472 Group Project — Finger Motion

Edward Pingel, Josiah Saunders, Robinson Paz Jesus
Department of Computer Science
Brigham Young University

Abstract

In this paper, we try to address the problem of predicting a virtual character's movement based on gestures from our real-world, physical hands. To solve this problem we used OpenCV, Unity, and Tensorflow sequential ML models. We used OpenCV to interpret our hand movement, Tensorflow to predict our outputs, and Unity to interpret and display our predictions on the virtual character. Doing this, we were able to successfully predict our virtual character's movement for our predefined animations. We were not, however, able to generalize our outputs to work with any hand gestures to predict an arbitrary virtual character limb configuration.

1 Introduction

It is projected that by the end of 2020, the total value of the animation industry will reach 270 billion USD [Laura Wood, 2020]. With fast growth in fields of animation and gaming, we see the large use of Artificial Intelligence and Machine Learning to come up with faster creations and iterations of computer-created characters and their regular physical movements. When using these approaches we see a positive impact in all these industries and a push for creativity and budget efficiency.

For this reason, we decided to experiment on creating our own animated character that is controlled by our hands and moves at our will. We are using computer vision to track the movements of the joints of the hand and send that data back to a client to display our intentions and mimic our movements.

We will be using TensorFlow as our main platform for Machine learning as it provides data serialization and is a widely used Machine Learning library [Nelli, 2018].

1.1 Unity Virtual Character

Our predictions for this paper were solely focussed on creating predictions for our virtual character. We developed a virtual pair of pants and placed it into the Unity game world (see figure 1). We decided to use the local rotation of each leg bone as our labels (upper right leg rotation, lower right leg rotation, upper left leg rotation, lower left leg rotation). Our machine learning model would then predict the virtual character's leg rotations based on an interpretation of our real-world hand configuration (see 1.2 for more details, which Unity would then display on the screen.

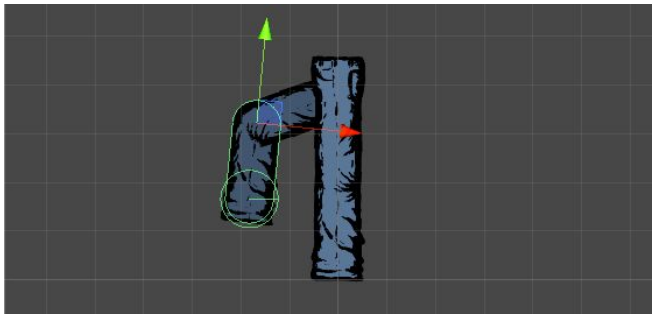

Figure 1

1.2 Hand Interpretation

One step vital to training and ultimately creating predictions of joint locations and rotations in the final model was finding a way to digitize hand movements and predictions. We considered a glove device which could measure the angles between joints, however, the complexity of building and using such a device, along with the barrier of entry this would cause for any potential users suggested that this approach would be unnecessarily prohibitive. Instead we decided to approach the problem from a computer vision perspective.

Using the python version of OpenCV we built a simple yet effective system which could visually track the

location of several markers which we affixed to a simple glove that the user would wear.

Each marker is distinguished by an unique color, so that the data can be easily constructed with a consistent structure. Each point corresponds to a joint, and the data is captured as an array of absolute positions within the camera frame. The result of this camera based finger tracking system is shown in figure 2.

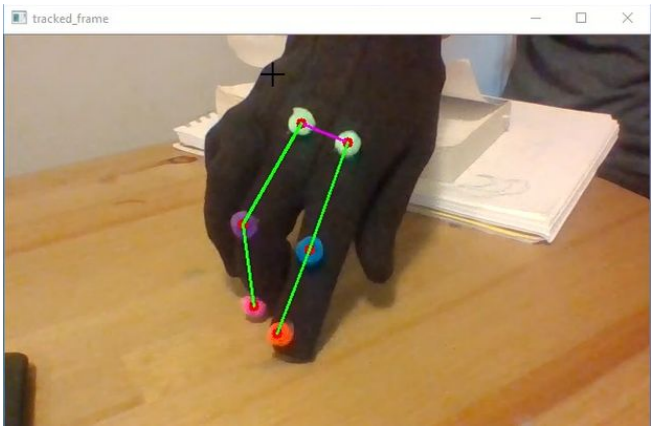

Figure 2

2 Approach

This section covers our approach to predicting hand rotations. In short, we communicated between the brain and the Unity client over UDP connections. The client would request predictions and provide target rotations while we gathered data. More details are provided in the following sections.

2.1 UDP Connections

In order to sync up and communicate between the Unity client and the brain, we used UDP. The client had two threads constantly spinning, one for receiving data from the brain over UDP and another for sending data to the brain over UDP. The brain had a similar configuration, one thread for receiving and one for sending data back to the client. These connections allowed us to send requests from our client to capture (gather) data, score a dataset, and predict our character's rotations. The original inspiration for this setup is attributed to Singh's article on Unity to Python communication [Singh, 2018].

2.2 Capturing Data

The method we devised for compiling our dataset relies on the client, the Unity virtual character, and the computer vision based server used to capture the finger locations, the Brain.

We first created animations for walking as well as crouching. These were simple looping animations in which the action we were trying to emulate were performed by the computer. The animations serve as the ground truth for our data collection represented by the angle data for each of the 4 joints in the character. This data could then be sent via UDP to the brain for processing. The user collecting the data would wear the glove, and match the motions of the animation while they were sent to the brain and concatenated together into a raw dataset containing the finger positions and deltas from the previous frame with the 4 angles as output labels.

To sync the ground truth animations with the users motions, we sent requests at a given pulling frequency from the Unity client to the brain. When the brain received one of these requests (by default every 0.1 seconds) it would pull the camera controller to get the instantaneous tracked locations of the fingers. By also keeping track of the finger locations from the previously tracked frame, we could calculate the deltas for each of the finger locations and add that information to the raw dataset allowing the model to be able to draw patterns based off of the motion of the fingers, not just the static positions.

2.3 Preparing Data

Once we captured all of our data (we'll call this raw data because the data hasn't been manipulated at all), we took several approaches toward preprocessing it, in order to boost our accuracy. First, we sent each instance in our datasets through a scikit imputer. Once the data was imputed, we then normalized all features in order to create no bias towards any particular feature.

Once we obtained our preprocessed raw dataset, we then sent the data over to what we called "data-builders." These data-builders effectively took what was provided in the processed dataset and transformed it into a new set of features (e.g. a new dataset which can be trained on) which correlated to our 4 output rotation labels. This was all in an effort to fit a more accurate model. A few of the key data-builders are described as follows.

2.3.1 Data-builder – Raw and Deltas

This data-builder took in the raw data and computed deltas between points in screen space coordinates. It did this by keeping a reference to the previous frame's points and subtracting those from the current frame's points. The data-builder then used these deltas and the current frame's points for each instance in the dataset.

2.3.2 Data-builder – Vectors and Magnitudes

In an effort to reduce the need for depending on screen space coordinates, we created a data-builder which induced normalized vectors and their magnitudes, prior to normalization, from the points on the real-world interpreted hands. For example, given the position between the blue and orange data points, we subtracted the two to get a vector.

We then pumped the resulting vectors of each instance into a dataset.

2.3.3 Data-builder – Big Hefty

Our models continued to show unpromising results. We decided to create a data-builder which would induce many possibly useful features for the model. We called this data-builder "Big Hefty" because of the very large feature set it produced. Based on the raw screen space coordinates, Big Hefty would append the following features to the dataset:

- The current and previous frames interpreted hand points in hip coordinate space (e.g. subtract the location of the hip origin from all other points in the hand interpretation).
- The angles between the key joints on the interpreted hand model (e.g. the angle formed where the green-to-blue line meets the blue-to-orange line.
- The vectors and magnitudes of each line on the interpreted hand model (described in section 2.3.2).

In total, the Big Hefty dataset had 58 features and 4 labels.

2.4 Training and Testing

Once the data was refined and processed, we moved to start training the model. For our training we split our datasets into 70% for training, 5% for validation, and 25% for testing.

Using the Keras API available in TensorFlow, we decided that a Sequential model would be the most appropriate to use (as we could stack, interchange, and test different layers). We tried training our models with different TensorFlow layers to improve accuracy, focusing on achieving a low MSE. MSEs of 20 or less were considered "good" because that meant our error in virtual character limb rotation was only around 1 degree off the target (because we are predicting for 4 output joints, so the MSE is the sum of all label MSEs).

To train our model, we ran our model for 500 epochs and saved the state of the model at every iteration (e.g. a checkpoint so that we could revert to promising models as we found them). We would then select the model who's MSE accuracy was the lowest and use it to continue on the prediction (preferring MSEs less than 50).

2.5 Predicting

Once we trained our model, we loaded the serialized model into memory and started our brain server (as discussed in section 2.1). The brain server would then process prediction requests from the client and run them through the pre-trained model's predict function.

With our prediction input features, we follow the same procedure to normalize, impute missing values, and induce features on the novel data, in the same way that the training data was created (as discussed in section 2.3). We then take our best serialized model and use the transformed data as the input to make the prediction. After we have a prediction, we send it back to the client and see the Unity client display the prediction in the form of a movement in our animated, virtual character.

3 Results

The results of our approach, at first, had no correlation that could accurately be drawn between our interpreted hand and our target rotations. As we improved our feature set, we were able to get reliable outputs which could predict our target rotations. The results, unfortunately, did not generalize well to animations not described during training. More details of our results will be discussed in the following sections. Please note the following throughout the discussion of our results:

- Our scoring function here was MSE. In the case of our project. The MSE was the mean squared error of all 4 of our output rotation labels combined. Our output rotations ranged from 90 to 270 (degrees).
- Halfway through gathering results, we decided to change our interpreted hand model to use a 6-point model instead of a 5-point model, so as to more accurately induce the orientation of the interpreted hand model based on hip rotation. Each result section is suffixed with 5-point or 6-point to indicate this.

3.1 Just Deltas (5-Point)

In the first stages of testing, we used only the deltas to make predictions. In doing so, we were able to get a decently low MSE, but we quickly realized this would not work because if we're applying these delta rotation predictions at each frame, our model would get very far off of the target desired rotations (as the delta prediction error would accumulate over time). Not only this, but due to an insufficient amount of data (only 1000 instances), our results showed that we overfit the dataset (see figure 3, not the validation and training set divergence). With these results, we then decided that it would be more efficient and reliable to predict absolute rotations on the virtual character instead of the deltas as well as to supply our models with much more data.

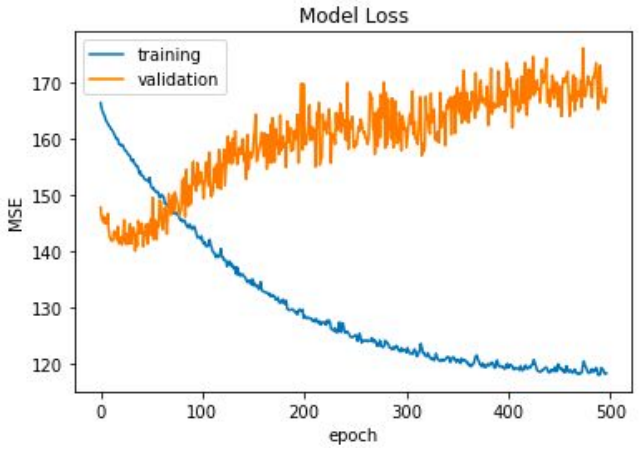

Figure 3

3.2 Unmodified Data Points (5-Point)

This dataset, the raw dataset as described in section 2.2, produced very poor results. The MSE was all over the place between different training sessions, but the average MSE we obtained for this training set was around 12,900 MSE for 2400 instances. The chart for this training run is shown in figure 4. These results suggest that the validation set was unrepresentative of the training set. We believe that this happened because the points are raw and unprocessed—the features are arbitrary points in the screen space coordinate system. We believe that this, combined with the fact that humans will likely not place their hands in the same place while they're training, cause the validation set to be unrepresentative of the training set.

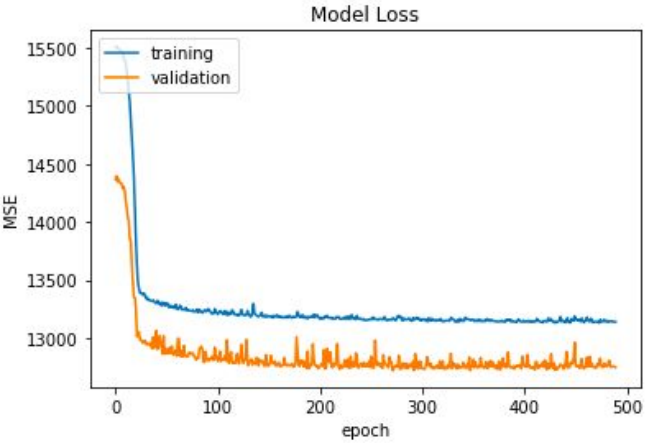

Figure 4

3.3 Vectors and Magnitudes (5-Point)

This dataset, as detailed in section 2.3.2, produced unsatisfactory results. Not only was each run of the vectors-and-magnitudes dataset producing horrible MSE, each training session produced much much different results. The average MSE we were able to achieve from this dataset, after running for 500 epochs on 5000 instances, was around 7000 MSE. A typical results graph of these runs is shown in figure 5. The most likely explanation for these results is that the model was under-fitting the dataset (because the validation set MSE was much higher than the training set MSE). We believe that just using the vectors and magnitudes, as the only features, wasn't enough information for the model to be able to create accurate predictions.

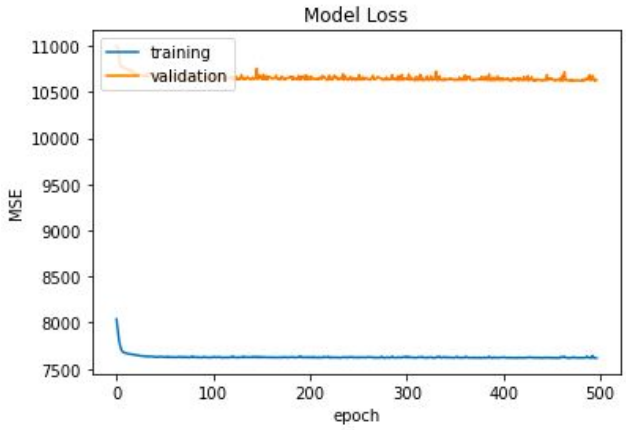

Figure 5

3.4 5-Point Vs. 6-Point Input Model

In order to combat the problems that we ran into previously with the unmodified dataset, as well as the 5-Point vectors and magnitudes set, we determined that it was likely that our model was unable to draw any conclusions because it lacked a frame of reference which meant that it was not possible for us to adequately normalize the data. This resulted in very poor prediction results especially when the captured hand positions within the frame varied even slightly from the locations on which it was trained.

Initially with the 5 point system, there was simply one point representing the hips of the character. While this was simple and initially seemed to be sufficient to be used as the origin of the tracked character representation, there is no way to form a robust and repeatable coordinate space for the tracked points' transformation using only one point on a 2d plane. To combat this, we moved to a 6 point system with 2 points representing the hips. During normal motions, these two points create a line which can be used to generate a normalized coordinate space.

The 6-point input model allowed us to build a much more consistent final dataset on which to train the network, and gave the network the ability to understand the relationship between the points and the tracked character coordinate space. This input model helped the network to generalize the patterns rather than simply memorizing the dataset.

3.5 Big Hefty (6-Point)

This dataset, as described in section 2.3.3, produced the most promising results (albeit, not perfect). We ran the Big Hefty model on a couple different datasets, walking and crouching (which are described in the following sections). After running the Big Hefty model on a small dataset, it generally produced promising MSEs of around 100-200. With these results, we decided to gather more data to run with the Big Hefty model.

3.5.1 Crouching Results

Using the described features, we created a dataset for the crouching animation which had around 22,000 instances in total. We trained on this dataset for 500 epochs multiple times. Each time we trained, similar results of less than 100 MSE were achieved.

We tried running this dataset through a sequential Tensorflow model with a few different layer configurations. The first configuration we tried was a 3-layer model with a RELU activation at each level. Each run with this model produced an MSE of around 70.

We then tried running this dataset with a sequential Tensorflow model with 10 layers, consisting of a normalization layer, a couple sigmoidal layers, an input layer, an output layer, and a few RELU layers. This model produced the most promising results for the crouching dataset with results of around 20-30 MSE. This effectively meant that there was around 1 degree of error on each label. The results for these runs are shown in figure 6. From this figure, it's clear that the model has generally learned to fit the problem.

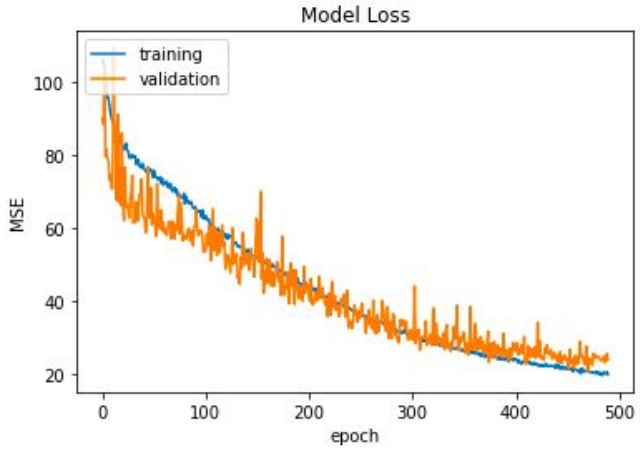

Figure 6

We can see this from our validation set MSE syncing up almost perfectly with our training set MSE.

One caveat to the crouching dataset is that the animation itself is trivial. The model can learn to rely solely on the angles between the joints. After running this model through Unity, applying live predictions to our hand gestures, the model could perfectly understand when to "crouch" and when to "stand up." It did not, however, generalize to understand any arbitrary movement. For example, if we tried to walk, the model would only predict something similar to what it saw in the crouching dataset.

3.5.2 Walking Results

While the results on the crouching dataset showed a great deal of promise, the walking dataset proved to be much more difficult and we were unable to achieve an entirely satisfactory result. Our walking dataset consisted of around 14,000 instances and was trained on 350 epochs as more than that began to overfit the training data. We began initially with the same sequential Tensorflow model as we had used in the crouching dataset, however this was unable to perform nearly as well as it had done with the crouching dataset with a consistent MSE of around 200.

In an attempt to produce more reasonable results, we adjusted the Tensorflow model to be wider instead of deeper consisting of just 4 hidden layers. The first and last hidden layers were close to the size of the input and output respectively, but the 2 middle layers were 1000 nodes wide each and had swish activation functions.

This augmented model resulted in less overfit within the dataset, and was able to bring the loss down to an MSE of 104 on the validation and test sets as shown in figure 7.

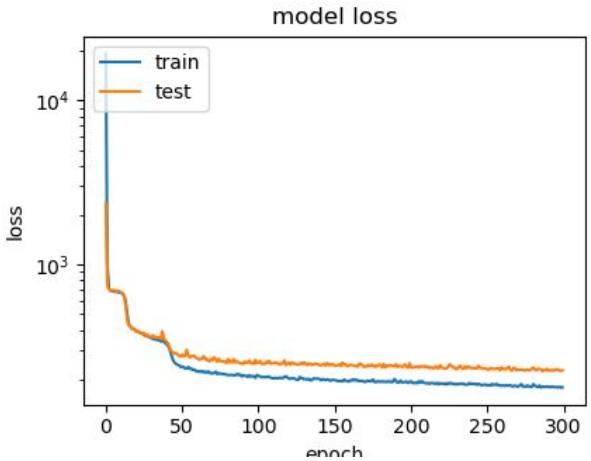

Figure 7

This new model did still cause underfitting of the dataset as can be seen by the variation between the train and test sets. Running this model through the brain to predict the angles still resulted in unfortunate results which showed very little semblance of walking.

One of the main main contributors to the poor result from this dataset could be subject to a large amount of missing data. Since the dataset was captured from a walking

finger motion, one or more data points became consistently obscured by the front finger because of this, the network had to deal with that missing point for nearly half of the walk cycle. We had provided some rudimentary logic to replace the missing point hoping that the model would learn to ignore that point when it was missing, however, it seemed to continue to cause trouble.

Although the MSE achieved in the dataset was around 104 in the end, or just around 10 degrees off from the ground truth, running the final network to predict the walking animation resulted in sporadic movements. The model had learned that somewhat random motions within a certain range would result in passable loss scores.

4 Discussion

As implied by our results section, our models were unable to generalize to any hand configuration. The model only produced results similar to output labels that it had seen before. In order to combat this issue, we believe that we would need much more data (and more animations to sync to) as well as a better mechanism for syncing our hand motion to the expected output (to gather training data). There is a lot of room for error in our data gathering approach. Humans can get distracted and miss the cues to move their hands to match the animation, the mocap system could become out of sync with the camera and cause the camera capturing to be delayed (due to slow UDP packet sending), and the camera could fail to track the right coordinates in screen space. These are all reasons for why we might be seeing the described errors in our results and why, perhaps, it's hard to generalize the model to work on any hand configuration.

5 Conclusion

Of all the improvements we tested, cleaning the features to be easier to learn was the higher indicator of accuracy. We noticed also that by having more data to train with, we were able to achieve much more accurate models and predictions. Other factors that would also add to the improvement would be a more quality camera that captures the joints with more precision. Having a more steady position of the hand when performing movements would help as well.

As mentioned before, using machine learning as a solution for improvement on speed of creations and iterations of computer-created characters and their regular physical movements is a win for any industry. If we can continue to improve the accuracy in our models, we could easily integrate the algorithm with a game or an animated movie and use significantly less computation and rendering power.

6 Future Work

There are several things that we could do going forward which would improve the accuracy of our neural network and allow us to produce better real results in the game engine.

First, gathering more generalized data would most likely make the largest difference. This would mean gathering data in a variety of different orientations, and with varied animations. This would be much easier if we were able to streamline our process and recruit more people to help with training. As it is, we were very limited to only a few animations, and essentially one orientation of the hand.

As a part of this more generalized data, adding several additional static points to the hand including possibly a ring around the wrist could allow us to estimate point locations in 3 dimensions instead of only 2 using homographies as a preprocessing step. That way, the hand's pose would not be tied to the hand's orientation on the screen. Additionally, using another open source pose estimator software to pull more reliable hand data would improve our data collection process.

Additionally, Missing points was always a problem. One method we could use is to predict the point's position based on a gaussian or parabolic trajectory from the position where the point was last seen in conjunction with it's momentum to the point where it reappears.

Alternatively in conjunction with a more generalized dataset, and improved tracking, we could address the issue of missing/hidden data points by intentionally randomly removing data and forcing our network to estimate the location of these points on it's own.

The scope of this project only allowed us to explore one method for gathering and training data because of the time consuming process to set up our data pipeline, future work and refinement will likely result in a model which can accurately and seamlessly transition between animations and predict a variety of motions.

References

[Singh, 2018]. "Introduction to Using OpenCV With Unity." Ray Wencerlich, 2018.

[Laura Wood, 2018]. "Markets, Research and Global Animation, VFX & Games Markets, 2018-2020: Total Value of Global Animation Industry Is Projected to Reach US$ 270 Billion." PR Newswire: news distribution, targeting and monitoring, January 7, 2019.

[Nelli, 2018]. "Deep Learning with TensorFlow." Python Data Analytics.