files.

2.2.5 Data Storage Interface

The images or videos taken by the cameras in Unity will be organized by dataset and by labeled gesture in the file explorer of the created Unity project. The script will save the full synthetic training set to a designated folder location. Each image will also be labeled according to the naming convention of the real dataset being replicated. For instance, if gesture images of one are labeled as “one” followed by the image number in the real, replicated dataset, the virtual dataset generation system will name all images of one in the same manner.

2.3 Gesture Data Collection Subsystem

2.3.1 Subsystem Introduction

The gesture data collection subsystem is a scene in the Unity project environment where a user can record his/her own gesture animations to be included in their desired dataset. To use this subsystem, the user must own a Leap Motion Controller and have downloaded the necessary tracking software and Unity plug-ins from the Ultraleap website as described in the previous section. All instructions for use of this subsystem have been written in a GitHub repository which will act as a user instruction manual. The gesture data collection subsystem outputs unique animation clips for future use in the model animation subsystem. For this research, the gesture recordings are designed to match existing, real dataset gesture performances to compare dataset recognition accuracy results in future analysis. Our virtual environment will be used to replicate three existing datasets: Sign Language for Numbers [12], American Sign Language [13], and HANDS datasets [14].

2.3.2 Subsystem Details

The purpose of the gesture data collection subsystem is to record and store user-created animation clips into the Assets of the Unity project folder, as shown in Figure 5. By storing

animation clips separately, they can be placed into Animation Controllers and therefore be scripted to play at random on imported virtual human models in future subsystems. This subsystem is the root of the dataset generation system as without it, uniquely identifiable gesture animations would not exist, and a hand gesture dataset could not be created. Due to this, this subsystem has been tested and altered various times to ensure ease of use and accurate gesture recordings.

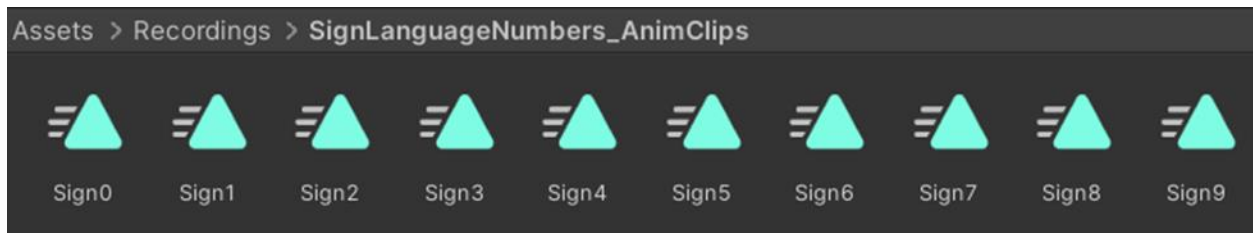


Figure 5: Stored Sign Language for Numbers Gesture Animation Clips

The main challenge with this subsystem was determining the best method of recording gesture animations while planning for animation application in future subsystems. Initially, gestures were recorded (and exported as .fbx files) on a sample hand prefab provided by Ultraleap. These prefabs contained the necessary scripts already written and configured by the Ultraleap developer team to simplify use. However, using this method, the hand bone transformation data contained in the created animation clip (within the exported .fbx file) led to compatibility issues with MakeHuman models. Using the Unity recorder, animation clips are simply keyframes of transformation data (position, rotation, and scale) recorded on a fixed interval for each bone of the model. The problem was that the Ultraleap-provided hand prefabs contained bone rigs that were structured and named differently than those within the MakeHuman models (shown in Figure 6) which were to be used in animation application. These differences resulted in the animation clips not being able to play on MakeHuman models. In

addition, Figure 7 shows that when exporting the gesture animation as a .fbx file, the animation clips that came with the file contained re-named bones, where each instance of ‘.’ was automatically renamed as ‘_’ (e.g., “wrist.L” turned into “wrist_L”), which once again resulted in naming incompatibility and animation application issues.



Figure 6: MakeHuman Default Rig Hand Bones Labeling



Figure 7: Animation clip using FBX Exporter - Not Compatible

The solution to this problem was to map Leap Motion hand tracking data directly to a MakeHuman model (shown in Figure 8) and export as an animation clip alone, which resulted in

animation clip keyframes similar to that displayed in Figure 9 that can be directly applied to freshly imported MakeHuman models.

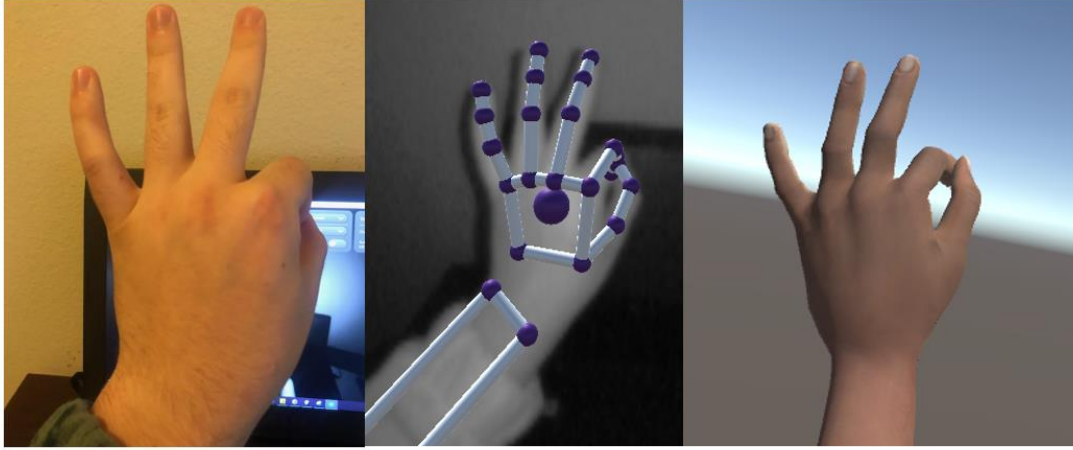


Figure 8: Recording of Animation Clips Directly on MakeHuman Model



Figure 9: Animation Clip using Finalized Method - Compatible and Accurate

2.3.3 Subsystem Validation

As the gesture data collection subsystem is the root of other systems, this subsystem was validated by ensuring that the output animation clips were easily accessible and were able to be applied onto imported MakeHuman models in the model animation subsystem. As mentioned in the above paragraph, rigorous testing of different methodologies to record gesture animations were attempted but the final factor in determining which to use was how well the gestures could be applied to a new MakeHuman model. To test the model animation application, a new Unity

scene was created where a freshly generated MakeHuman model with a “Default” rig would be imported. An Animator component was then added to the “lowerarm02.L” bone of the model since each animation was recorded from the lowerarm02. A sample animation clip was then placed into an Animation Controller, shown in Figure 10, which controlled the Animator component on the model. Once the animation was able to be applied correctly and stop its motion in the final hand gesture position as desired, the subsystem proved to be applicable to future subsystems.

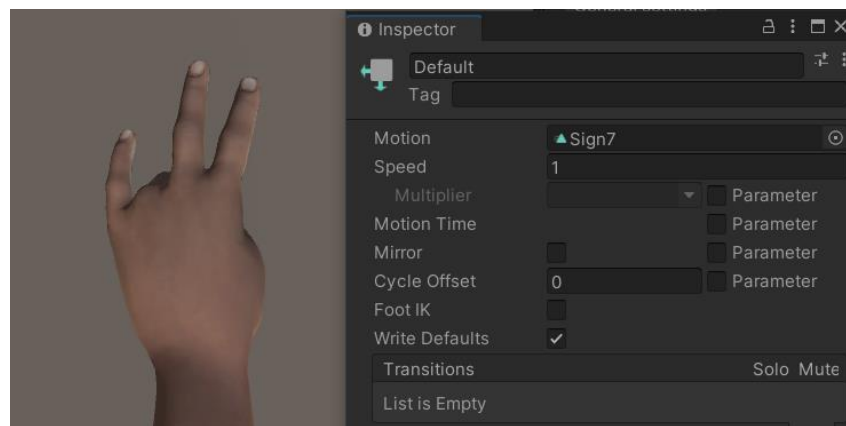


Figure 10: Test Application of Sign7 Gesture Animation Clip

In addition, the use of the Leap Motion Controller was validated within this subsystem by testing various LMC mounting positions under different lighting conditions to see which resulted in the highest tracking accuracy upon inspection. It was confirmed that by head mounting the LMC, the best motion tracking was achieved for the gestures that were planned to be recorded. Many of the gestures within the real hand gesture datasets involved putting one or more fingers down (into palm region) while still holding up others. It had to be ensured that the LMC could recreate this type of movement accurately. Figure 11 depicts the inaccuracy of recording this type of motion using Desktop Mounted mode as opposed to Head Mounted mode.

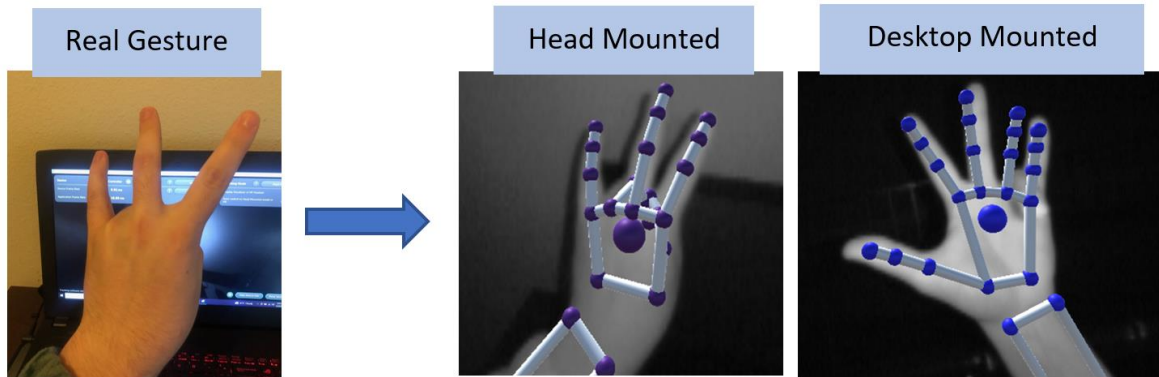


Figure 11: Tracking Accuracy of Head Mounted LMC vs. Desktop Mounted LMC

2.4 Human Model Generation Subsystem

2.4.1 Subsystem Introduction

The human model generation subsystem is responsible for the creation of the human models through the MakeHuman software for gesture animation, as well as the loading and random spawning of the models within the final system's environment. This subsystem works in conjunction with the model animation subsystem, where the created human models (with diverse skin tones, handwear, etc.) are animated in the virtual environment. The subsystem takes input from the user on the number of models to use and loads in the models from the assets folder with names specified in the created Unity script. When the final system is run, the script spawns the model into the scene for the model animation subsystem to manipulate.

2.4.2 Subsystem Details

Adjustments to the MakeHuman code were originally planned to automate the generation of human models. However, after looking through the Mass Produce plugin, shown in Figure 12, and MakeHuman's code and running into multiple issues with the process, it was determined that it would require extensive work to make such changes and as a result, MakeHuman models were generated using a manual approach since this was outside the scope of our work. The Mass

Produce plugin is used to create numerous MakeHuman model files (.MHM extension) that are then reloaded back into MakeHuman. For each model, a default rig is added in ‘Skeleton’ of the ‘Pose/Animate’ tab, and handwear/nails are added to 20-30% of the models in ‘Clothes’ of the ‘Geometries’ tab. The handwear/nails are not in MakeHuman by default, so the user must go to the ‘Download assets’ in the ‘Community’ tab to search for and download the desired textures, shown in Figure 13. The model’s heights are all adjusted to roughly 165 cm to ensure the hands of the models are within the frame of the cameras when loaded into the virtual scene. Finally, the model is exported as a FilmBox file (.fbx extension) directly into the model folder within the Unity directory’s ‘Assets’ folder, where the scene controller script can access them.

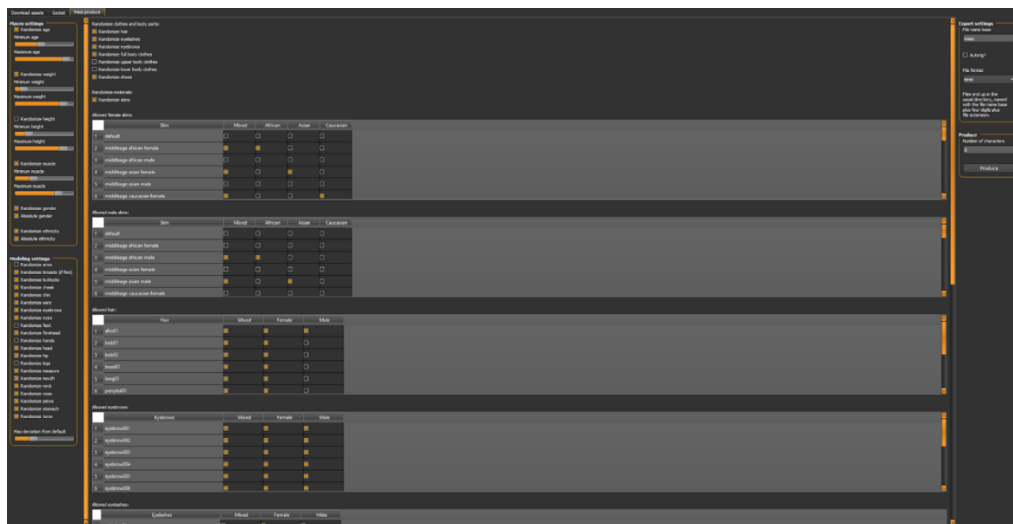


Figure 12: Mass Produce Plugin Page

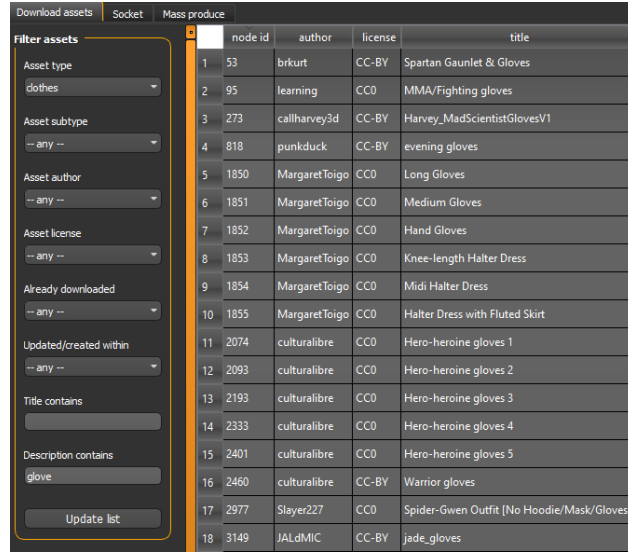


Figure 13: MakeHuman Page to Download Assets

For the loading and random generation of the human models, functions within the scene controller script were written and are shown in Figure 14. The model loading method was provided by Pranav Dhulipala; this script function pulls the models from the Unity Assets and loads each of their names into an array of a size specified by the user. Once in the human model array as elements, the model generation function will destroy any model within the scene first and will then spawn a random model by index of the array into the scene. The script then runs the model animation subsystem on the spawned model.

```
//GetModels inputs all created MakeHuman models into an array for GenerateRandom to use
//Reference
void GetModels()
{
    DirectoryInfo dir = new DirectoryInfo("Assets/Resources/Models"); //selects directory to grab models from
    FileInfo[] files = dir.GetFiles("*.fbx"); //places all files with name starting with human and ending in .fbx
    //from chosen directory into a files folder

    foreach (FileInfo file in files)
    {
        string name = file.Name.Split('.')[0]; //for each file, disconnect the .fbx from the name
        fileList.Add(name); //and add the model name into the list of file names
    }

    humanModels = fileList.ToArray(); //translates file of models into GameObject Array for use

    //GenerateRandom is used to spawn a randomly selected human model into scene
    //Reference
    void GenerateRandom()
    {
        if (spawned[0] != null) //checks if there is already a spawned model in the scene
        {
            Destroy(spawned[0]); //if there is, delete it before placing a new one
        }

        int HumanIndex = Random.Range(0, humanModels.Length); //selects random index of human model array
        string filename = "Models/HumanModels/" + humanModels[HumanIndex]; //gets human model filename depending on randomly chosen index
        GameObject spawnedModel = Instantiate(Resources.Load<GameObject>(filename)); //places chosen model in scene
        spawned[0] = spawnedModel; //indicates a new model is spawned
        PlayAnimation(spawnedModel); //calls PlayAnimation function
        //Synth.OnSceneChange();
    }
}
```

Figure 14: C# Script Functions for Loading and Spawning Human Models

2.4.3 *Subsystem Validation*

The parts of the human model generation subsystem were validated separately as each part was created. The exporting of the models into the Unity Assets folder was first validated, ensuring that the environment had access to the model files that got imported from MakeHuman and that the models could be manually spawned into the scene. Scripts were validated with the newly imported models, testing the loading of the models' names into the array as elements and the spawning of the models. The spawned models were observed to ensure that all loaded models could be spawned into the environment sequentially and that they were destroyed at each call of the function. The subsystem itself was also validated in tandem with the model animation subsystem, validating the spawning, animation, and destruction of each model with each call of the script function.

2.5 **Model Animation Subsystem**

2.5.1 *Subsystem Introduction*

The model animation subsystem is responsible for placing the proper animation components on the imported model and randomly selecting a gesture animation to be performed. This subsystem has been implemented through script and acts only after the gesture data collection subsystem has been completed. This subsystem works in tandem with the human model generation and data capturing/collection subsystems, as once an animation is applied to a randomly spawned human model, an image is taken almost exactly at the same time. The model animation subsystem recognizes which dataset the user is looking to replicate (Sign Language for Numbers, American Sign Language, HANDS, or a custom-built dataset) and pulls only the animation clips included in that dataset, selecting clips to play until all clips have been displayed a user selected maximum number of times.

2.5.2 Subsystem Details

As mentioned in the validation process of the gesture data collection subsystem, to play an animation, the model must have an Animator component located on its lowerarm02.L bone as well as a selected Animation Controller, which houses the animation clips that are to be applied to the model. Initially in testing using one additional MakeHuman model, adding an Animator component to the lowerarm02 bone and an Animation Controller for the Animator to use would be done manually. A singular animation clip would be placed manually into the Animation Controller to test performance as shown in Figure 15.

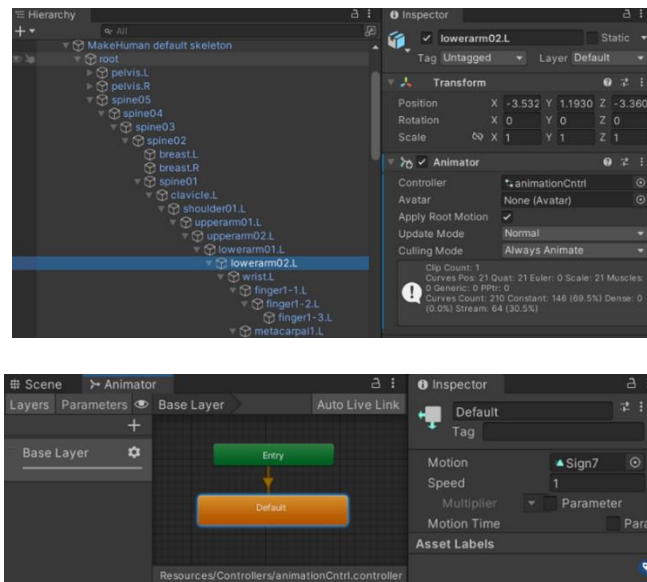


Figure 15: Manual Placement of Animator and Controller with One Animation Clip

However, this approach had to be automated after the Human Model Generation subsystem was implemented since freshly spawned models are being instantiated consecutively into the scene without these components on a constant time interval. To begin implementation of this change, before the Unity scene is run, there must be unique Animation Controllers for each of the datasets being replicated. Each of these animation controllers are labeled according to the

dataset they represent and house all animation clips required to build the intended dataset, as shown in Figure 16. It was also necessary to speed up the performance of the gesture animations by at least 100x to decrease the time it takes to play all animations required to completely generate a virtual training set.

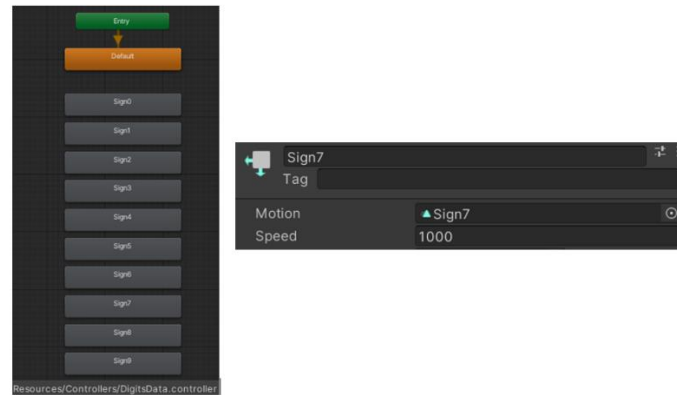


Figure 16: Dataset Specific Animation Controller Containing all Sped Up Gesture Clips

The next automation steps must be implemented through script. Once a MakeHuman model has been instantiated into the scene, the first scripted step is to locate the lowerarm02.L bone as shown in Figure 17.

```
//Indexes through the models bone hierarchy until the lowerarm02 bone is reached
GameObject rig = model.transform.Find("MakeHuman default skeleton").gameObject;
GameObject root = rig.transform.GetChild(0).gameObject;
GameObject spine5 = root.transform.GetChild(2).gameObject;
GameObject spine4 = spine5.transform.GetChild(0).gameObject;
GameObject spine3 = spine4.transform.GetChild(0).gameObject;
GameObject spine2 = spine3.transform.GetChild(0).gameObject;
GameObject spine1 = spine2.transform.GetChild(2).gameObject;
GameObject clavicle = spine1.transform.GetChild(0).gameObject;
GameObject shoulder = clavicle.transform.GetChild(0).gameObject;
GameObject uparm1 = shoulder.transform.GetChild(0).gameObject;
GameObject uparm2 = uparm1.transform.GetChild(0).gameObject;
GameObject lowarm1 = uparm2.transform.GetChild(0).gameObject;
GameObject lowarm2 = lowarm1.transform.GetChild(0).gameObject;
```

Figure 17: Scripting Search for Lowerarm02 Bone of MakeHuman Model

Once located, it is possible to add the Animator component to the root bone of motion (lowerarm02.L) and enable the component. Next, depending on user input of the dataset they

wish to replicate, the correct Animation Controller is placed on the Animator. Playing an animation consists of selecting a random number/string index, where each number/character represents an animation. Depending on the character chosen, a distinct animation clip is played and a counter keeping track of the number of times each gesture has been performed is incremented. A portion of this scripted process is displayed in Figure 18.

```
else if (ChooseDataset == 1)
{
    //using st to pick random character A-Z as well as s for Space and n for Nothing
    int indexNum = Random.Range(0, st.Length); //chooses random index of string
    char randChar = st[indexNum]; //assigns character at that chosen index to randChar variable
    IncreaseCount(randChar, ChooseDataset); //Calls IncreaseCount, which keeps track of how many times each animation has played

    //Places animation controller containing alphabet animations onto lowerarm animator component
    animatorArm.runtimeAnimatorController = (RuntimeAnimatorController)AssetDatabase.LoadAssetAtPath("Assets/Resources/Controllers/AlphabetData.controller")
    //These sections determine camera placement and animation to play based on chosen character
    if (randChar != 's' & randChar != 'n')
    {
        animatorArm.Play("Sign" + randChar); //if not space or nothing, play ..., SignB, SignC, etc...
    }
    else if (randChar == 's')
    {
        animatorArm.Play("SignSPACE"); //if space is chosen, play signSpace
    }
    else
    {
        Destroy(spawned[0]); //if nothing is chosen, delete the model from the scene
    }
    int letterCount = DisplayCount(randChar, ChooseDataset); //Calls DisplayCount to read the value of the counter for a given gesture
}
```

Figure 18: Alphabet Dataset Example for Randomly Selecting and Playing an Animation Clip

Optimization for the time it would take to build the entire datasets was implemented by removing the animation from the string of potential choices to be randomly chosen after it had been played the maximum number of times. Once all gestures had been performed the maximum number of times selected, the data capturing/collection subsystem would stop the running of the Unity environment. Figure 19 displays the coded implementation of this optimization.

```
//OPTIMIZATION: if maxNum images have been taken for a given gesture, we dont want to play that gesture anymore
if (letterCount >= maxNum)
{
    for (int i=0; i<st.Length; i++)
    {
        if (st[i] == randChar) //sift through st and once randChar is found, remove it from the string
        { //so it cannot be selected again in the next updates
            st = st.Remove(i,1);
        }
    }
}
```

Figure 19: Removal of Gesture Choice from String of Gestures

2.5.3 Subsystem Validation

The methodology of the model animation subsystem was validated manually during the validation process of the animation clips produced from the gesture data collection subsystem. The automation of adding components/controllers to sequentially spawned models was then implemented, which was tested and validated on its own using an independent “ModelAnimation” script (which was then incorporated into the finalized SceneController script in the completed Unity environment). Figure 20 depicts the performance of the gesture animation K on 3 unique, consecutively spawned MakeHuman models who did not have Animator components placed before being instantiated into the scene.

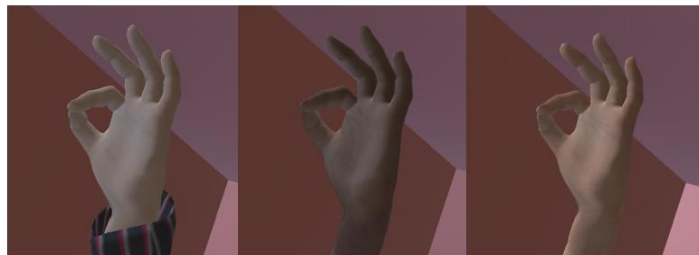
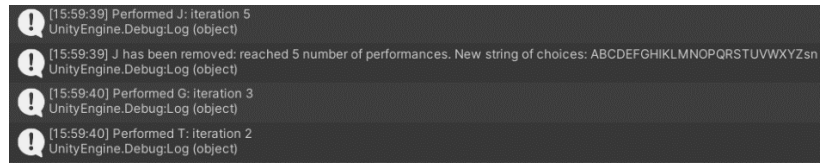


Figure 20: Gestures Performed on Distinct Imported MakeHuman Models Automatically

It was validated that all animations from a given animation controller were able to be selected and played on any “Default” rigged MakeHuman model imported into the scene. It was also validated that each animation was performed a maximum number of times selected by the user by checking the counter of each gesture by the end of the run. Finally, it was validated that the subsystem would remove gestures from the pool of possible animations to play after they had reached the maximum number of performances by running the system with a maximum number of 5 images per gesture and outputting to the console the number of times each gesture animation had been played. Once the console wrote that 5 images had been captured of a certain image, the new string of choices was printed, and it was observed that the choice of the particular gesture

had been removed as seen in Figure 21 and Figure 22. Table 1 summarizes all tests used to validate the model animation subsystem.

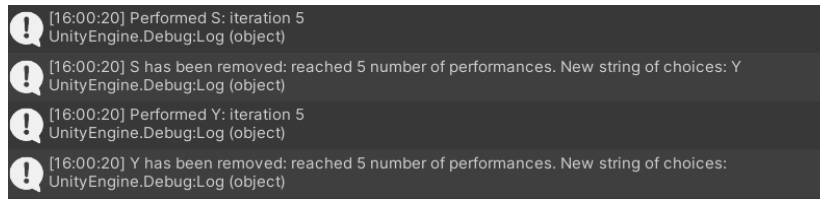


```

[15:59:39] Performed J: iteration 5
UnityEngine.Debug:Log (object)
[15:59:39] J has been removed: reached 5 number of performances. New string of choices: ABCDEFGHIKLMNOPQRSTUVWXYZsn
UnityEngine.Debug:Log (object)
[15:59:40] Performed G: iteration 3
UnityEngine.Debug:Log (object)
[15:59:40] Performed T: iteration 2
UnityEngine.Debug:Log (object)

```

Figure 21: Log After First Letter J Reached Maximum Number of Iterations



```

[16:00:20] Performed S: iteration 5
UnityEngine.Debug:Log (object)
[16:00:20] S has been removed: reached 5 number of performances. New string of choices: Y
UnityEngine.Debug:Log (object)
[16:00:20] Performed Y: iteration 5
UnityEngine.Debug:Log (object)
[16:00:20] Y has been removed: reached 5 number of performances. New string of choices:
UnityEngine.Debug:Log (object)

```

Figure 22: Log After Last Letter Y Reached Maximum Number of Iterations

Table 1: Overview of all Tests Performed to Validate the Model Animation Subsystem

Test Name	Test Summary	Test Result
Hand Gesture Application	When the Unity environment is run, a singular recorded hand gesture animation can be performed on a MakeHuman model.	Pass
Auto Animator Placement	When the Unity environment is run, each spawned MakeHuman model (with no Animator component on its lowerarm02 bone prior to run) contains an automatically placed (through script) Animator component on the lowerarm02 bone and can perform animations.	Pass
Recorded Gesture Animation Functionality	Each recorded hand gesture animation clip may be applied and performed accurately on a MakeHuman model with a “Default” rig.	Pass
Gesture Performance Amount	Each hand gesture animation clip is performed only until a user-specified number is reached.	Pass
Gesture Performance Optimization	After a hand gesture animation clip is performed the user-specified number of times, it is removed from the potential choices of animation clips to be performed.	Pass

2.6 Data Capturing/Collection Subsystem

2.6.1 Subsystem Introduction

The data capturing/collection subsystem records images of the animated hand of each spawned model after it has performed a random gesture animation. It then stores the image as a .jpg in folders sorted by dataset and gesture using a scripted “TakeImage” coroutine. This subsystem controls the placement of unique cameras in the scene using a “PlaceCamera” method to ensure that the proper camera angle is used to capture an image of the applied gesture. In addition, this subsystem introduces slight randomness in the position of the enabled camera to provide more diverse data. Images are sized to be 512x512 pixels for neural network usage.

2.6.2 Subsystem Details

This subsystem works in tandem with the model animation subsystem, as both the PlaceCamera and TakeImage functions are called directly after a random animation is performed. For this subsystem to function properly, many virtual cameras had to be placed within the scene hierarchy that faced each performed gesture from the correct angle to ensure that the gesture is portrayed as it is meant to be. All cameras are disabled in the scene view at the beginning of the run.

The PlaceCamera script is responsible for enabling the correct camera depending on the gesture that was performed. For instance, gesture G requires a different camera than gesture B as shown in Figure 23. This is because all gestures were recorded using the exact same head mounted LMC positioning and virtual arm placement in the gesture recording stage. Since the gesture G required the hand to be pointing towards the right of the scene view, it was necessary to alter the camera angle when looking at the played animation to make the gesture look as though it was recorded with a horizontally orientation.



Figure 23: Camera Placement for Gesture G vs. B

After the correct camera is enabled, a few random number generators select values within a predetermined range and add the random x, y, and z components to the coordinates of the starting camera position, as shown in Figure 24. By adding randomness in the camera position, it is ensured that the gesture will be viewed from several unique angles and add to the diversity of our data.

```
//If ASL for Numbers is being run,
if (dataset == 0)
{
    digitCam.enabled = true; //enable digitCam
    Vector3 startingPos = new Vector3(-13.58f, 13.79f, 110.88f); //assign its starting position to a variable
    float x_pos_change = Random.Range(-.35f, .35f);
    float y_pos_change = Random.Range(-.35f, .35f); //select random adjustments in the x,y,z directions
    float z_pos_change = Random.Range(-.35f, .35f);
    //place camera at its starting position so it is moved from here every iteration
    digitCam.transform.localPosition = startingPos;
    //place camera at starting position + alterations in x,y,z directions
    digitCam.transform.localPosition = startingPos + new Vector3(x_pos_change, y_pos_change, z_pos_change);
}
```

Figure 24: Scripting Randomness in Camera Position

The TakeImage coroutine is responsible for recording a screenshot of the gesture performance in game view, naming the output file, and designating the output folder path depending on the chosen dataset and gesture animation. The Sign Language for Digits dataset named its files using the structure: “name of the gesture_image number of that gesture”. The American Sign Language dataset only named images according to the image number. Depending on the dataset selected for replication using our system, the TakeImage coroutine identifies the

correct naming convention, names each image, and determines where to store the image depending on the signed gesture. A portion of the TakeImage coroutine is displayed in Figure 25.

```
IEnumerator TakeImage(float delayTime, char letter, int gestureNum, int dataset)
{
    int gesturesDone = 0;
    string folderPath;
    string filename = $"{gestureNum.ToString().PadLeft(5, '0')}.jpg"; //naming output file as dataset we are replicating has
    yield return new WaitForSeconds(delayTime); //again delays so animation can play out before screen capture
    if (dataset == 1)
    {
        folderPath = Directory.GetCurrentDirectory() + $"/AmericanSignLanguage/Train";
        //these next if-else statements decide where to store the output images depending on the character being animated
        if (letter != 's' & letter != 'n')
        {
            folderPath = Directory.GetCurrentDirectory() + $"/AmericanSignLanguage/Train/{letter.ToString()}";
        }
        else if (letter == 's')
        {
            folderPath = Directory.GetCurrentDirectory() + $"/AmericanSignLanguage/Train/Space";
        }
        else
        {
            folderPath = Directory.GetCurrentDirectory() + $"/AmericanSignLanguage/Train/Nothing";
        }
    }
}
```

Figure 25: TakeImage File Naming, Output Folder Designation, and Delay

Before a screenshot is taken using the enabled camera from the PlaceCamera script, the TakeImage coroutine introduces a slight delay. This delay is necessary because all animation clips begin in the same position, but over time move differently to form the final gesture sign. If a screenshot were recorded at the instant that a gesture animation clip was played, all images would resemble the sign five as that is the starting hand position of every created gesture clip. However, waiting for the gesture animation to play out in real time would take anywhere from 2-5 seconds depending on the complexity of the gesture. This would be hugely detrimental in the time it would take to create an entire dataset of 12,000 images per gesture. Instead, each animation clip was sped up by a factor of at 1000 and the TakeImage coroutine implemented a delay of .02 seconds so that it would be ensured that each gesture image was taken with the final hand position in place. After the delay, the screenshot is taken and stored according to the settings discussed previously. Finally, the system checks the number of screenshots that each gesture has stored and once all gestures have been recorded the maximum number of times, the script stops the Unity environment from running. Figure 26 shows how this is implemented.

```

//sifts through alphaCounter to check if all gestures have maxNum images
foreach (int i in alphaCounter)
{
    if (i >= maxNum)
    {
        gesturesDone += 1;
    }
}

//if all gestures have maxNum images each,
if (gesturesDone == 28) //28 for 26 letters, 1 space, 1 nothing
{
    EditorApplication.isPlaying = false; //stops Unity player
}

```

Figure 26: Alphabet Dataset Example of Dataset Completion/Environment Exit

2.6.3 Subsystem Validation

The validation of the data capturing/collection subsystem consisted of several tests. First, it was ensured that the images were taken at the correct time after a gesture animation was performed, resulting in photos of the gestures looking as intended (as explained in the previous section). It was determined that by delaying .02 units of time after the gesture was performed (while gestures were sped up 1000x), all screenshots were successful in recording data only after the full animation clip played out and the hand was in its final position, as seen in Figure 27. This resulted in accurate gesture images and a decreased amount of time required to play/record all gesture animations the maximum number of times.

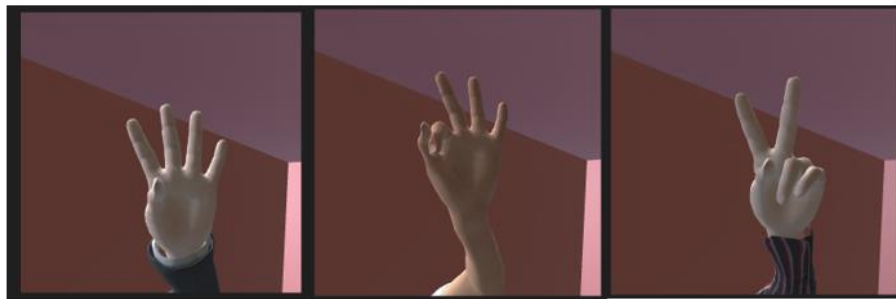


Figure 27: Example Validation for Images of Completed Gestures

It was also validated that each gesture image was stored in the correct folder according to the dataset it belonged and named following the same convention as the real dataset used. To test

each of these, the system was run using a maximum number of 5 images per gesture for all desired datasets. The system was allowed to play until it completed all required gestures and stopped itself. Referring to the folders that contained the output files, it was observed that each labeled folder contained images of the gesture that it specified (e.g., folder labeled one stored all images of one being portrayed). In addition, it was validated that each image was named correctly (e.g., images of one being portrayed were named “one...” for Sign Language for Numbers) and included a counter indicating the image number (up to 5 in this case), as demonstrated in Figure 28.

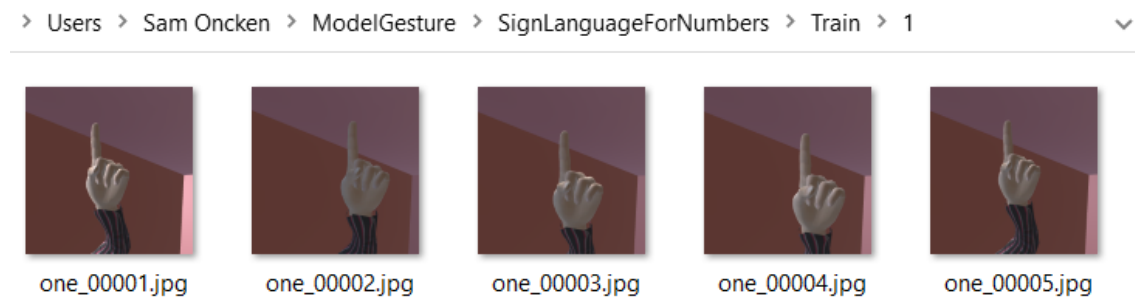


Figure 28: Example Validation of Images Properly Named and Stored

Finally, it was validated that the subsystem would stop the running of the Unity environment once all desired gestures were performed. This was tested by running the scene multiple times, each with a different number of desired images per gesture. During each run, an output message was displayed in the console once all gestures had been performed the maximum number of times as specified by the user. After this condition is satisfied, it was verified that the Unity environment would exit runtime. Referring to the dataset storage location, it was confirmed that all gesture folders contained the desired amount of gesture images, proving that the Unity environment stopped its run only after the desired dataset was fully generated. Table 2 summarizes all tests performed to validate the Data Capturing/Collection subsystem.

Table 2: Overview of all Tests Performed to Validate the Data Capturing/Collection Subsystem

Test Name	Test Summary	Test Result
Gesture Image Accuracy	After the Unity environment is run, gestures in each output image are performed accurately and are as desired by the dataset creator.	Pass
Image Storage Accuracy	After the Unity environment is run, each image is stored in the dataset folder that it belongs and is inside of its correctly labeled gesture folder within the dataset.	Pass
Image Naming Accuracy	After running the Unity environment, all gesture images are properly named according to the real gesture dataset that was replicated.	Pass
Exit Functionality	After the Unity environment has performed every gesture within a dataset to the user-specified maximum number, the Unity player stops. Each gesture folder within the replicated dataset contains only the user-specified maximum number of images.	Pass

2.7 Dataset Generation Full System

2.7.1 System Introduction

The full dataset generation Unity environment is the combination of all previously described subsystems with the addition of a few extra customizations for increased diversity in data, which will be explained below. The dataset generation Unity environment is capable of fully replicating a real hand gesture dataset with completely virtual data. The system consists of one scene for recording gesture animations using the LMC and a MakeHuman model, and another scene for importing random models, applying gesture animations, and recording/storing quality images of each hand gesture to form an entire virtual dataset. We have completed virtual dataset replication of an American Sign Language letters dataset (28 gestures), a Sign Language for Numbers dataset (11 gestures), and a HANDS dataset (12 gestures) consisting of 12,000 images per gesture.

2.7.2 *System Details*

As described in each of the subsystem reports in the Methods section of this paper, each subsystem proved to be compatible with one another. All scripting for each of the subsystems including the placement of animation components, playing of random animation clips depending on the dataset selected, placement of virtual cameras, recording/storing of images, and Unity exit functionality have been merged into a single script called “SceneController.cs”. From the SceneController script, users can select the dataset they wish to replicate along with their desired dataset size and can begin dataset generation by playing the Unity scene.

To provide a brief overview of the dataset generation system process after gesture animation clips have been recorded, the first step involves the human model generation subsystem, which imports all rigged MakeHuman models from the assets folder into an array of strings by name. Every 30 frames, a random index of this array is chosen and the name of the model corresponding to this index is instantiated into the scene view of the environment. Next, the model animation subsystem is called, where a random animation from the chosen dataset is applied to the model. Next, the data capturing/collection subsystem is called where (depending on the performed animation) the correct camera is enabled in the scene, the output file name is determined, the folder path is selected, and a screenshot is captured and stored. This process repeats until all animations have been played a user-selected maximum number of times and the Unity environment stops running. At this point, the finalized dataset is created.

2.7.3 *Additional System Characteristics*

2.7.3.1 Randomization in Background Conditions

Background image randomization has been incorporated into our environment to increase diversity in our virtual datasets. To incorporate the written background randomization script into

our virtual dataset generation environment, wall objects were created around the location where the models would be spawned. By placing the wall objects into the public variable “Walls” array within a created background randomization script, upon running the environment, each wall will appear as a random image from the Describable Textures Dataset [15], as shown in Figure 29.



Figure 29: Background Image Randomization within Images of Gesture A

2.7.3.2 Randomization in Lighting Conditions

The final environment also includes a lighting condition randomization feature, where the intensity of each of the two light sources within the scene hierarchy are randomly set after each iteration of a gesture animation playing. The goal is to replicate real-world scenarios as best as possible, where lighting conditions are not always optimal, and hands might appear darker or brighter than desired as displayed by Figure 30. We plan for a gesture recognition neural network trained using our synthetic data to recognize images in various conditions.



Figure 30: Alterations in Lighting from Image 2 vs. Image 5 on Gesture 1

2.7.4 System Validation

The final system was validated first by running the Unity environment under various user-selected conditions. The Unity dataset generation system was tested using the Sign Language for Numbers dataset with a selection of 5, 10, 50, 1,500, and 12,000 images per gesture. After each run, it was validated that each gesture image was accurately recorded, stored in the correct labeled gesture folder within the dataset folder, and was named according to the convention of the real gesture dataset that was replicated. It was also validated that each gesture image clearly contained the hand gesture without too much hand/skin cut out of frame. After the Sign Language for Numbers dataset passed all validation tests, the same trials were run while replicating the American Sign Language alphabet and the HANDS datasets. For both additional dataset replication cases, all tests passed as all images were accurately taken of each gesture and stored in the correct folder paths, as displayed in Figure 31 with the gesture for the letter T. The 12,000 image per gesture dataset test is used as the final form of the replicated synthetic dataset set that is experimented with in training and testing.

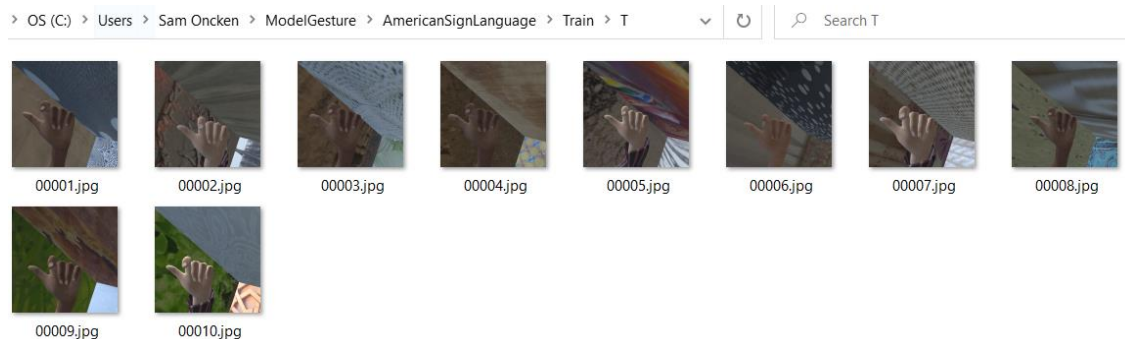


Figure 31: Validation Example - Letter T Performed and Stored 10 Times

The scene customizations were also validated by running a “RandomTexture.cs” script (random background image script) alone to see whether each wall changed appearance. While running both scripts (RandomTexture and SceneController) at the same time, it was determined

that the best rate at which the scene background should change was .12 units of time, which allowed enough time for a new background to appear after each iteration of a model being spawned.

Finally, the light source randomization was validated by inspecting each of the output images of a certain gesture class to make sure that images were not all subject to the same light source intensity. Table 3 summarizes all tests performed to validate the Dataset Generation System.

Table 3: Overview of Tests Performed to Validate the Dataset Generation System

Test Name	Test Summary	Test Result
Dataset Generation	After each run of the Unity environment, all gestures within the selected dataset being replicated are performed the user-selected maximum number of times and all images are stored correctly with correct naming conventions.	Pass
Background Randomness	During the running of the Unity environment, each plane surrounding the MakeHuman model changes appearance upon a new character being spawned.	Pass
Light Intensity Randomness	During the running of the Unity environment, light source intensities change every time a new MakeHuman model is spawned into the scene. Output images contain noticeably different lighting/shadow characteristics.	Pass

2.7.5 System Conclusion

The full system passed all validation tests, proving that all previously designed subsystems were integrated successfully, and the final dataset generation system had been completed. The finalized system was used to generate fully virtual versions of the Sign Language for Numbers dataset, the American Sign Language dataset, and the HANDS dataset, containing 132,000, 336,000, and 144,000 images respectively.

3. RESULTS

The virtual dataset generation Unity environment has been used to fully replicate three real hand gesture datasets of static gestures: American Sign Language, Sign Language for Numbers, and HANDS Datasets. The created virtual datasets are currently being used to train various gesture recognition models either independently or in tandem with the existing real data to evaluate how useful synthetic data can be in image classification and object detection.

3.1 Synthetic Dataset Generation/Replication

3.1.1 American Sign Language Dataset Replication

The American Sign Language Dataset is comprised of 28 distinct, static gestures: 26 letters of the English alphabet, a gesture for “Space”, and a gesture for “Nothing”. A fully synthetic version of the American Sign Language Dataset has been generated. Table 4 depicts each real dataset hand gesture in comparison with the synthetically generated version.

Table 4: Comparison of Real and Synthetic Gestures for American Sign Language Dataset.





Gesture C



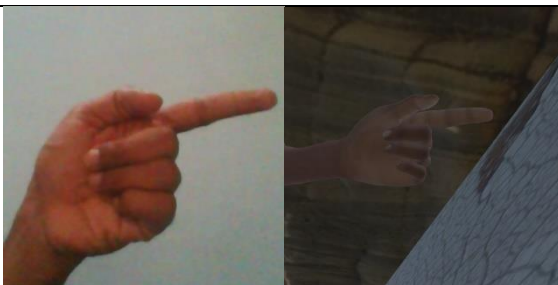
Gesture D



Gesture E



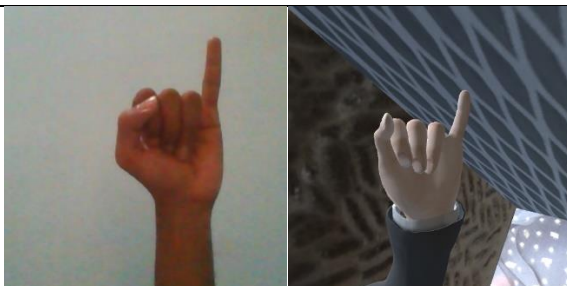
Gesture F



Gesture G



Gesture H



Gesture I



Gesture J



Gesture K



Gesture L



Gesture M



Gesture N



Gesture "Nothing"



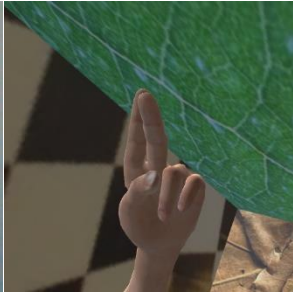
Gesture O



Gesture P



Gesture Q



Gesture R



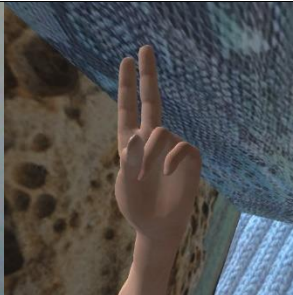
Gesture S



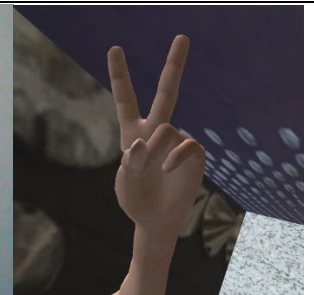
Gesture "Space"



Gesture T



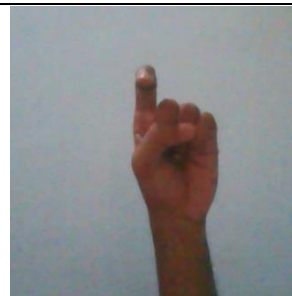
Gesture U



Gesture V



Gesture W



Gesture X



Using the virtual environment developed in this research, a fully synthetic version of the American Sign Language dataset has been created. This virtual dataset is comprised of 336,000 images (12,000 images per gesture). Upon inspection of the example images in the table above, the gestures for the letter “D” are flipped in hand performance from real to synthetic data. All gestures implemented in the virtual environment were recorded using a left hand and therefore are stored as left-handed gestures. To account for this mismatch in this dataset or any other dataset being replicated, gesture images are mirrored at random during image preprocessing in the training of gesture recognition neural networks. By flipping images at random, the final gesture recognition model should recognize both right and left-handed gestures.

Certain hand movements were difficult to track/depict using the Leap Motion Controller, which led to gestures not appearing exactly as desired. . For example, looking at the gesture “W”, the pinky and thumb do not touch as they do in the real dataset image. This is a result of LMC’s tracking inaccuracy. A recurring issue faced in the recording of many gestures was the inability for the thumb to bend towards the palm of the hand. Fixing this issue was out of the scope of this project as it is a Leap Motion tracking/device issue. As a result, the recording of gestures was performed to the best accuracy as possible, and the screenshotted images have been stored and analyzed in a later section.

The largest inaccuracy in gesture recording can be seen in letters “M” and “N”. Both gestures required that the fingers bend and completely cover the palm region of the hand. This motion was not possible to obtain using the LMC, which resulted in poor gesture performance quality. To replicate the real gesture performance as closely as possible, virtual cameras were situated in specific locations so that the perspective makes it seem like the fingers are covering the palm.

There are a few more gestures in this dataset that were challenging to record due to LMC tracking errors, many of which are due to the problems listed above. A few examples include gestures “A”, “E”, and “S”, which all look very similar virtually due to these inaccuracies.

3.1.2 Sign Language for Numbers Dataset Replication

The Sign Language for Numbers Dataset is comprised of 11 distinct, static gestures: numbers 0-9 and a gesture for “unknown”. A fully synthetic Sign Language for Numbers Dataset has been generated using the Unity hand gesture dataset generation environment. Table 5 depicts each gesture from the real dataset in comparison to the synthetically generated gestures.

Table 5: Comparison of Real and Synthetic Gestures for Sign Language for Numbers Dataset.

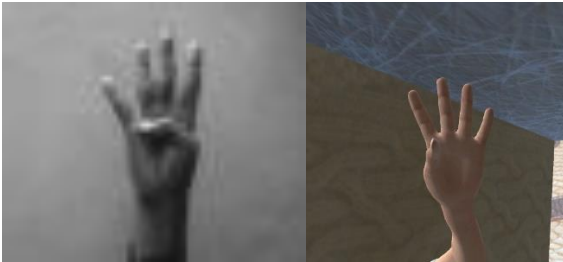
 <p>Gesture 0</p>	 <p>Gesture 1</p>
--	---



Gesture 2



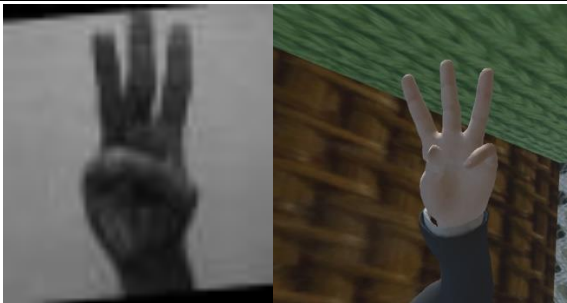
Gesture 3



Gesture 4



Gesture 5



Gesture 6



Gesture 7



Gesture 8



Gesture 9



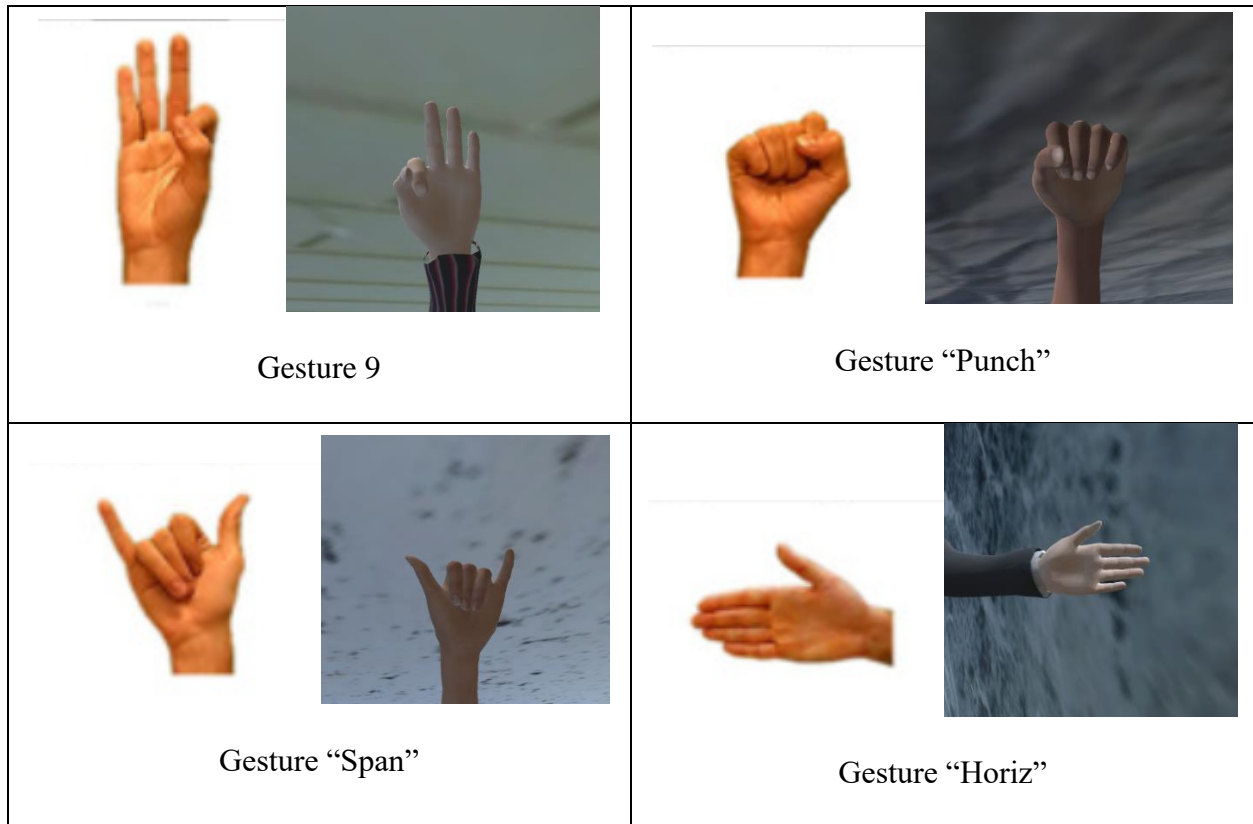
Using the virtual dataset generation environment, a synthetic Sign Language for Numbers dataset has been created, consisting of 132,000 images (12,000 per gesture). Once again, the fact that the virtual images are left-handed gestures while the real images are right-handed gestures is irrelevant as a random number of synthetic images will be flipped during the image preprocessing in the training of a gesture recognition neural network for the model to recognize both left and right-handed gestures. In addition, the LMC tracking inaccuracies discussed in the American Sign Language Dataset Replication section caused problems for a few gestures such as the numbers “6” and “8”. Nevertheless, the recognition accuracies of gesture recognition models trained on this dataset will be evaluated in a future section of this paper.

3.1.3 *HANDS Dataset Replication*

The HANDS Dataset is comprised of 15 distinct, static gestures. However, only 12 of these gestures can be performed with one hand. These 12 gestures have been replicated synthetically and include: numbers 0-9, a gesture for “Punch”, a gesture for “Horiz”, and a gesture for “Span”. Table 6 compares each real, one-handed gesture from the HANDS Dataset to its synthetically generated counterpart.

Table 6: Comparison of Real and Synthetic Gestures for HANDS Dataset.

  <p>Gesture 1</p>	  <p>Gesture 2</p>
  <p>Gesture 3</p>	  <p>Gesture 4</p>
  <p>Gesture 5</p>	  <p>Gesture 6</p>
  <p>Gesture 7</p>	  <p>Gesture 8</p>



Using the virtual dataset generation environment, a synthetic HANDS dataset has been created, consisting of 144,000 images (12,000 per gesture). New animation recordings were used to depict each gesture within this dataset even though many of the gestures included in the HANDS dataset were similar to those in other completed datasets. This extra work was done to produce more high-quality gesture images. Another reason was to match the gesture performances to those within the real HANDS dataset more accurately as some gestures were performed differently than those within the Sign Language for Numbers dataset. Once again, image flipping will take place in the image preprocessing of training a hand gesture recognition neural network so that the model can recognize both right and left-handed gesture performances.

3.2 Dataset Training and Testing Validation

3.2.1 Synthetic Dataset Validation

Using the generated synthetic datasets, many different CNN models were trained to validate the use of the synthetic datasets in training image classification models. This process was also done to determine the most optimal training method, considering transfer learning, training from scratch, and using or excluding image augmentation. The training and testing results were obtained by our graduate student, Pranav Vaidik Dhulipala, using our system and generated datasets.

For the synthetic data training and validation in Figure 32 (left), a gradual learning curve is seen, indicating a well-trained model that reaches acceptable accuracy metrics. For the real data training and validation in Figure 32 (right), the significantly smaller dataset size led the model to overfit on the data very quickly. This shows the benefit of the virtual environment as the number of images per gesture can easily be increased to expand the dataset significantly.

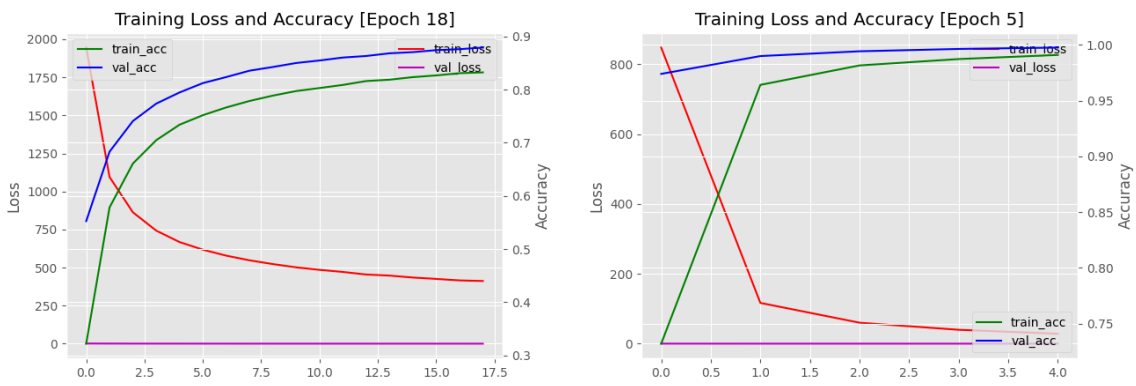


Figure 32: Inception CNN Architecture Trained with Transfer Learning and Image Augmentation for the Synthetic ASL Alphabet Dataset (left) and real ASL Alphabet Dataset (right)

For models trained on images without image augmentation, the general outcome was immediate overfitting of the model due to the lack of complexity in the image, as seen in Figure

33 (left). This lack of complexity also allowed the training and validation accuracy to almost match since the training data less difficult to learn. However, some cases showed different behaviors, like with the ResNet50 CNN architecture in Figure 33 (right) where the model still maintained a gradual learning curve but failed to reach a high enough accuracy. This change in behavior reinforces the idea of requiring extensive analysis of various combinations of methods and data to exhaust all possibilities.

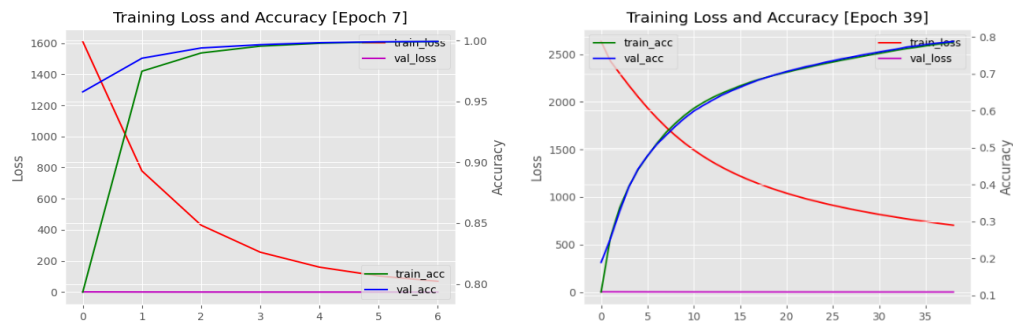


Figure 33: VGG19 CNN Architecture Trained on Synthetic ASL Alphabet Dataset (left) and ResNet50 CNN Architecture on Real ASL Alphabet Dataset (right) with Transfer Learning and Without Image Augmentation

As seen in Figure 34 (left), training with transfer learning again maintained a gradual learning curve that reached acceptable accuracy metrics. When training from scratch as seen in Figure 34 (right), the limitation of using a small dataset is clearly visible as the model quickly overfit. Without the pretrained parameters from ImageNet, which is a multi-million image dataset with 1000 different classes, the model overfits and would struggle to classify any images that differ from the training data.

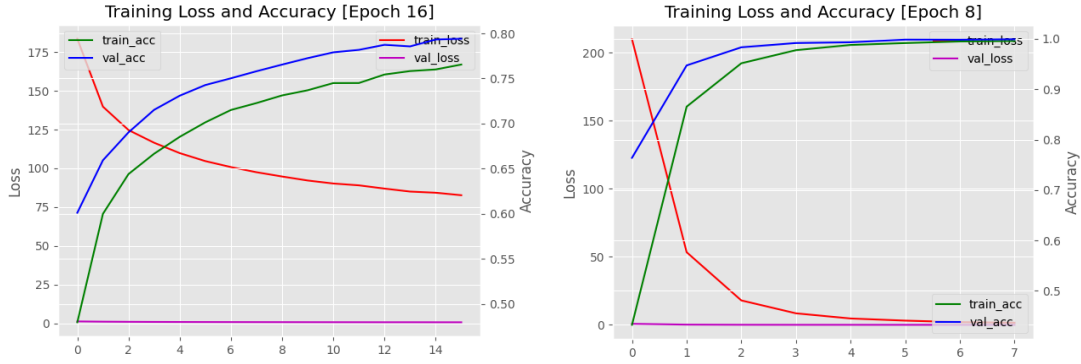


Figure 34: Xception CNN Architecture Trained on Synthetic ASL Numbers Dataset with Transfer Learning and Image Augmentation (left) and Real ASL Numbers Dataset from Scratch with Image Augmentation

From the analysis using various training methods, using transfer learning and image augmentation is determined to be the optimal training method for most model architectures. This was because the models trained using transfer learning and image augmentation generally avoided overfitting and gradually trained to a sufficient image classification accuracy, with most models ending with around 80% or higher validation accuracy. This final training accuracy of 80% or higher on models trained with synthetic data also validate the use of synthetic image datasets for training a CNN.

3.2.2 Optimal CNN Model Architecture Validation

Then, to verify which CNN model architectures performed best with the image data used, further analysis was performed for the following model architectures: Inception, ResNet50, VGG16, VGG19, and Xception. Comparisons between the training for the real and synthetic alphabet dataset are shown in Figures 35, 36, 37, 38 and 39 below. For training with the real alphabet dataset, most of the models trained to 100% accuracy rapidly, which is a sign of overfitting, with just ResNet50 barely reaching 70% validation accuracy after 39 epochs. However, the overfitting is expected, since the real alphabet dataset is much smaller than the

synthetic dataset and lacks complexity. For training with the synthetic alphabet dataset, Inception, ResNet50, VGG19, and Xception trained to near or above 80% validation accuracy, while VGG16 only reached 50% validation accuracy. The models trained gradually, which is likely due to the larger dataset size and increased image complexity.

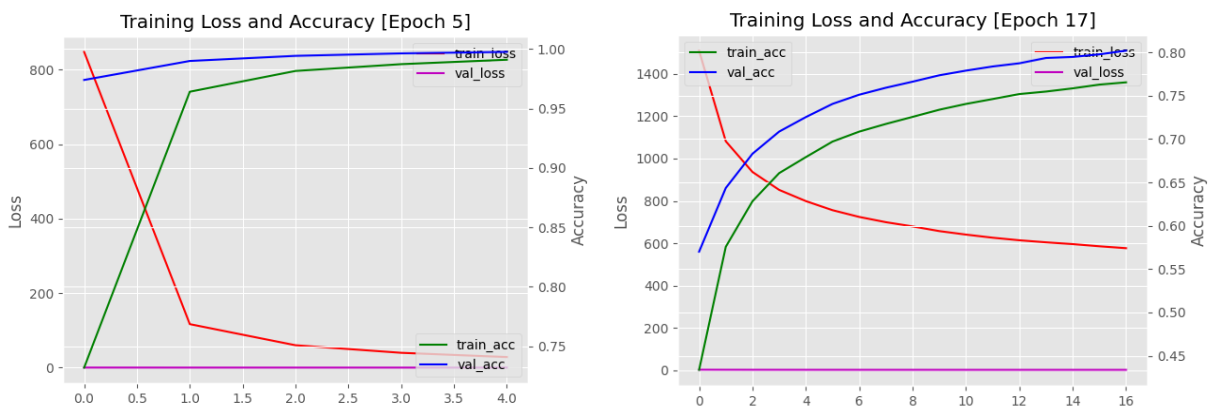


Figure 35: Inception CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

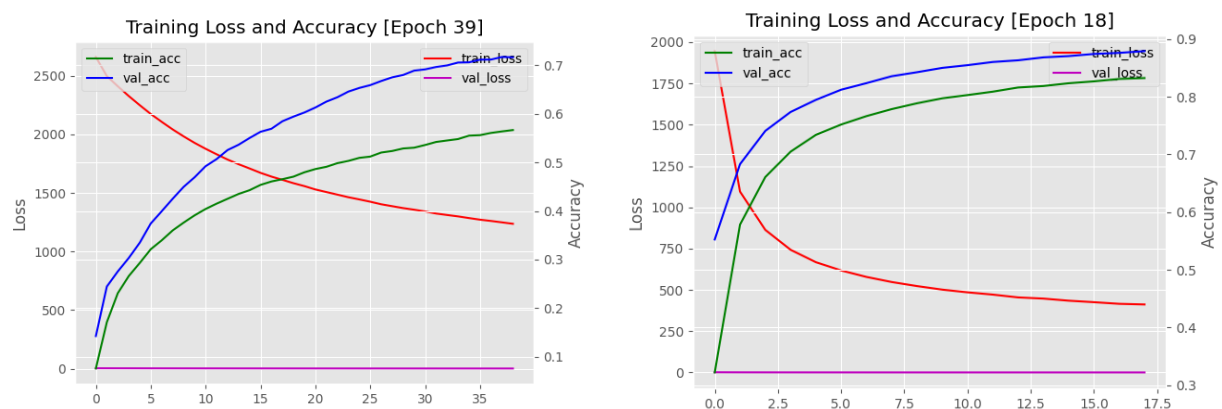


Figure 36: ResNet50 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

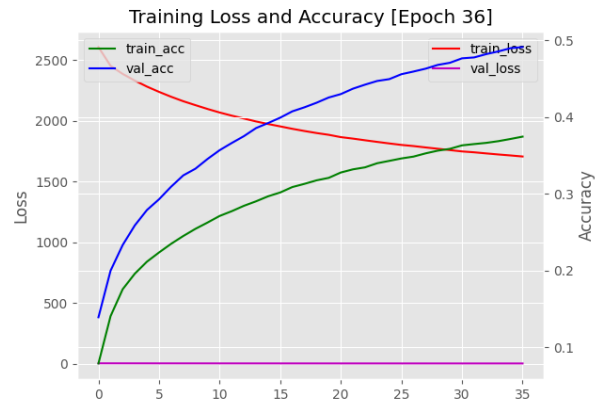


Figure 37: VGG16 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

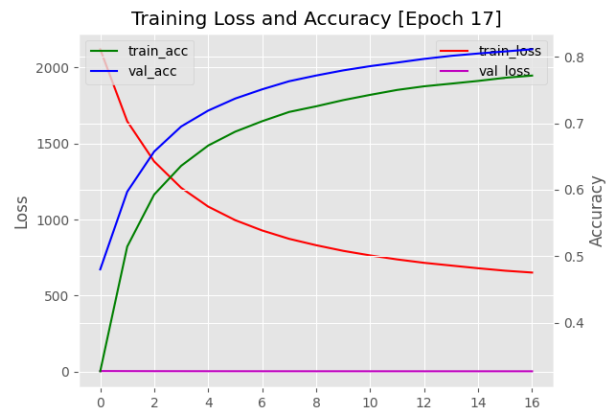


Figure 38: VGG19 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

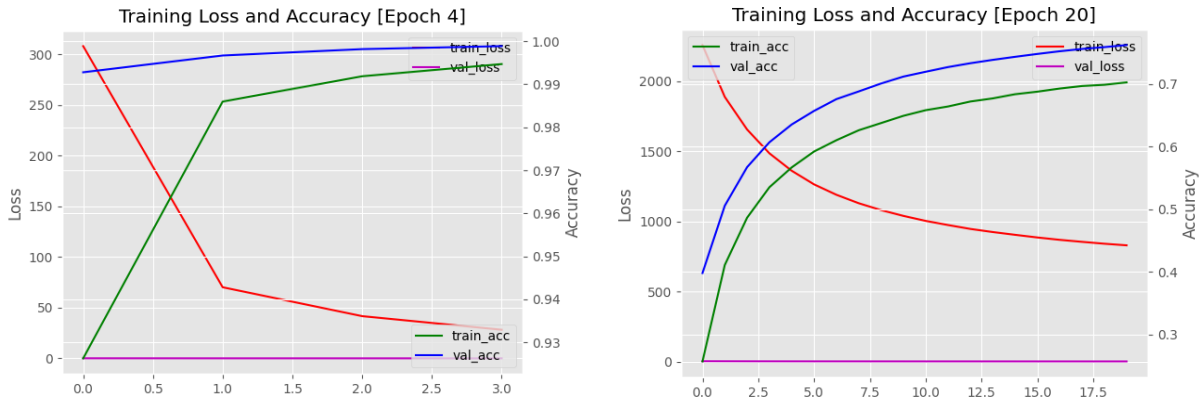


Figure 39: Xception CNN Architecture Trained on Real (left) and Synthetic (right) ASL Alphabet Dataset using Transfer Learning and Image Augmentation

For the alphabet dataset, the models that train best are Inception and VGG19. Both models trained to validation accuracies above 80% for both real and synthetic data, allowing for reliable comparisons between the two in further analysis. The Xception model trained adequately on synthetic data, but the validation accuracy under 80% is not ideal for reliable comparisons. The real training for ResNet50 and synthetic training for VGG16 both had poor training accuracy metrics as opposed to their counterparts, so these models will not be used for future analyses using the alphabet dataset since any comparisons between real and synthetic would be insignificant because of the gap in accuracy between real and synthetic data training.

The same comparisons between the real and synthetic training with the numbers dataset are shown in Figures 40, 41, 42, 43 and 44. For the training with the real numbers dataset, ResNet50 and VGG19 failed to reach 80% validation accuracy, while the other three models reached a sufficient validation accuracy of 80% or higher. For training with the synthetic numbers dataset, ResNet50 failed to train above 50% validation accuracy, while all other models reached 80% or higher.

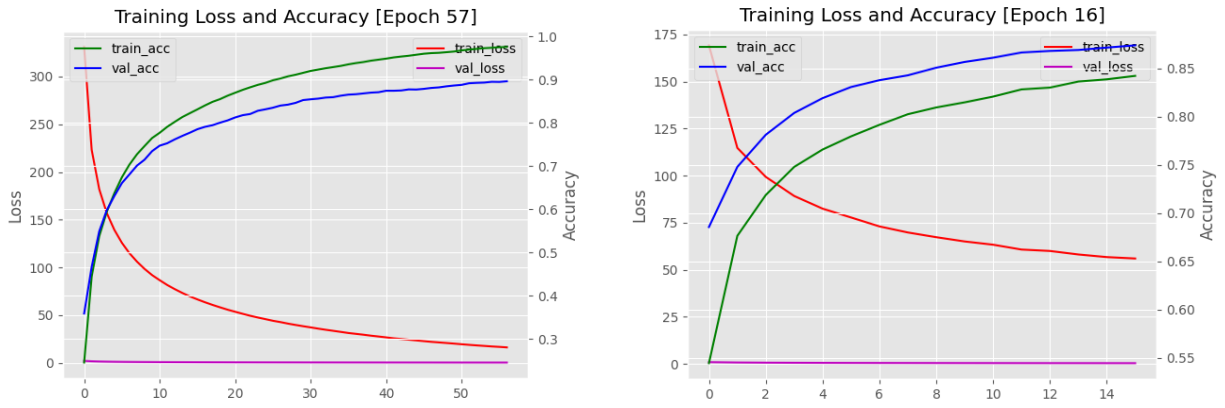


Figure 40: Inception CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

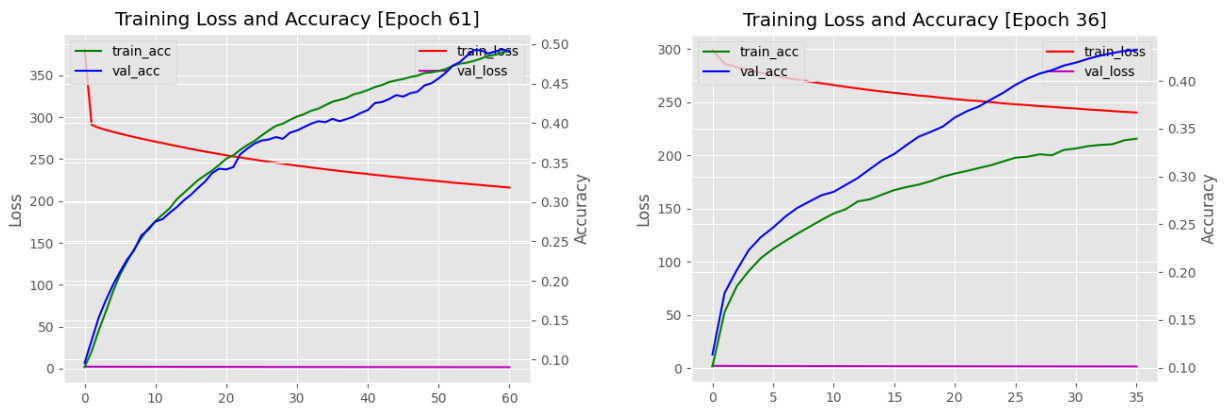


Figure 41: ResNet50 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

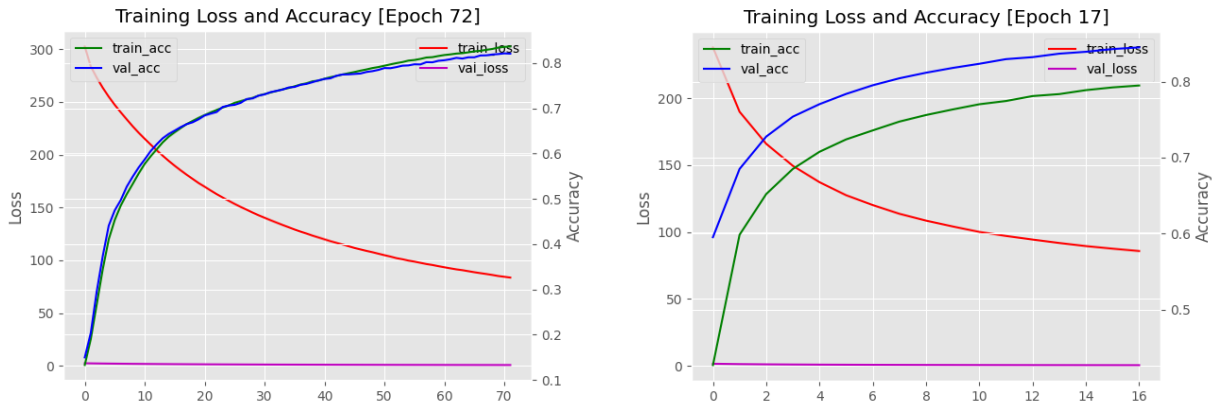


Figure 42: VGG16 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

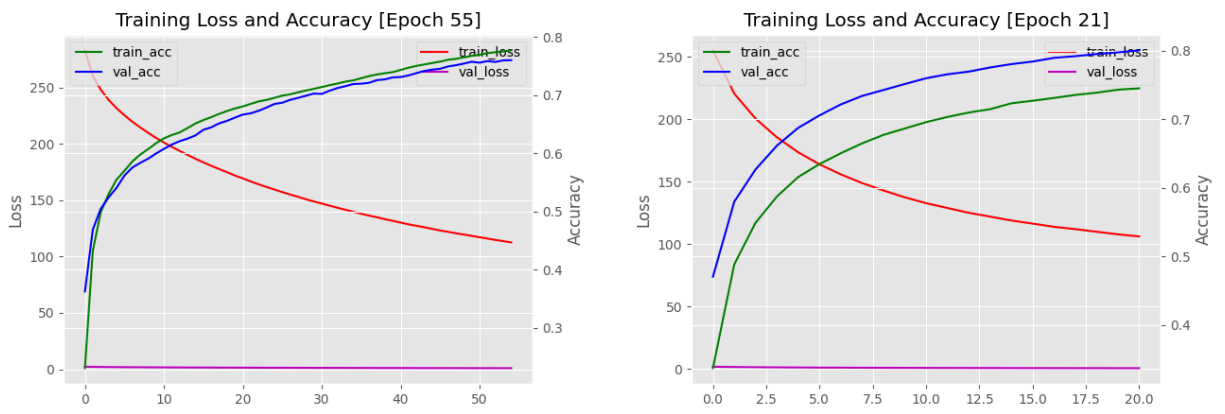


Figure 43: VGG19 CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

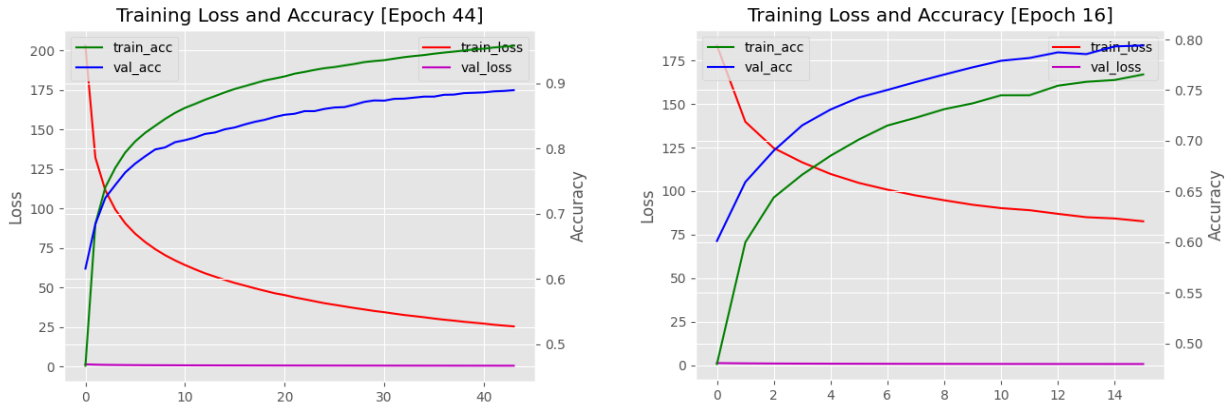


Figure 44: Xception CNN Architecture Trained on Real (left) and Synthetic (right) ASL Numbers Dataset using Transfer Learning and Image Augmentation

For the numbers dataset, the models that train best are Inception, VGG16, and Xception. These models trained to near or above 80% validation accuracy for both real and synthetic data, which allows for reliable comparisons in future analyses. The ResNet50 model struggled to train with both the real and synthetic data, which wouldn't allow for reliable comparisons of classification accuracy using this model. The VGG19 model trained well with the synthetic data but couldn't reach 80% validation accuracy when training on real data, which would also cause unreliable comparisons when using this model. For those reasons, these models will not be used for future analyses using the numbers dataset.

4. CONCLUSION

4.1 Interpreting Results

In this work, we created a virtual environment to replicate existing real image datasets virtually and used both to train convolution neural networks for image gesture classification. It was demonstrated that we could take a set of hand gestures and generate a large synthetic gesture dataset with diverse models and backgrounds, reducing the cost and time needed and completely bypassing the need for diverse participants. With the original real gesture datasets and our synthetic gesture datasets, we trained convolution neural networks. It was demonstrated that some models could successfully train to accuracies above 80% on both the real and synthetic data, proving that the data used for training works successfully with those models. The training using synthetic data showed gradual but sufficient training metrics, as opposed to the rapid training with the real data, which is a sign of overfitting. This is a testament to our ability to choose the size of our synthetic dataset in our virtual environment, simplifying the process of generating datasets of sufficient size for the CNN models.

4.2 Applications

With the growing field of virtual games and the greater importance of virtual society, computer vision within the virtual environment is becoming more prevalent. Our results indicate that training CNNs for image classification in the virtual environment can be done easily using synthetic image data generated with our system. With the tested accuracy of 80% or higher, the models can accurately classify within a virtual environment. The synthetic datasets necessary to train CNNs used for image classification in a virtual environment can easily be generated using

our dataset generation environment, using whatever models, backgrounds, and camera angles desired.

4.3 Future Analysis

Since all models were trained and tested on one set of data (exclusively real or synthetic), further testing is in progress to test the ability to classify real gesture images using models trained on synthetic gesture data. For this testing, models will be trained on varying compositions of real and synthetic data, and then tested on the real data. Models trained on only synthetic data will validate the viability of replacing real datasets with synthetic datasets, and models trained on a combination of the real and synthetic datasets will validate the ability to improve smaller real datasets by augmenting them with synthetic data.

To investigate further into the uses of the synthetic data in computer vision, further research is in progress for object detection with real and synthetic data. Using the trained CNNs, a region-based convolution neural network (R-CNN) will be used to test the ability of a model trained on synthetic data to both localize and classify gestures [16] within a real image from the HANDS dataset. This further analysis along with the continued CNN testing will hopefully validate the use of synthetic image data in computer vision and advance the field of machine learning.

REFERENCES

- [1] J. Nagi et al., “Max-pooling Convolutional Neural Networks for vision-based hand gesture recognition,” IEEE Xplore, 2011. [Online]. Available: <https://ieeexplore.ieee.org/document/6144164/>. [Accessed: 5-Sep-2022].
- [2] J. Yu, M. Qin, and S. Zhou, “Dynamic gesture recognition based on 2D convolutional neural network and feature fusion,” Research Gate, Mar-2022. [Online]. Available: https://www.researchgate.net/publication/359223015_Dynamic_gesture_recognition_based_on_2D_convolutional_neural_network_and_feature_fusion. [Accessed: 5-Sep-2022].
- [3] Karen Simonyan, Andrew Zisserman, “Very Deep Convolution Networks for Large-Scale Image Recognition,” arXiv:1409.1556, 04-Sep-2014. [Online]. Available: <https://arxiv.org/abs/1409.1556>. [Accessed: 20-Dec-2022].
- [4] M. Asadi-Aghbolaghi et al., “A Survey on Deep Learning Based Approaches for Action and Gesture Recognition in Image Sequences,” IEEE Xplore, 2017. [Online]. Available: <https://ieeexplore.ieee.org/document/7961779>. [Accessed: 4-Sep-2022].
- [5] C. C. de Amorim and C. Zanchettin, “ASL-Skeleton3D and ASL-Phono: Two Novel Datasets for the American Sign Language,” arXiv:2201.02065, 06-Jan-2022. [Online]. Available: <https://arxiv.org/abs/2201.02065>. [Accessed: 5-Sep-2022].
- [6] J. Shermeyer, T. Hossler, A. Van Etten, D. Hogan, R. Lewis, and D. Kim, “RarePlanes: Synthetic Data Takes flight,” arXiv:2006.02963, 10-Nov-2020. [Online]. Available: <https://arxiv.org/abs/2006.02963>. [Accessed: 3-Oct-2022].
- [7] S. James, P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis, “Sim-to-real via sim-to-SIM: Data-efficient robotic grasping via randomized-to-canonical adaptation networks,” arXiv:1812.07252, 21-Jul-2019. [Online]. Available: <https://arxiv.org/abs/1812.07252>. [Accessed: 3-Oct-2022].
- [8] S. Hinterstoisser, O. Pauly, H. Heibel, M. Marek, and M. Bokeloh, “An Annotation Saved is an Annotation Earned: Using Fully Synthetic Training for Object Instance Detection,” arXiv:1902.09967, 26-Feb-2019. [Online]. Available: <https://arxiv.org/abs/1902.09967>. [Accessed: 3-Oct-2022].
- [9] Z. Helton, “VR Hand Tracking for Robotics Applications”, unpublished.

- [10] Ultraleap, Mountain View, CA, United States. *Leap_Motion_Controller_Datasheet*. Accessed: 1-Sep-2022. [Online]. Available: https://www.ultraleap.com/datasheets/Leap_Motion_Controller_Datasheet.pdf
- [11] Unity Technologies, “Unity user manual 2021.3 (LTS),” Unity, 2021. [Online]. Available: <https://docs.unity3d.com/Manual/index.html>. [Accessed: 1-Sep-2022].
- [12] M. Khalid, “Sign Language for Numbers”, Kaggle, 2020. [Online]. Available: <https://www.kaggle.com/datasets/muhammadvhalid/sign-language-for-numbers>
- [13] K. Londhe, “American Sign Language”, Kaggle, 2021. [Online]. Available: <https://www.kaggle.com/datasets/kapillondhe/american-sign-language>
- [14] C. Nuzzi, S. Pasinetti, R Pagani, G. Coffetti, G. Sansoni, “HANDS: a Dataset of Static Hand-Gestures for Human-Robot Interaction,” Mendeley Data, V2, 2021. doi: 10.17632/ndrczc35bt.2
- [15] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, A. Vedaldi, “Describing Textures in the Wild,” arXiv:1311.3618, 15-Nov-2013. [Online]. Available: <https://arxiv.org/abs/1311.3618>. [Accessed: 16-Oct-2022].
- [16] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” arXiv:1506.01497, 04-Jun-2015. [Online]. Available: <https://arxiv.org/abs/1506.01497>. [Accessed: 20-Dec-2022].