Hand Gesture Recognition

Samuel Oncken, Steven Claypool

SUBSYSTEM REPORTS

REVISION - 1
28 November 2022

SUBSYSTEM REPORTS
FOR
Hand Gesture Recognition

PREPARED BY:

Team 72 11/28/2022
Author Date

APPROVED BY:

Samuel Oncken 11/28/2022
Project Leader Date

John Lusher, P.E. Date

T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
-	11/28/2022	Samuel Oncken Steven Claypool		Release for Final Report

Table of Contents

List of Tables

List of Figures

1. Introduction

The virtual dataset generation Unity environment that we have created to this point has allowed us to fully replicate two real hand gesture datasets of static gestures: American Sign Language and Sign Language for Numbers. The created system is broken down into gesture data collection, human model generation, model animation, and data capturing/collection subsystems. We have tested each subsystem individually across various Unity scenes to validate that all functionality is as we designed and we have merged all subsystems into a common Unity scene to form our complete virtual hand gesture dataset generation system, which has also been validated. We are currently training various gesture recognition models using our virtual datasets either independently or in tandem with the existing real data to evaluate how useful synthetic data can be in increasing gesture recognition accuracy. In addition, we are exploring additional real datasets that include dynamic gestures and full body images/videos to expand upon our research in the coming semester.

Links to each replicated gesture dataset and background image dataset are found below:

American Sign Language: https://www.kaggle.com/datasets/kapillondhe/american-sign-language
Sign Language for Numbers: https://www.kaggle.com/datasets/muhammadkhalid/sign-language-for-numbers
Describable Textures Dataset: https://www.robots.ox.ac.uk/~vgg/data/dtd/

2. Gesture Data Collection Subsystem Report

2.1. 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 the dataset they want to generate. 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. All instructions for use of this subsystem have been written in the GitHub repository we created to store our project files. The gesture data collection subsystem outputs unique animation clips for future use in the model animation subsystem.

2.2. 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. By storing animation clips separately, we can place them into Animation Controllers and therefore play them at random on imported virtual human models in future subsystems. This subsystem is the root of the system as without the gesture data collection subsystem functioning correctly, we would not be able to create uniquely identifiable gestures and therefore would have nothing to animate models with and record images of. Due to this, this subsystem has been tested and changed various times to ensure ease of use and accurate gesture recordings.

Figure 1: Stored Sign Language for Numbers Gesture Animation Clips

The main challenge with this subsystem has been determining the best method of recording gesture animations while planning for animation application in future subsystems. In the beginning of this project, I had attempted to record gestures (exporting as .fbx files) on the sample hand prefabs provided by Ultraleap. These prefabs contained the necessary scripts already written and configured by the Ultraleap developer team to simplify use. The issue arose when I began recording the transformation data of the hands. 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 the MakeHuman models that I was attempting to apply the animation onto. This incompatibility resulted in the animation clips not being able to play on the MakeHuman models. In addition, when exporting the gesture as a .fbx file, the animation clips that came with them contained re-named bones, where each instance of '.' was renamed as '_' (example "wrist.L" turned into "wrist_L"), which once again resulted in naming incompatibility and animation application issues.

Figure 2: MakeHuman Default Rig Bone Labeling

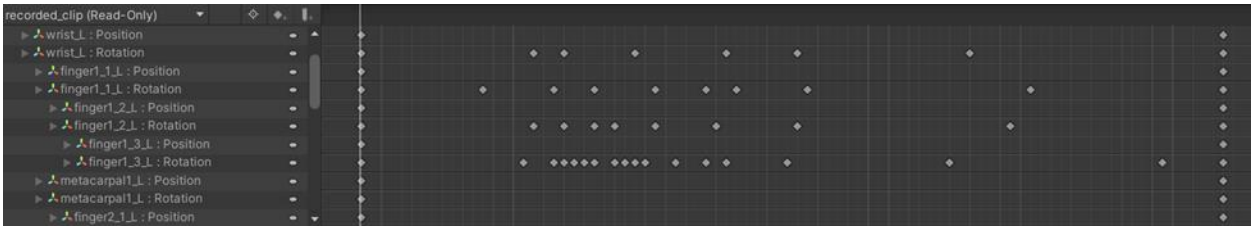


Figure 3: Animation clip using FBX Exporter - Not Compatible

The solution to this problem was to map Leap Motion hand tracking data directly to a MakeHuman model (which took more time to configure correctly) and export as an animation clip alone, which resulted in animation clip keyframes that could be directly applied to freshly imported MakeHuman models.

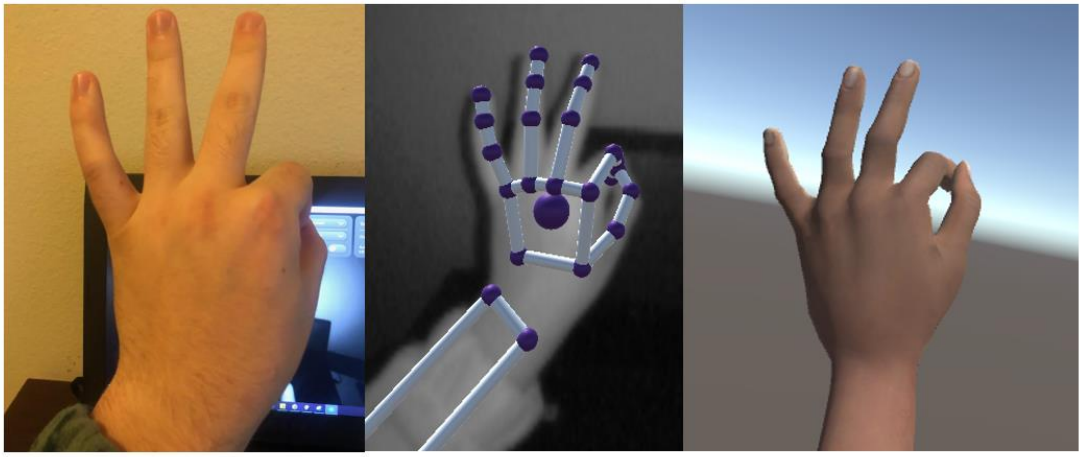


Figure 4: Recording of Animation Clips Directly on MakeHuman Model

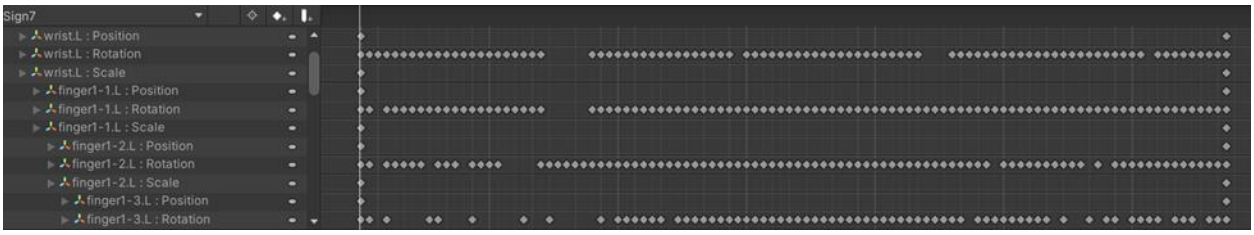


Figure 5: Animation Clip using Finalized Method - Compatible and Accurate

2.3. 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 model. To test the model animation application, I created another scene in Unity where I placed a freshly imported MakeHuman model with a “Default” rig and added an Animator component to the “lowerarm02.L” bone of the model, since each animation was recorded from the lowerarm02. I then placed a sample animation clip into an Animation Controller which controlled that Animator component and ran the scene. Once the animation was able to run smoothly and stop in the final hand gesture position as I had recorded, the subsystem proved to be applicable to future subsystems.

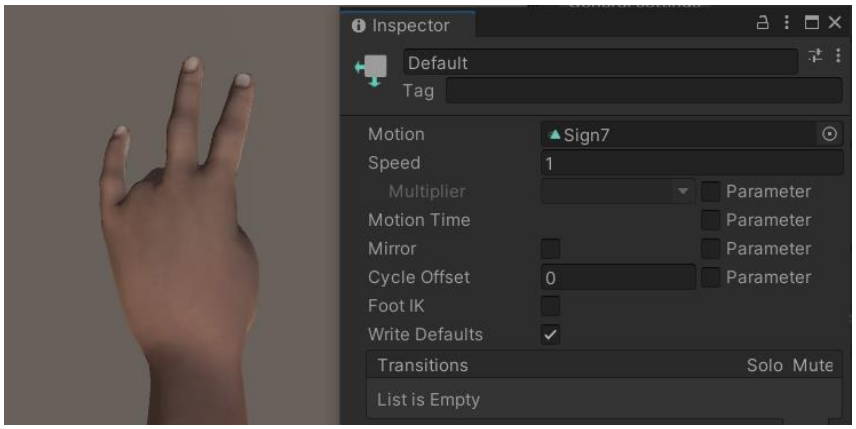


Figure 6: Test Application of Sign7 Gesture Animation Clip

In addition, I validated the use of the Leap Motion Controller within this subsystem by testing various LMC mounting positions under different lighting conditions to see which resulted in the highest tracking accuracy upon inspection. I confirmed that by head mounting the LMC, I was able to achieve the best motion tracking possible for the gestures in datasets that we planned to replicate. Many of the gestures within the American Sign Language and Sign Language for Numbers datasets involved putting one or more fingers down (into palm region) while still holding up others, which I had to ensure the LMC could recreate accurately. In the images below, you can see how finger tracking is easily lost and inaccurate when using the LMC in desktop mounted mode vs. in head mounted mode.

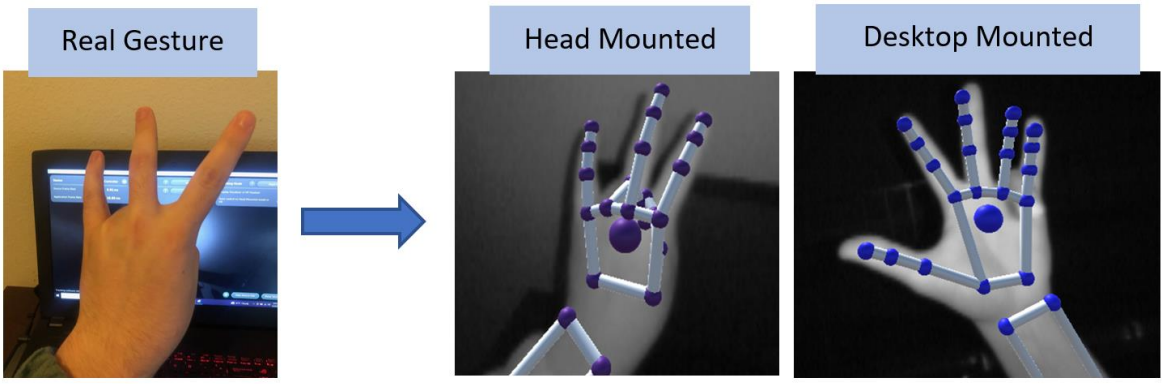


Figure 7: Tracking Accuracy of Head Mounted LMC vs. Desktop Mounted LMC

2.4. Subsystem Conclusion

The gesture data collection subsystem has been tested and validated and has proved to be functioning as intended. This subsystem successfully models tracking data from the user directly onto a MakeHuman model and records and stores the transformation data of the hand bones to usable animation clips that can be applied to future imported models in the subsystems to come.

3. Human Model Generation Subsystem Report

3.1. 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 script function. When the final system is run, the script spawns the model into the scene for the model animation subsystem to manipulate.

3.2. 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 and MakeHuman's code and running into multiple issues with the process, we spoke to our sponsor and determined that it would require extensive work to make such changes and resorted to 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. 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.

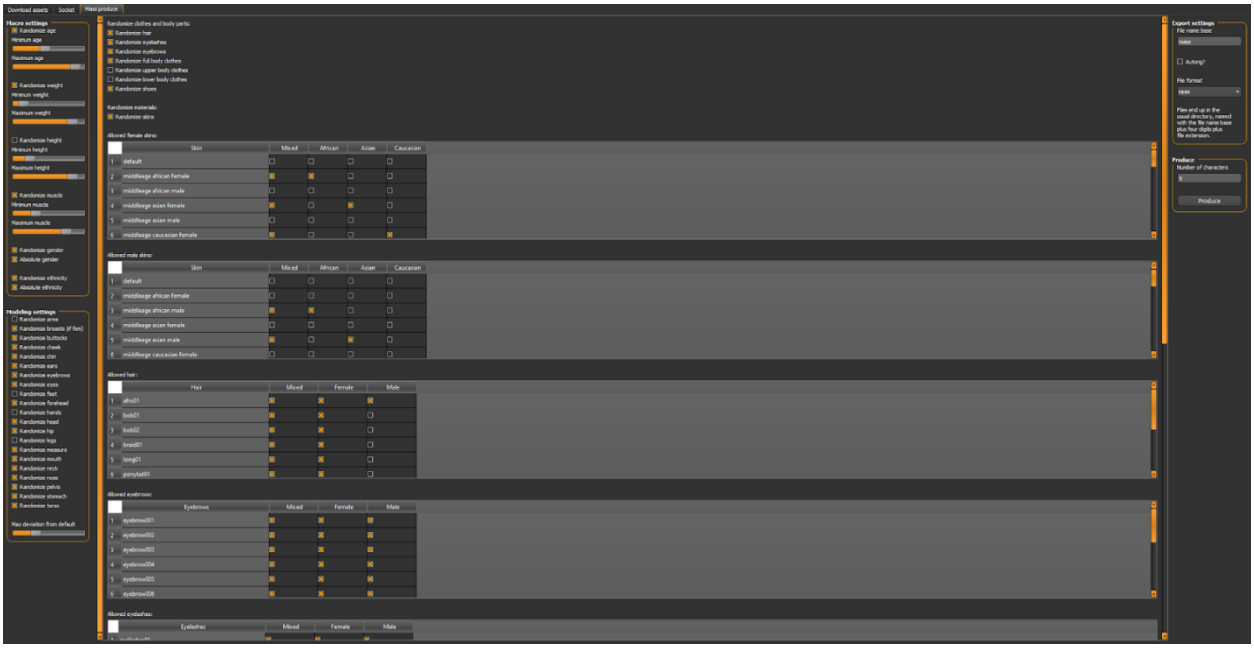
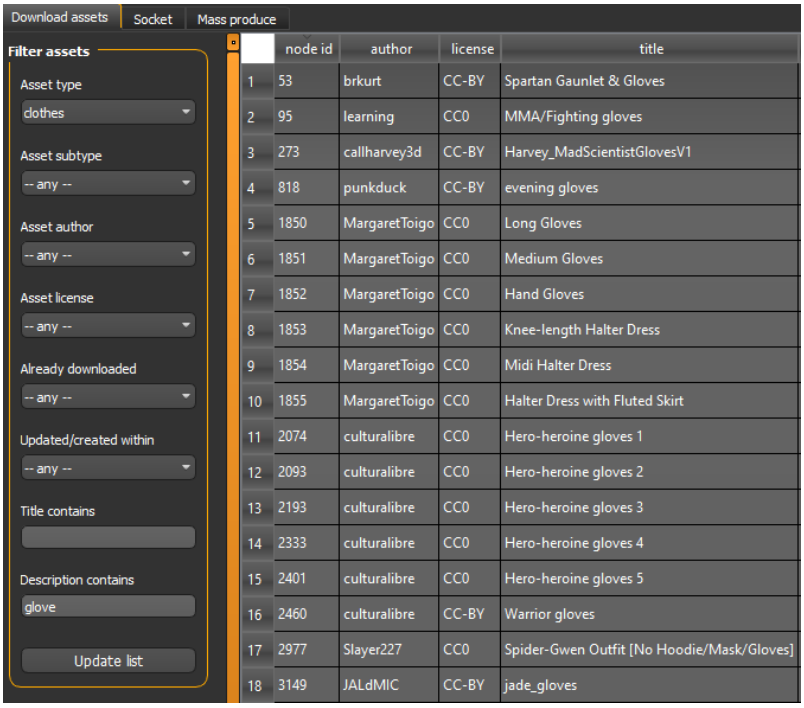


Figure 8: Mass Produce Plugin Page



	node id	author	license	title
1	53	brkurt	CC-BY	Spartan Gaunlet & Gloves
2	95	learning	CC0	MMA/Fighting gloves
3	273	callharvey3d	CC-BY	Harvey_MadScientistGlovesV1
4	818	punkduck	CC-BY	evening gloves
5	1850	MargaretToigo	CC0	Long Gloves
6	1851	MargaretToigo	CC0	Medium Gloves
7	1852	MargaretToigo	CC0	Hand Gloves
8	1853	MargaretToigo	CC0	Knee-length Halter Dress
9	1854	MargaretToigo	CC0	Midi Halter Dress
10	1855	MargaretToigo	CC0	Halter Dress with Fluted Skirt
11	2074	culturalibre	CC0	Hero-heroine gloves 1
12	2093	culturalibre	CC0	Hero-heroine gloves 2
13	2193	culturalibre	CC0	Hero-heroine gloves 3
14	2333	culturalibre	CC0	Hero-heroine gloves 4
15	2401	culturalibre	CC0	Hero-heroine gloves 5
16	2460	culturalibre	CC-BY	Warrior gloves
17	2977	Slayer227	CC0	Spider-Gwen Outfit [No Hoodie/Mask/Gloves]
18	3149	JALdMIC	CC-BY	jade_gloves

Figure 9: MakeHuman Page to Download Assets

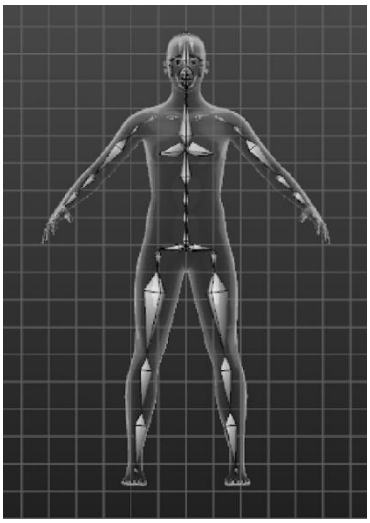


Figure 10: Default Rig for MakeHuman Models

For the loading and random generation of the human models, functions within the scene controller script were written. The model loading script was provided by our sponsor: 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
1 reference
void GetModels()
{
    DirectoryInfo dir = new DirectoryInfo("Assets/Resources/Models");   //selects directory to grab models from
    FileInfo[] files = dir.GetFiles("mass*.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
1 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[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 11: C# Script Functions for Loading and Spawning Human Models

3.3. 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.

3.4. Subsystem Conclusion

The validation of the human model subsystem is completed and is operating as designed. It takes human models imported from MakeHuman, loads each model name into an array within the Unity environment, and spawns a random model picked from the array into the scene sequentially for gesture animation, after which it reruns the script, destroying the old model and restarting the process with a new random model. This proves the subsystem's function and its incorporation with the related model animation subsystem.

4. Model Animation Subsystem Report

4.1. Subsystem Introduction

The model animation subsystem is responsible for placing the proper animation components on the imported model and randomly selecting an animation to play. 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, 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.

4.2. 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, I would simply add an Animator component to the lowerarm02 bone directly, select an Animation Controller for the Animator to use, and add a singular animation clip into the controller to play as shown in the figure below.

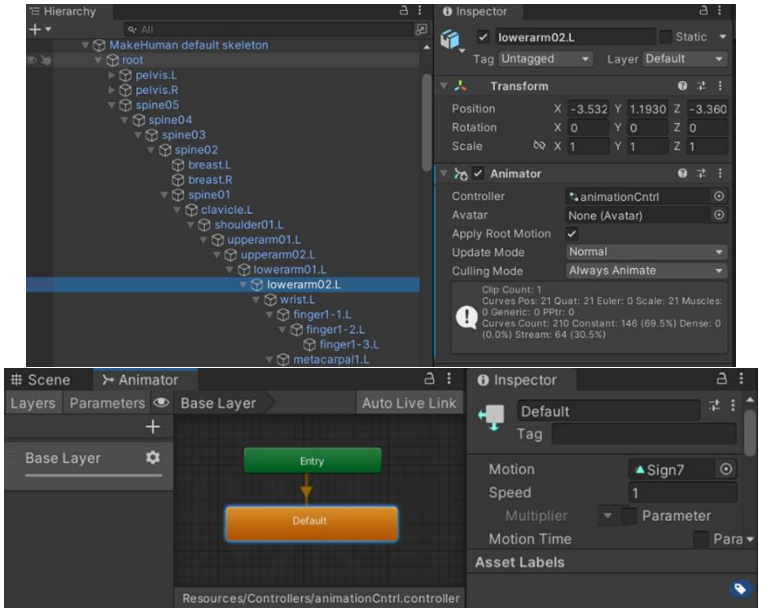


Figure 12: Manual Placement of Animator and Controller with One Animation Clip

After the Human Model Generation subsystem is implemented, freshly spawned models are being instantiated into the scene (without these components), thus the process of component placement and animation clip selection had to be scripted. But 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. 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 13: Dataset Specific Animation Controller Containing all Sped Up Gesture Clips

Once a MakeHuman model has been instantiated into the scene, the first step is to locate the lowerarm02.L bone as shown below so that an Animator component can be placed on it.

```
//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 14: Scripting Search for Lowerarm02 Bone of MakeHuman Model

Once located, it is possible to add the Animator component and enable it. 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.

```
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 15: Alphabet Dataset Example for Randomly Selecting and Playing an Animation Clip

I also implemented an optimization for the time it would take to build the entire alphabet dataset (28 gestures total, 12,000 images per gesture) 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 have been performed the maximum number of times selected, the data capturing/collection subsystem stops the running of the Unity environment.

```
//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 16: Removal of Gesture Choice from String of Gestures

4.3. 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. I then implemented the process of adding components/controllers to sequentially spawned models through script, which was tested and validated on its own using an independent “ModelAnimation” script (which was then incorporated into the SceneController script in the completed Unity environment). Below you can observe the gesture animation K being performed on 3 unique, sequentially spawned MakeHuman models who did not have Animator components placed before being instantiated into the scene.

Figure 17: Gestures Performed on Distinct Imported MakeHuman Models Automatically

I 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. I 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, I validated that the subsystem would remove Alphabet animations from being selected 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, I printed the new string of choices and saw that the choice of the particular gesture had been removed.

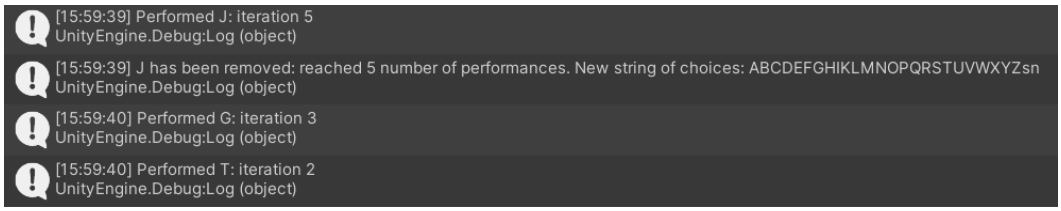


Figure 18: Log After First Letter J Reached Maximum Number of Iterations

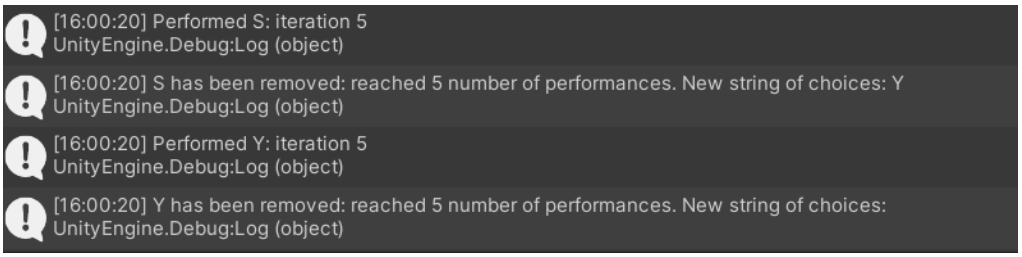


Figure 19: Log After Last Letter Y Reached Maximum Number of Iterations

4.4. Subsystem Conclusion

The model animation subsystem has been fully validated and is functioning as designed. It uses the previously recorded animation clips from the gesture data collection subsystem and the sequentially spawned MakeHuman models from the human model generation subsystem to play each gesture animation a maximum user selected number of times. This proves that the created subsystems to this point are easily integrated together to form the full synthetic gesture training set generation system.

5. Data Capturing/Collection Subsystem Report

5.1. 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 and stores the image as a .jpg in folders sorted by dataset and gesture using the scripted TakeImage method. 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 diverse data. Images are sized to be 512x512 pixels for neural network usage.

5.2. Subsystem Details

This subsystem works directly alongside the model animation subsystem, as both the PlaceCamera and TakeImage functions are called directly after a random animation is performed. In order for this subsystem to function properly, I had to create a number of virtual camera objects within the scene hierarchy that faced the 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 the figure below. This is because I was unable to record the gesture animation for G with my left hand rotated to the left using the LMC in a head mounted position. Instead, I recorded the gesture animation for G (and other letters) with my hand in the same orientation as I had with B, but altered the camera angle when looking at the played animation to make the gesture look as though it is horizontally oriented.

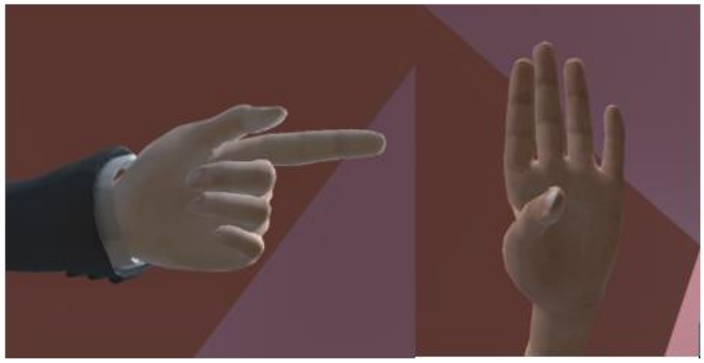


Figure 20: Camera Placement for Gesture G vs. B

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. By adding randomness in the camera position, we can ensure that the gesture will be viewed from a number of 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 21: Scripting Randomness in Camera Position

The TakeImage coroutine is responsible for recording a screenshot of the game view, naming the output file as well as the designating the output folder path depending on the chosen dataset and randomly played gesture animation. The Sign Language for Digits dataset named its files as the name of the gesture followed by the image number of that gesture, while 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 first determines how to name the image following the same conventions mentioned above and determines where to store the gesture image depending on what gesture was signed.

```
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 22: TakeImage File Naming, Output Folder Designation, and Delay

Before taking the screenshot using the enabled camera from the PlaceCamera script, the TakeImage coroutine introduces a slight delay. The reason for this delay is because all animation clips begin in the same position, but over time move differently to form the final gesture sign. If a screenshot was recorded the instant that the gesture was performed, 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, I sped up the animation clips by a factor of at 1000x and implemented a delay of .02 seconds so that I could ensure 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 amount 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.

```
//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 23: Alphabet Dataset Example of Dataset Completion/Environment Exit

5.3. Subsystem Validation

The validation of the data capturing/collection subsystem consisted of a number of tests. First, I ensured that the images were taken at the correct time after a gesture animation was performed, resulting in photos of the gestures looking as we intended (as explained in the previous section). I found that by delaying .02 units of time after the gesture was performed (while gestures were sped up 1000x), all images of each gesture were recorded after the entire animation played out and the hand was in its final position. This resulted in accurate gesture images and a decreased amount of time required to play/record all animations the maximum number of times.

Figure 24: Example Validation for Images of Completed Gestures

I also validated this subsystem by ensuring that each gesture image was stored in the correct folder according to the dataset it belonged to and named following the same convention as the real dataset used. To test each of these, I ran the system using a maximum number of 5 images per gesture for both datasets. I allowed the system to play until it completed all required gestures and stopped itself. I then referred back to the folders I had created to store the output files and checked that each labeled folder contained images of the gesture that it specified (i.e. folder labeled one stored all images of one being portrayed). In addition, I validated that each image was named correctly (i.e. images of one being portrayed were named “one…”) and included a counter indicating the image number (up to 5 in this case).

Figure 25: Example Validation of Images Properly Named and Stored

Finally, I validated that the subsystem would stop the running of the Unity scene once all gesture clips were performed by running the scene and letting all animations play out until they reach the maximum number of performances and checking that the Unity environment will exit runtime after the condition is met.

5.4. Subsystem Conclusion

The data capturing/collection subsystem has passed all validation tests and is functioning as designed. This subsystem is able to record images directly after an animation has played on a randomly spawned model and store each into dataset specific and gesture specific folders as the real datasets we replicated had. In order for this subsystem to function properly, I had to use the implementations of the last 3 subsystems, proving that each subsystem can work together to form the virtual dataset generation environment. These subsystems form the backbone of the final system, but we will explore additional scene components/scripts that we implemented to improve our dataset generation system in the next section.

6. Dataset Generation Full System Report

6.1. 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 completely virtually. 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) and a Sign Language for Numbers dataset (11 gestures) consisting of 12,000 images per gesture and we are currently working on implementing a replication of the HaGRID dataset, which includes full body images (useful continuation of our research into winter break and 404).

6.2. System Details

As described in the data capturing/collection subsystem report, we have validated that all subsystems can work together to form a consistent dataset generation environment. 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 conditional logic to quit the Unity run once the dataset is fully generated 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 the amount of images per gesture they would like to have outputted and with the click of a play button, the dataset generation process will begin.

To provide a brief overview of the dataset generation system process, 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 depending on the dataset chosen to be replicated, a random animation 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 is named and the folder path is selected, and a screenshot is captured and stored. This process repeats until all animations have been played a maximum number of times and the Unity environment stops running. At this point, the finalized dataset is created.

6.2.1. Additional System Characteristics

6.2.1.1. Randomization in Background Conditions

As instructed by our graduate student sponsor, we have additionally incorporated a background image randomization script to increase diversity in our virtual datasets. This script was written by our graduate student sponsor Pranav Dhulipala, but he gave us permission to use it within our scene. In order to incorporate this script, I created wall objects around the location where the models would be spawned. By simply placing the wall objects into the public variable Walls array

within the background randomization script, upon running the environment, each wall will appear as a random image from the Describable Textures Dataset[3] found online and linked above.

Figure 26: Background Image Randomization within Images of Gesture A

6.2.1.2. 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. We are attempting to replicate real world scenarios as best as possible, where lighting conditions are not always optimal and hands might appear darker or brighter than wanted. We plan for a gesture recognition neural network trained using our synthetic data to recognize such outliers.

Figure 27: Alterations in Lighting from Image 2 vs. Image 5 on Gesture 1

6.3. System Validation

The final system was validated first by running the Unity environment under various user-selected conditions. I tested the Unity dataset generation system using the Sign Language for Numbers dataset and selection of 5, 10, 50, 1,500, and 12,000 images per gesture. After each run, I validated that the gesture images were accurately recorded and stored with correct naming conventions. I validated that each picture clearly contained the hand gesture without too much cut out of frame as well. Next, I tested the same things using the American Sign Language dataset replication choice selected. All of these tests passed as all images were accurately taken of each

gesture and stored in their corresponding folder paths. The 12,000 image per gesture dataset is the final form of the replicated synthetic training set used for testing.

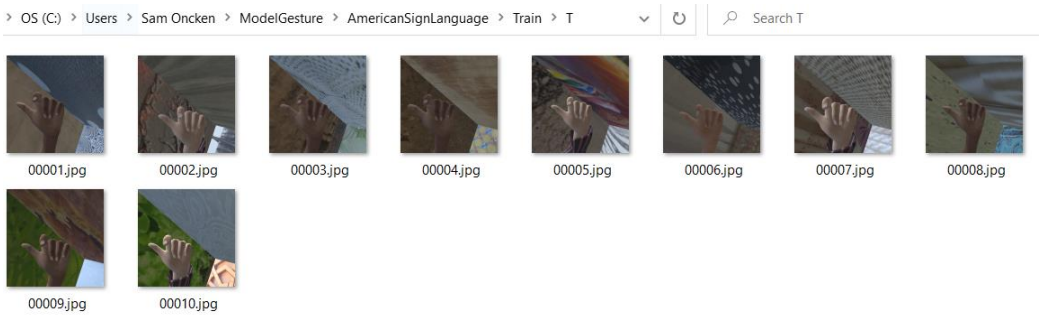


Figure 28: Validation Example - Letter T Performed and Stored 10 Times

I also validated the scene customizations by running the RandomTexture (random background image script) alone to see whether each wall changed appearance. While running both scripts at the same time, it was determined that the best speed at which we should change scene background was .12 units of time, which allowed for a new background set to appear after each new iteration of an animation playing.

Finally, I validated the light source randomization script 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 as shown below.

6.4. System Conclusion

The full system passed all validation tests, proving that all previously designed subsystems were integrated successfully. The final dataset generation system has been completed. Using the finalized system, we were able to replicate both the Sign Language for Numbers dataset as well as the American Sign Language dataset which contained 132,000 and 336,000 images respectively. These dataset outputs are stored on the lab computer in WEB 156 and are currently being pipelined into the neural networks that Steven is training and testing with, which will be described in the next section.

7. Virtual Training Set Testing Report

7.1. Training Set Testing Introduction

The training set testing involves training image classification Convolution Neural Networks (CNNs) using the benchmark datasets we found or the synthetic datasets we created and testing their hand gesture recognition accuracy. The CNN used so far is ResNet18, a well known image classification model. All training and testing is done using the Keras API in TensorFlow with Python. After a model is trained with a certain dataset, tests are run to see the model's ability to classify new images it hasn't seen before that are from the same dataset. This is done with both the real benchmark dataset and our synthetic dataset, and the accuracy metrics are compared to see if synthetic data can result in similar accuracy results when testing the model. Then, further validation for training on synthetic datasets and testing on real data is done to prove the ability to use CNNs trained on synthetic data for hand gesture recognition with real data. This opens up the possibility to use synthetic data to augment or potentially even replace real data when training CNN models.

7.2. Training Set Testing Progress/Methodology

The process starts with creating an input pipeline for the datasets using built in functions within the Keras API of TensorFlow. The input pipeline allows for pulling of images in batches from specified dataset directories instead of loading the entire dataset into an array, which prevents the system from running out of memory. Preprocessing is applied in the pipeline, where we normalized the values of the pixels to be values from zero to one, and randomly flipped the images horizontally to simulate both left and right hands since our data had only left handed images. The batches of images are also split into a 50/25/25 training, validation, and testing split. Three pipelines are created for the benchmark dataset, our synthetic dataset, and our personally created dataset. The next step involves setting up the model for training and testing. The ResNet18 model is created using an image classifiers Python package. Adjustments are made to the base model, either adding more convolution layers, and/or freezing the weights for transfer learning. All models must have a final global average pooling layer and a densely connected layer that has the same number of neurons as there are classes in the dataset for the final classification.

With the setup complete, the model is then compiled to specify the loss, accuracy, and optimizer used during training. Training is run using one of the datasets, and the metrics are recorded and graphed. A test using the testing partition of the training dataset is then run to ensure the model's ability to classify images from the training dataset. The model is then tested on one of the other datasets to validate its ability to generalize when classifying images that differ from the training dataset. The accuracy of the test is recorded, and a confusion matrix is generated to visualize the classification of the test images.

The following results are from the American Sign Language digits dataset. Due to constraints in drive space, we could not run tests on the other datasets since they were too large. However, we now have the capability to run these tests on the other datasets and plan on doing so over Winter break. The three datasets used are the synthetic dataset, the real benchmark dataset, and a real dataset we made ourselves consisting of images of my own hands.

For the synthetic training, tests were run using transfer learning and training all weights from scratch. However, when using pre-trained imagenet weights, the model showed almost no

improvements in training accuracy and virtually no improvement in validation accuracy. After adding two trainable convolution layers with 512 filters, the training accuracy improved to around 50% after 30 epochs, but validation accuracy still had no improvements at all. Even after adding two more convolution layers with double the filters, the training accuracy reached around 80% and validation accuracy still had no improvements.

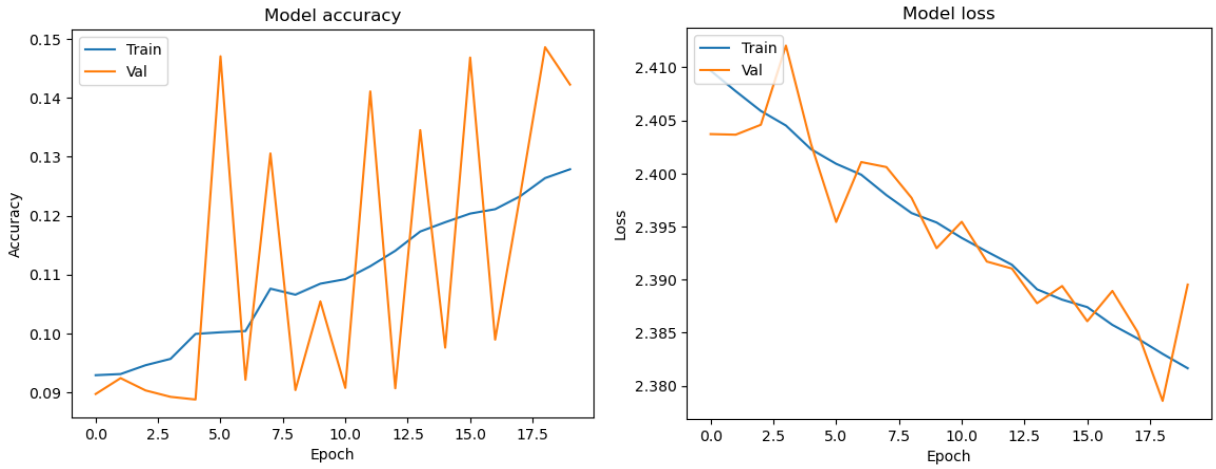
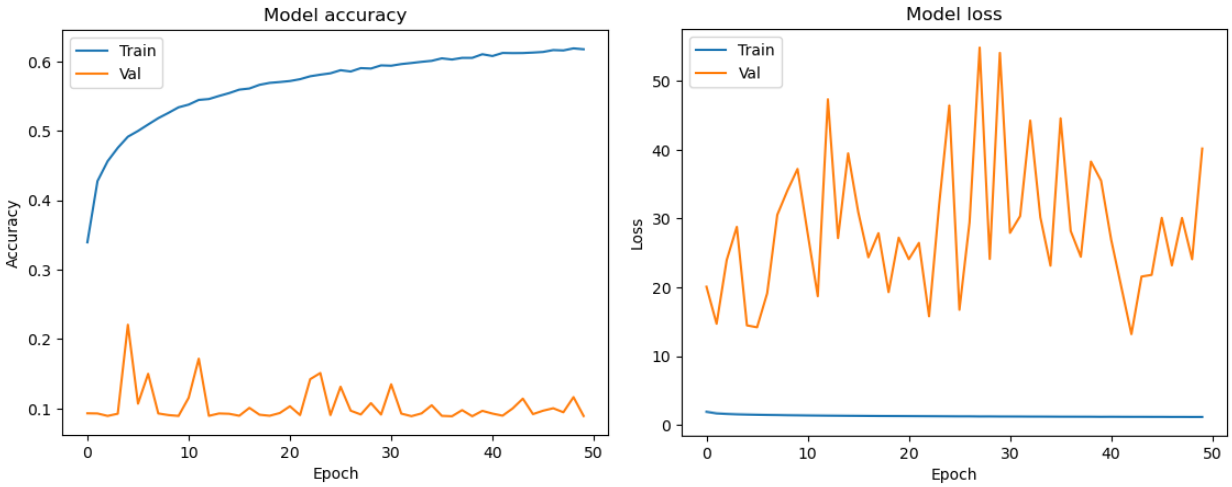


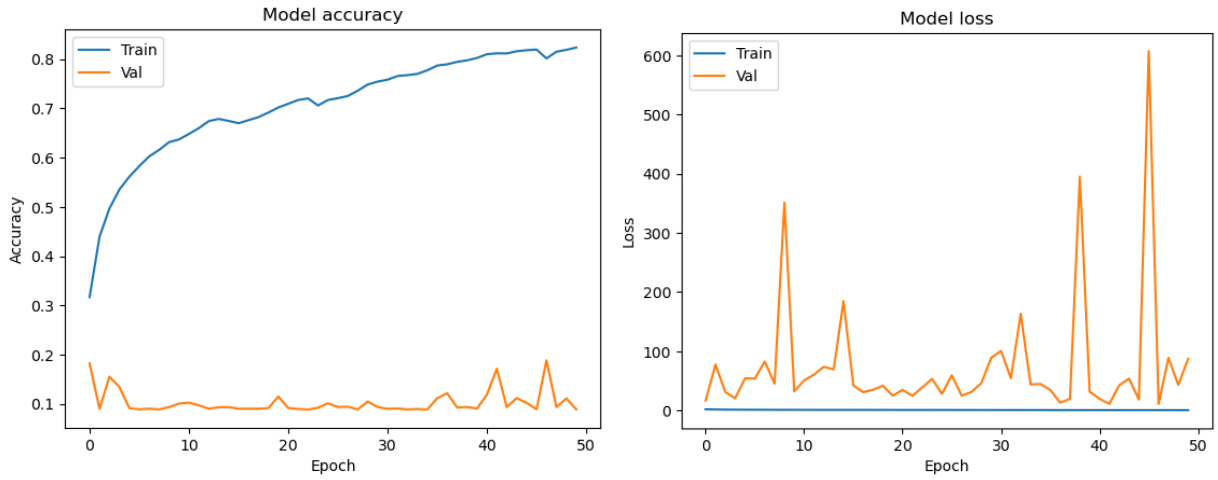
Figure 29: Synthetic Data Training and Evaluation with no Additional Layers Using Transfer Learning




```
rneval = model.evaluate(test, verbose=1)
```

```
1031/1031 [==============================] - 267s 93ms/step - loss: 40.0944 - accuracy: 0.0904
```

Figure 30: Synthetic data Training and Evaluation with 2 Additional Convolution Layers (2x512) Using Transfer Learning

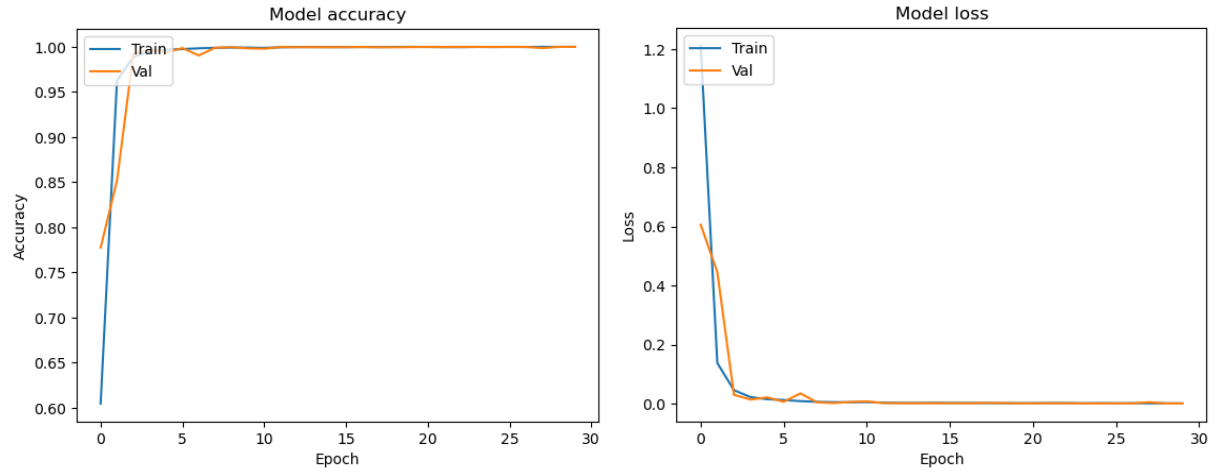



```
rneval = model.evaluate(test, verbose=1)
```

```
1031/1031 [==============================] - 257s 91ms/step - loss: 87.0687 - accuracy: 0.0921
```

Figure 31: Synthetic Data Training and Evaluation with 4 Additional Convolution Layers (2x512, 2x1024) Using Transfer Learning

With training from scratch with the synthetic data and no extra convolution layers, the training and validation accuracy reached 100% after 30 epochs of training. The testing accuracy on the test partition was 99.98%. However, when tested on the real benchmark data, the accuracy was only 7.24%, and the confusion matrix showed that the model was completely misclassifying the images. This was also seen when testing on our own created dataset, which had 19.73% accuracy testing, and was also clearly misclassifying images when viewing the confusion matrix.




```
rneval = model.evaluate(test, verbose=1)
```

```
1031/1031 [==============================] - 273s 92ms/step - loss: 5.4867e-04 - accuracy: 0.9998
```

Figure 32: Synthetic data Training and Evaluation with no Additional Layers, Training Weights from Scratch

```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 28s 81ms/step - loss: 9.6084 - accuracy: 0.0724
```

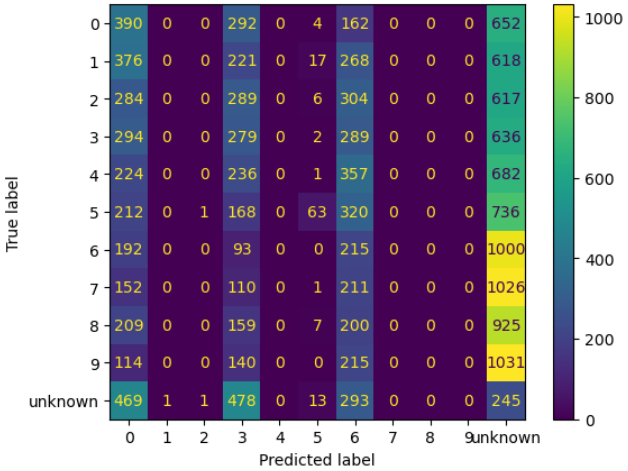


Figure 33: Evaluation and Confusion Matrix from Test Using Real Benchmark Dataset

```
rneval3 = model.evaluate(extratest, verbose=1)
```

```
83/83 [==============================] - 7s 83ms/step - loss: 7.6715 - accuracy: 0.1973
```

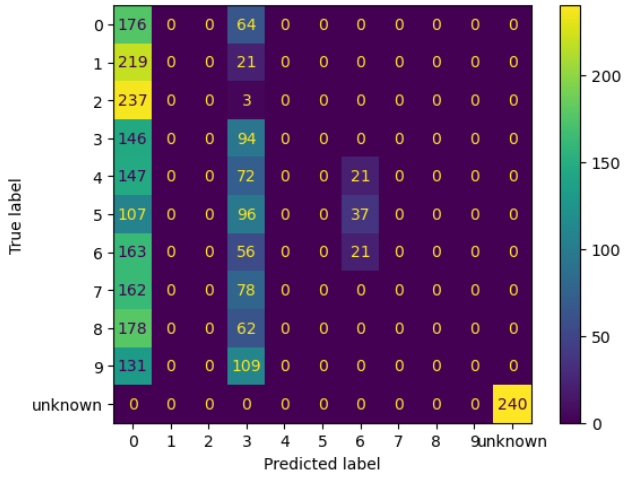
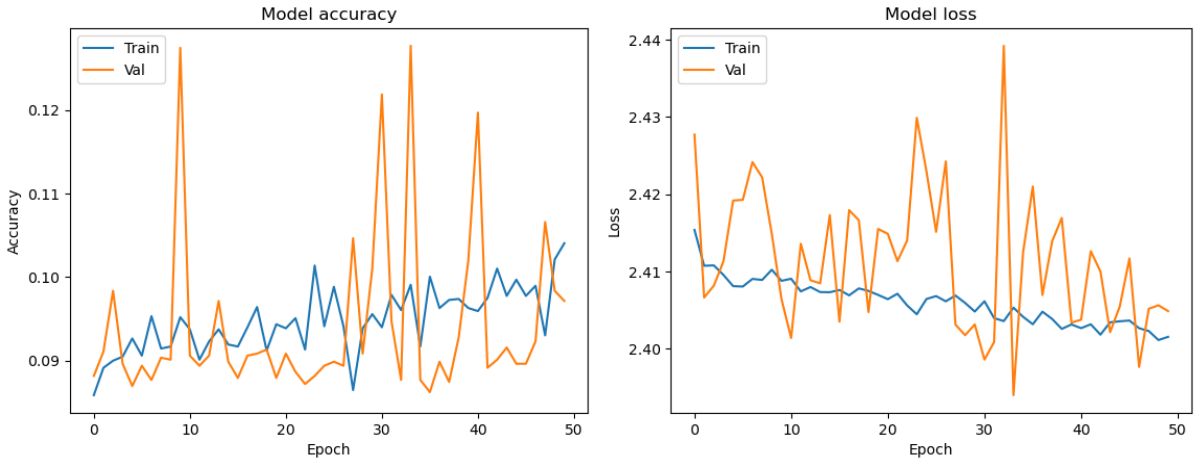


Figure 34: Evaluation and Confusion Matrix from Test Using my Hands Dataset

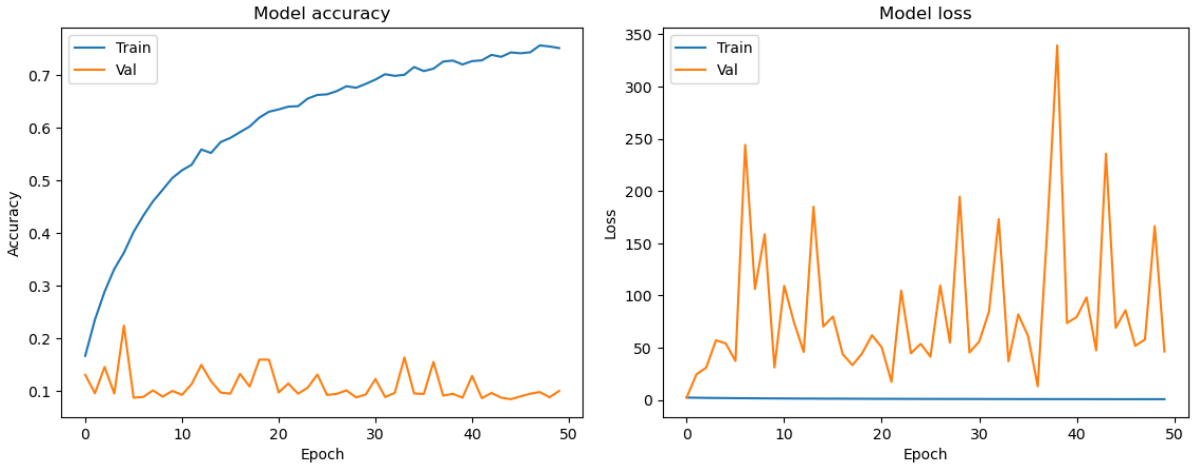
Using transfer learning training with the real benchmark data resulting in similar results as with the synthetic data. Training with no extra convolution layers resulted in almost no improvements in training and validation accuracy after 50 epochs. After adding four convolution layers (the first two with 512 filters and the last two with 1024 filters), the training accuracy improved to around 75%, while the validation accuracy showed no improvements again.




```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 35s 90ms/step - loss: 2.4065 - accuracy: 0.1028
```

Figure 35: Benchmark Dataset Training and Evaluation with no Additional Layers Transfer Learning

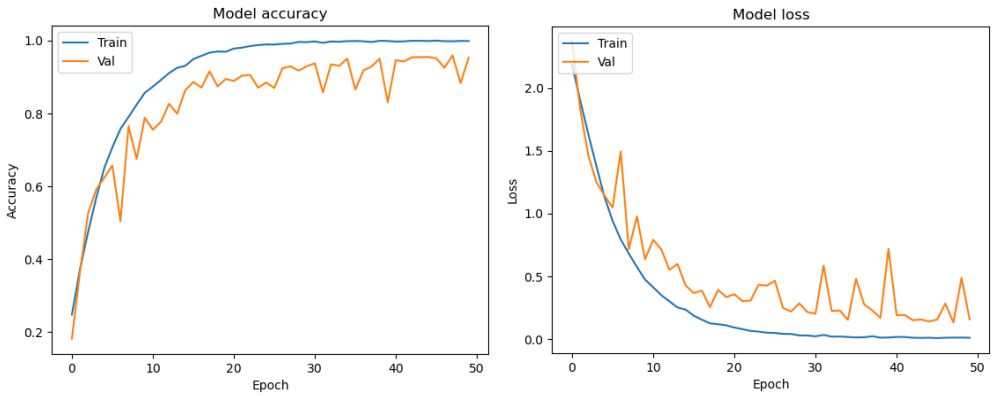



```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 25s 79ms/step - loss: 46.5351 - accuracy: 0.0984
```

Figure 36: Benchmark Dataset Training and Evaluation with Added Convolution Layers (2x512, 2x1024) Transfer Learning

When training using the real benchmark dataset from scratch with no additional convolution layers, 50 epochs achieved 99.84% and 95.3% accuracy for training and validation accuracy respectively. When tested on the synthetic data, the accuracy was only 9.31%, and on our real data it was 22.05%, with both confusion matrices showing clear misclassification of the data.




```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 25s 76ms/step - loss: 0.1566 - accuracy: 0.9529
```

Figure 37: Benchmark Dataset Training and Evaluation with no Additional Layers, Training from Scratch

```
rneval = model.evaluate(test, verbose=1)
```

```
4125/4125 [==============================] - 439s 106ms/step - loss: 9.1008 - accuracy: 0.0931
```

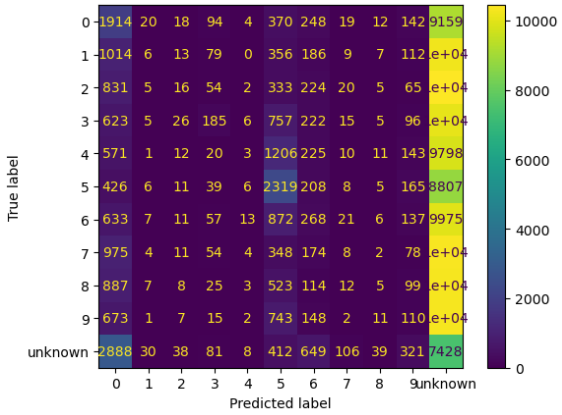


Figure 38: Evaluation and Confusion Matrix from Test using Synthetic Dataset

```
rneval3 = model.evaluate(extratest, verbose=1)
```

```
83/83 [==============================] - 7s 83ms/step - loss: 4.0121 - accuracy: 0.2205
```

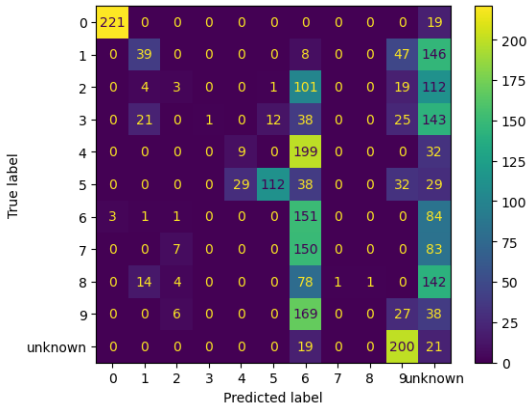


Figure 39: Evaluation and Confusion Matrix from Test Using my Hands Dataset

Below are two tables representing the data from above. For the transfer learning training, we did not run the tests (represented as 'X') because the model was already unable to validate correctly on the training data and would show no better than a guess for the image classification. For the tests from scratch, as accuracy roughly matched the training accuracies >95%, the latter tests were then run.

The following table shows data from synthetic training and testing after 50 epochs for transfer learning and after 30 epochs for learning from scratch.

Table 1: Synthetic Data Training and Testing

Additional Changes	Training Method	Final Training Accuracy (%)	Final Validation Accuracy (%)	Final Evaluation Accuracy (%)	Benchmark Data Test Accuracy (%)	My Hands Data Test Accuracy (%)
No added layers	Transfer learning	12.79	14.23	X	X	X
	Scratch	100	100	99.98	7.24	19.73
2 convolution layers (2x512)	Transfer learning	61.77	8.91	X	X	X
4 convolution layers (2x512, 2x1024)	Transfer learning	82.38	8.88	X	X	X

The following table shows data from real benchmark training and testing after 50 epochs for both transfer learning and from scratch.

Table 2: Real Benchmark Data Training and Testing

Additional Changes	Training Method	Final Training Accuracy (%)	Final Validation Accuracy (%)	Final Evaluation Accuracy (%)	Synthetic Data Test Accuracy (%)	My Hands Data Test Accuracy (%)
No added layers	Transfer learning	10.40	9.71	10.28	X	X

	Scratch	99.84	95.3	95.29	9.31	22.05
4 convolution layers (2x512, 2x1024)	Transfer learning	75.07	10.08	9.84	X	X

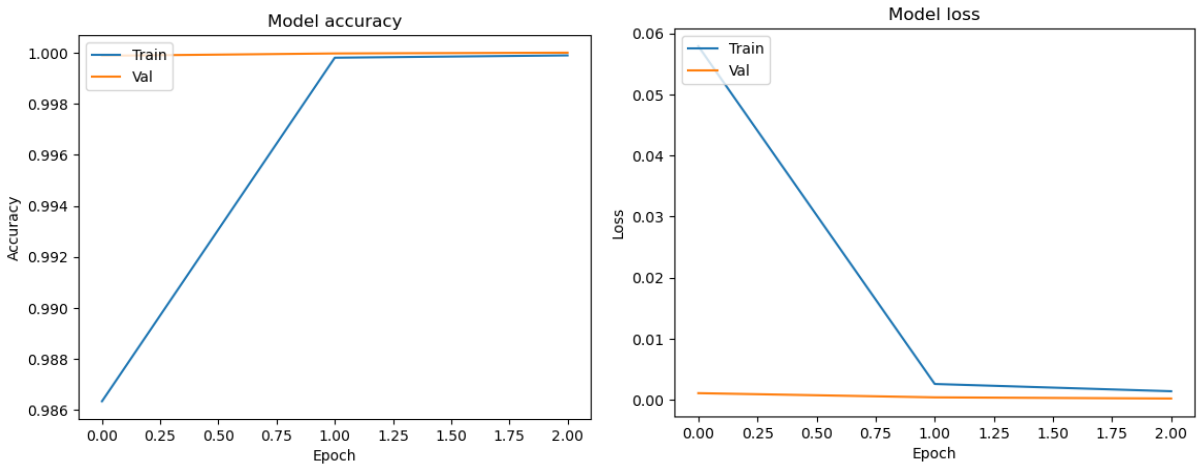
7.3. Training Set Testing Problem Diagnostic and Mitigation

When training from scratch on the real benchmark ASL digits dataset, the model's failure to classify images is likely due to the small size of the dataset. There are only 1,500 images per gesture with 11 gestures, totaling 16,500 images. For image classification, this is far too small to properly train any type of significant models for classification. The data is also black and white, which gives the models less information to work with as more data leads to a more robust and accurate model.

When training from scratch on the synthetic ASL digits dataset, the reality gap between synthetic data and real data is most likely the source of the model's failure to classify the gestures accurately. As the dataset has 12,000 images per gesture totaling 132,000 images, the size could be larger, but it is large enough to achieve good accuracy for classification. Knowing this, the model would need to be fed data that has more diversity of information, especially considering the background since our synthetic data's background is not necessarily representative of realistic backgrounds.

With transfer learning on both the synthetic and real ASL digits datasets, the model wasn't able to improve in validation accuracy in either case. There are many possibilities for why this is happening, but what we believe to be the issue is the difference between the imagenet data and our data. Imagenet generally contains objects and animals, and as we are doing hand gesture recognition, we believe the features from the convolution layers using imagenet weights are causing the model to fail. Even after adding convolution layers to the end of the ResNet18 layers that were trainable, the training accuracy slowly increased while the validation accuracy still showed no improvements.

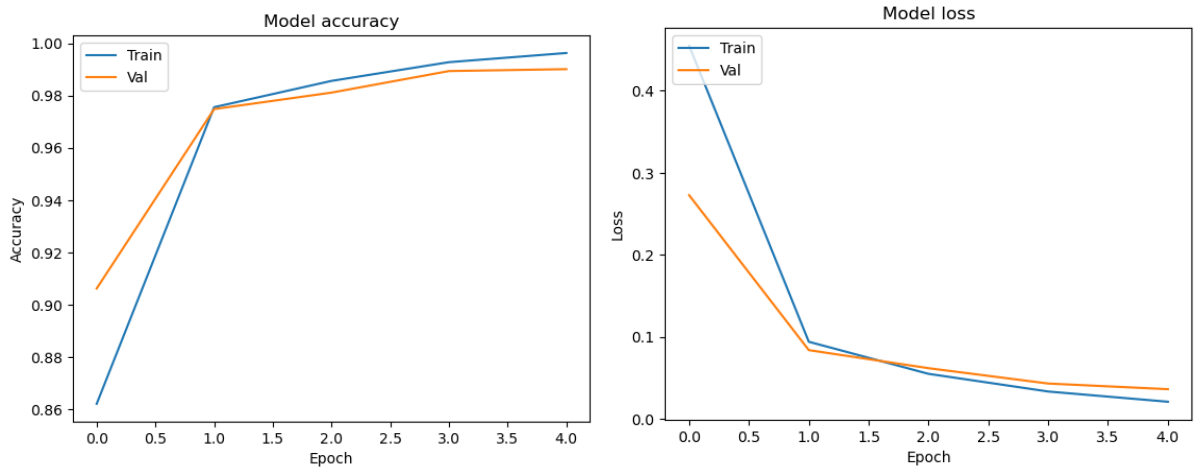
Training was also done using the imagenet weights but leaving them trainable, and then training with the synthetic data. The finished model was then immediately trained again on the real data, and tested on our real dataset. After speaking to our sponsor, they said retraining the imagenet weights is something we should not do. Though the method was slightly wrong, the results are reported below.




```
rneval = model.evaluate(test, verbose=1)
```

```
1031/1031 [==============================] - 273s 98ms/step - loss: 2.2976e-04 - accuracy: 1.0000
```

Figure 40: Synthetic Training and Evaluation Using Trainable Imagenet Weights




```
rneval2 = model.evaluate(subtest, verbose=1)
```

```
129/129 [==============================] - 25s 76ms/step - loss: 0.0333 - accuracy: 0.9891
```

Figure 41: Real Training and Evaluation using Trainable Imagenet Weights after the Synthetic Training

```
rneval3 = model.evaluate(extratest, verbose=1)
```

```
83/83 [==============================] - 7s 80ms/step - loss: 0.3403 - accuracy: 0.8723
```

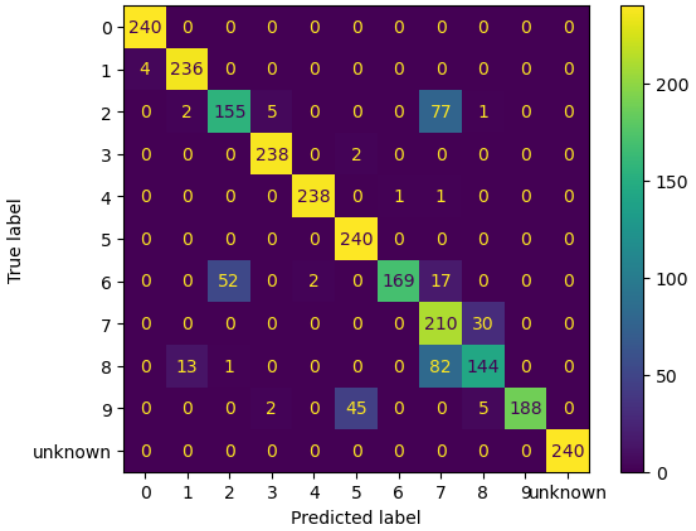


Figure 42: Test Evaluation and Confusion Matrix after Synthetic and Real Training using Trainable Imagenet Weights and My Hands Dataset

The following table shows the synthetic and real benchmark training and testing. The order in this table is the order of training, with synthetic first and real data following. The synthetic data was trained for 3 epochs and the real data was trained for 5 epochs. After all the training, the test on our dataset was run. The test with our data was run after fully training with both datasets, so the test accuracy with our data for the synthetic training was not run (represented with 'X').

Table 3: Synthetic and Real Benchmark Training and Testing

Training Data	Training Method	Final Training Accuracy (%)	Final Validation Accuracy (%)	Final Evaluation Accuracy (%)	My Hands Data Test Accuracy (%)
Synthetic	Trainable imagenet weights	99.99	100	100	X
Real benchmark	Trainable imagenet weights	99.62	99.01	98.91	87.23

As seen in the table and data above, the model is able to classify the images correctly with only some mistakes being made for certain gestures. This accuracy for testing on our dataset of 87.23% was significantly higher than the 19.73% test accuracy by training the model from scratch with synthetic data and 22.05% test accuracy by training the model from scratch with the real benchmark data. Though the method was wrong, this method did reveal the direction our training should go, which is augmenting the real data with synthetic data. This would make up for the reality gap with the synthetic data and the lack of images with the real data.

Going forward into Winter break, we will need to continue with training using both synthetic and real data using correct training methods and different model structures to optimize and improve

the test accuracy. We will also need to expand our dataset to have more images per gesture and have more variances in backgrounds and lighting to truly test the robustness of our trained models. We will continue to run these tests on the ASL digits dataset, as well as the ASL alphabet dataset.

7.4. Training Set Testing Conclusion

Though our testing of datasets was delayed and the initial testing showed accuracies well below acceptable values, we are moving in the right direction with our testing and will make progress over Winter break towards improving the test accuracy through correct model training methods. True transfer learning with ResNet18 showed no improvements in accuracy results, so it is likely that we will continue with training from scratch with no pre-trained weights. We will also continue using structures of well known image classification models, potentially expanding to models other than ResNet18. Once the adjustments are made to the training, we will be able to take large datasets and train image classification CNNs to recognize hand gestures accurately.