Human Controller

Yoav Orenbach
208847749
yoav.orenbach@mail.huji.ac.il

Almog Cohen
207005950
almog.cohen@mail.huji.ac.il

Group 307
Mentor: Omri Avrahami

Abstract

In the past years, video games have become a more engaging and immersive experience by playing with body movements and gestures, however, existing solutions are often too costly, require extra hardware, and fit only specific games and sets of movements. In our project, we created a virtual game controller for playing games with body poses. Users can map poses to keyboard or mouse input and play any game using the poses of their choice at no extra cost. Our approach involves three steps. First, we collect pose data from users, then we train a neural network model to classify the different poses, and finally, we predict the poses and simulate the wanted keypresses when playing a game. Due to the nature of our problem, we created a model that is robust to multiple environmental scenarios and capable of running in real time. The code is available here.

1 Introduction

The gaming industry is now one of the biggest entertainment industries in the world, surpassing traditional entertainment venues like movies and music. Nowadays, there is a shift in the gaming industry to turn games into a realistic experience and play by doing physical activity in an environment instead of a keyboard, mouse, or game controller. However, this experience is not accessible to everyone, since current solutions have fixed sets of movements that not all users can perform. In addition, these movements are made specifically for a limited number of games while also requiring additional hardware to have the system work. Therefore, the need to make this experience accessible to everyone arises, not just for specific games, but rather for all games old and new.

Different games play differently and different people play them in different ways, yet the one thing they have in common is that they are played by using some kind of a controller. Thus, to create a unified solution for all games we wanted to make the person playing into the game controller itself, playing with body poses as suggested in [1] and [2]. This way every game could potentially be played while not limiting users to predefined movements, and the only requirement is a working web camera for capturing the poses.

The first hurdle in this approach is learning what poses users can and want to use, hence the first step is to collect pose data from users for each key. The second challenge is to then classify the poses to the corresponding key, so we set out to find the best classifier model for this pose classification task. Finally, once the classifier is ready, in the third and final step, our system simulates the wanted keypress when users perform poses in front of their camera.

The main goal is to make our model robust so users will only need to collect pose data and train it once with no need to repeat the first and second steps. Afterward, they can play games with their desired poses under different scenarios, such as wearing different clothes, standing closer or further away from the camera, switching backgrounds, and room lighting. Additionally, since the goal is to allow users to play any video game, where many of which require fast and precise button presses, we must have our model run in real-time.

One challenge we faced was testing a human controller system as no public data exist for it. Since each user can choose his own poses, we start by assuming a single-person usage.

Therefore, the training and testing datasets consist of images of a single person that we generate using commodity webcams with 25 poses to simulate 25 different controller keys. The training dataset consists of 300 images per pose taken in optimal lighting. The testing dataset contains images of different backgrounds, different camera perspectives, and poor lighting. Eleven of the tests have 2500 images to evaluate the accuracy of classifying poses correctly. Furthermore, 2 additional tests with 400 and 5000 images are used to evaluate cases where a user switches between poses.

We started by evaluating an image classification model as an end-to-end solution. Unfortunately, it exhibited poor results (Section 4.4), which made us consider a more complex framework with a real-time pose estimation model to extract keypoints on the human body, feature engineering processing these points, and passing them to a neural network for the final classification (elaborated in Section 3 Methods & Materials).

We choose the best component for each stage empirically. First, for the pose estimation component, we tested three well-known real-time pose estimation models (Section 4.4) exhibiting 85%-95% accuracy and chose MoveNet [3] as it had the best overall accuracy. Second, For the feature engineering component we tested 5 methods (Section 4.5), and normalizing keypoints followed by pairwise distances between them had the best overall results of about 95% accuracy. Third, for the neural network (NN) we tested 7 different NN architectures and 5 machine learning (ML) classifiers, and the most suitable model for our system is a stacked multilayer perceptron ensemble that showed about 90% accuracy (Section 4.6). When testing with very bad lighting we observe only a 1%-10% drop in accuracy.

In addition, to understand the robustness of our approach to new poses and users we used a public dataset [4] of 10 different users performing different poses as a test dataset. We train our model on other users than the dataset, but performing the same new poses and results are encouraging. We find that training on multiple users improves the performance over training on a single user and our model reaches an accuracy of 77%-93%.

Finally, our final model connects to modern high-frame per second (FPS) games, unlike prior work that was used for simple games such as Pacman or browser games [2, 5, 6, 7], and allows users to play them with a traditional immersive experience.

2 Related Work

Multiple products in the market are doing quite well, along with several attempts similar to ours, however, they do not answer the need at hand as we explain below.

1. Kinect/Wii/Playstation Move/VR [8]:
   These are some of the main products produced in order to make gaming a more interactive experience. In fact, Kinect is the cornerstone that integrated pose estimation into gaming, and yet none of them allow the user to play every game and each one is very costly. The difference between them and our solution is that they require buying extra hardware and they are compatible with only specific games tailored for them.
2. Transfer learning to play Pacman [5]:
   This demo uses transfer learning with the MobileNet model to classify user images to move up, down, left, and right in Pacman on the browser. The difference to our project

is that this solution is limited to one game - Pacman and it uses an image classification model, whereas we use a pose estimation model to extract keypoints and feed them to a classifier (while also showing results for a baseline image classification model).

3. Browser games with pose estimation [2, 6, 7]:
These are browser games played with poses using the PoseNet pose estimation model [9]. The difference in our project lies in the fact that these are specific games with predefined poses (in the case of these games the poses are quite straightforward), so we attempt to extend it to all games and let users choose their own poses. Moreover, we use newer pose estimation models that achieve higher accuracy on public datasets.
4. Action replication in GTA5 [10]:
This paper introduces a method to control games in real-time by doing physical activity. In particular, the authors collected data of standing, walking, and running, used the PoseNet pose estimation model to extract body keypoints and fed them to LSTM cells followed by a fully connected layer, and finally showed the results in the GTA5 game. So, once again, although it is not a browser game and it can be extended, this solution targets one specific game with predefined poses, its network architecture is different from ours and there are no reports about the actual speed of their model.
5. Pose2Play [11]:
Pose2Play is probably the closest existing solution to our own, allowing users to play with body poses for free. However, the biggest difference in our project lies in the implementation. While Pose2Play claims it can be applicable to almost any game, in reality, it uses one game without proper mapping between poses and keyboard input, so it has predefined poses (their algorithm might be able to support their claim but their current implementation doesn't). Pose2Play uses the BlazePose pose estimation model [12], whereas we test different pose estimation models including BlazePose. In addition, its implementation is quite slow and runs at a lower FPS than our solution.
6. Yoga/fitness pose classification [13, 14, 15]:
While not video game related, classifying poses in various fitness applications closely relates to our project, and is the main work we built upon when creating our project. [13, 14, 15] all use different pose estimation models and classifier types, with [13, 14] also using feature engineering. We used the knowledge shared in these references so we could find the best methods for a video game pose classification application like ours.

3 Methods & Materials

The full solution pipeline from the user performing a pose when a game is played to simulating its corresponding keypress is as follows: Given an image of the user performing a pose that matches some keyboard key or mouse click, we pass the image to a pose estimation model which in turn outputs a set of points on the human body. We apply feature engineering on these points to improve performance and pass this processed input to a neural network. The network predicts the key matching the pose and this prediction enters a prediction queue. Finally, when the queue is filled with the same pose prediction we simulate the corresponding keypress. The full system architecture is seen below in Figure 1.

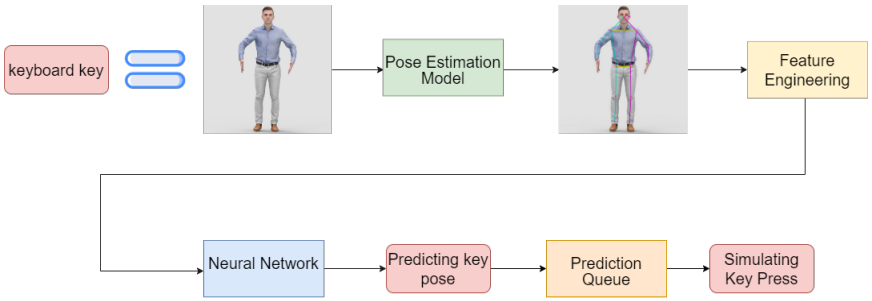


Figure 1: Full system architecture

We will now present a detailed description of every algorithm used in our pipeline, where each one was tested and compared to other algorithms and achieved the best results (elaborated extensively in Section 4 Evaluation & Verification).

3.1 Pose Estimation Model

Pose estimation is a computer vision task that estimates the pose of a person from an image or a video by estimating the spatial locations of key body joints (keypoints). In our case, where a user performs poses in front of a camera, every frame of the video is passed to a pose estimation model so we could get an estimation of the pose performed. The pose estimation model we use is the MoveNet model by TensorFlow [3], which outputs 17 keypoints on the human body:

[nose, left eye, right eye, left ear, right ear, left shoulder, right shoulder, left elbow, right elbow, left wrist, right wrist, left hip, right hip, left knee, right knee, left ankle, right ankle]

There are two variations of MoveNet called Thunder and Lightning, where Thunder is more accurate yet slower than Lightning, but both can run in real-time. It is possible and recommended to train models with Thunder and make predictions with either Thunder or Lightning. Therefore we train the model with Thunder and offer users the choice between them, without having to train again, should users need higher FPS when playing a game.

3.2 Feature Engineering Method

Once we have the keypoints from the pose estimation model, we process them so we can improve the input to our network and achieve better performance. This process is known as feature engineering which is by definition using domain knowledge to select and transform the most relevant variables from raw data when creating a predictive model.

We know keypoints change between frames, for example when a user is closer or further away from the camera, or when the camera is set at a different angle. Therefore, the feature engineering method we use is normalizing and centering the keypoints produced from the pose estimation model. Centering by subtracting all points from the hips' middle points and normalizing by dividing the points by the size of the pose. This helps our model to stay robust in these situations and is a common procedure when classifying poses [13, 14].

Once the keypoints are normalized we further compute pairwise distances between every pair of normalized points. The idea behind pairwise distances is that when two or more landmark points are close to each other, they can be used to recognize gestures as in [14]. In particular, for 17 keypoints, we choose 13 points (all points but the eyes and ears, since the distance between them and the rest of the points is proportional to the distance between the nose to the rest of the points), and compute the distance for every pair of points. Namely for $(x_i, y_i), (x_j, y_j)$ we use $(x_i - x_j, y_i - y_j)$ for all pairs with no repetition. So, for 13 pairs, we get $\binom{13}{2} = 78$ possible combinations, and the input to our neural network is a $(78, 2)$ vector.

3.3 Classifier

The classifier we then use is a neural network that given the processed $(78, 2)$ vector, outputs a softmax vector with the probabilities of every pose, and we take the argument with the highest probability to predict which pose was performed. To achieve the best predictions we tested many classifiers (Section 4.6) and found that the best classifier for the task is a stacking ensemble of multilayer perceptron networks (basic fully connected networks we call MLP for brevity) simmilar to [16]. This way we can create a robust network that generalizes better and reduces the high variance of a single network. Specifically, we use 3 MLPs with two hidden layers, and to avoid overfitting to the given poses, we use a dropout regularization of 0.5 after each hidden layer [17], and we also keep a small percentage of the data as validation data to use for early stopping as another means of regularization [18]. We then concatenate the outputs of the 3 networks and pass them to a "meta learner" which we also train for a small number of epochs (however when the meta learner trains, the previous networks do not train as well so that their weights won't change). The full network architecture is seen in Figure 2.

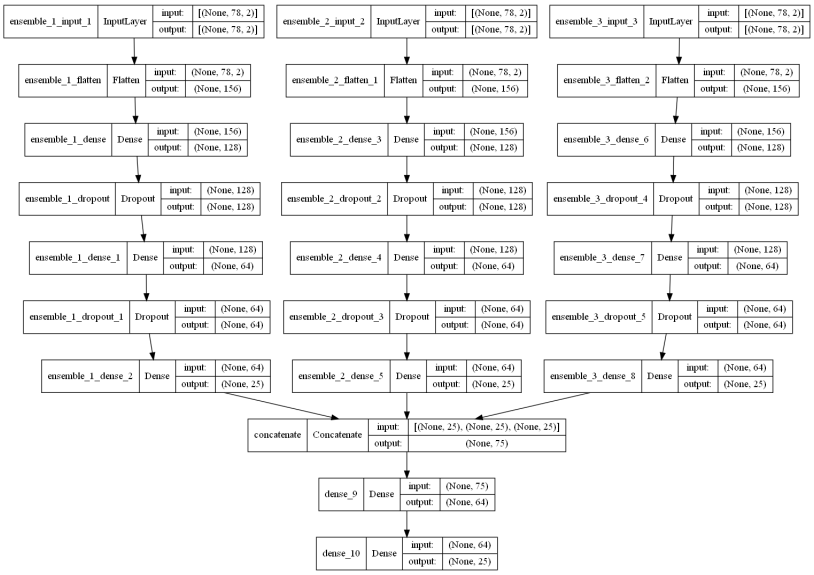


Figure 2: Full stacking ensemble network architecture

3.4 Prediction Queue

Finally, once we have the predicted pose, we insert the prediction into a prediction queue of variable length and simulate a keypress only when the queue is filled with the same prediction and this prediction probability passes a 0.9 threshold (which was found to work best as described in Section A.1). The goal of the queue is to avoid simulating unwanted keypresses, for instance when a user switches from one pose to another or when a user accidentally changes to a pose he did not mean to, there is no need to simulate a keypress as it was not intended. However, since we do not know how fast a user is when he is switching poses, we cannot know the size of the queue in advance so we let the user choose the best size for him as further explained in Section 4.7.

3.5 Technical Details

- We created an app to allow users to use our system, from collecting pose data and mapping each pose to a keyboard or mouse input, to training the model, and finally to playing using the poses that were collected.
- In our implementation, we ask the user for two required poses - a "Normal" pose and a "Stop" pose. The former is meant to be a pose where no keys are being pressed to allow the user to stop simulating keypresses while the program is still running (similar to how one would not always press keys on a game controller). The latter is meant to be a pose to end the program run as it could potentially run for an indefinite amount of time depending on the user's playtime.
- Many games have the option of pressing multiple keys for actions that are not possible with only one keypress, as well as long key presses. In order to allow for multiple key presses, we allow each pose to map to a variable number of keys, and in order to allow for long button presses, we simulate a keypress for as long as the user is performing a pose that is akin to pressing a button on a controller.
- In the first step, collecting pose data takes about 10 seconds per pose and we collect 300 images per pose as it gave us the best results, and collecting more data results in diminishing returns (Section 4.6).
- In the second step, training the full model takes about $\frac{1}{3}x + 1$ minutes where x is the number of poses the user collected data for. This is the time taken on a standard CPU with no other running tasks, and this time can decrease with the use of a GPU instead.
- In the third step, where the user is playing games with poses and our system creates keystrokes in the background, our program runs at 10 FPS with the default model using Movenet Thunder. This processing time is taken on a standard CPU with a basic commodity laptop camera, while a faster processing camera can reach 15 FPS. We find this inference time to be sufficient for playing the games we've tested with poses (we tested the newest high FPS modern games) however if the inference time is not sufficient for some games, our faster model using Movenet Lightning reaches 25 FPS on a standard CPU. Similar to training, if a GPU is being used, our default model with Movenet Thunder can also reach 25 FPS with a basic web camera.

3.6 Hardware Requirements

As previously stated we wanted to make the experience of playing games with the body accessible to everyone and for our project to work we must ask the user for a usable camera to capture poses. Nowadays, many laptops come packaged with a web camera and even the phone can be used as one with many cheap options available on the market apart from that.

Our process runs in the background while a game is played, however, most games running as the main processes do not register virtual key presses and some are expecting DirectInput/XInput keypresses [19] under the windows operating system. Therefore, to accommodate all games our project is compatible with the windows OS.

3.7 Implementation Details

We use the Python programming language and the code is publicly available (here). The code was structured with object-oriented programming principles in mind. Specifically, we created abstract classes for each of the three major components of our pipeline - pose estimation model, feature engineering, and classifier, and implemented classes that extended them. So, while our default models and methods for each of the three are the ones that achieved the best results, it is possible to run the program with any other model or method that we tested by running our code with different arguments. This way, if a new state-of-the-art pose estimation model is published, one could extend the abstract class with it, and use our system with it to achieve even better performance. Similarly, if a new feature engineering method or a better classifier is found, our code can be extended by implementing the corresponding abstract class.

4 Evaluation & Verification

A large portion of our project was dedicated to evaluating the different components of our pipeline and verifying our system works with different poses in modern games. A major drawback to asking users to give us pose data to train on is that we cannot know the poses in advance and verify that our system works with every pose the users would choose, however, there are a few assumptions we can make. For instance, we can assume the user will face the screen since they would need to look at the game they are playing. On top of that, users will stand at a distance that allows their bodies to fit in the camera frame (they could use part of their body, but our solution consists of body poses and we use keypoints on the entire body), so we can assume they would stand about 3-6 feet away from the camera. Finally, in order to make fast actions in the game, users will most likely choose poses that are close to each other as that would mean faster key presses.

4.1 Verification Metrics

With that in mind, to verify our project is successful in simulating keypresses for matching poses while games are running and doing so in real-time, we simulated real use cases of our system. We selected numerous modern action games, as they require quick and responsive

button presses, and for each game, we chose different poses to match the actions that take place in the game. Then we trained our network on each game's poses and played every game with its chosen poses. We created 2 demo videos to show this, where one is a short video demonstrating our app and a few selected scenes of us playing the selected games with our poses (found here). The second is a longer video demonstrating more of the various games being played by using our poses (found here). In the videos, we can see different poses create different actions in every game while running in real-time (the recording software caused a decreased in FPS, and without it, our system works even faster), so we can verify our project in fact classifies the poses correctly, simulates the matching keypresses and runs in real-time.

4.2 Datasets

In order to evaluate our project for single-person usage we created training and testing data with 25 poses since a standard controller has 23 buttons and we added 2 more for the 'Normal' and 'Stop' poses. Following our assumptions, we face the camera, our entire body fits the frame, and the poses we chose are relatively close while still being challenging for a pose estimation model (e.g. body occlusions where some keypoints can't be seen). Specifically, for the training data, as stated in Section 3.5 we collect 300 images per pose in a well-lit room.

First, to evaluate our model's robustness to different scenarios than the training data as described in Section 1, we created 11 test sets to see how good our model is with classifying poses without classifying pose switches. Namely to only measure the accuracy of our classifier when training on a user and testing on the same user. The tests consists of:
(1) different clothes, (2) different backgrounds with similar lighting, (3) different background with different lighting, (4) an outdoor test, (5) very bad lighting (the user is barely seen), (6) different camera height and angle, (7) standing at a closer distance in a different background, (8) standing at a far distance in a different background, (9) standing very far from the camera in a different background, (10) standing at a far distance in a different background with a different camera perspective, and (11) combinations of the other tests. Each test consisted of 2500 images with a 100 images per pose.

Second, to evaluate our model when not only poses are performed, but also movements, pose switches, or poses that were not meant to occur, we created 2 more test sets where a user performs pose switches as he would in a game with the same 25 poses. The goal was to see that our model not only predicts poses correctly but also dosn't invoke keypresses when he should not, e.g., when there's a pose switch. The tests are: (12) recreation of the eleventh' test set with pose switches (400 images), and (13) a very long video of a user constantly changing poses in a standard scenario (5000 images). Because of the arbitrary nature of these tests, we had to manually label each image.

Third, to evaluate our model with multiple users and with different poses we did not think of, we used the Gaming 3D dataset [4, 20, 21], which is a dataset of 10 subjects performing 20 gaming actions such as walking, jumping, punching, kicking, driving, waving, swinging, and so on. Every action that the subjects performed can be seen as a pose, and we find that these poses fit perfectly as a test set for our project since each pose is akin to an action in a game scenario where subjects are also performing pose switches. This way we can train our model on different users than the 10 subjects of the G3D dataset and measure how our

model fares when training on one person and testing on others (even though our goal focuses on a single user, we wanted to verify it is possible and evaluate to which extent).

4.3 Evaluation Metrics

To evaluate our system we use common machine learning metrics, and to better explain the different metrics let's first denote the case of simulating a keypress as positive (P), and the case of not simulating a keypress as negative (N). We do not simulate a keypress when the highest probability of the network doesn't pass the 0.9 threshold or when the prediction queue is not filled with the same prediction.

- A true positive (TP) occurs when our system simulates a keypress and it should have, e.g. when a user performs a pose and wants actions to take place in the game.
- A True negative (TN) occurs when our system does not simulate a keypress and it shouldn't have, e.g. when a user switches between poses and doesn't want actions to take place in the game.
- A false positive (FP) occurs when our system simulates a keypress but it shouldn't have, e.g. when a user performs a pose and the wrong key is being pressed, or the user switches between poses yet a keypress is still being simulated - this is probably the worst we could do, as it means the user is no longer in control of the system.
- A false negative (FN) occurs when our system does not simulate a keypress but it should have, e.g. when a user performs a pose for an action to take place, but no keypress is being simulated.

Now, the evaluation metrics we use are as follows:

1. $Accuracy = \frac{TP+TN}{P+N}$: Accuracy measures our system's ability to simulate correct keypresses and also not simulate keypresses when there is no need to simulate anything. In the first 11 test sets, there are only positive examples, so we look at $\frac{TP}{P}$.
2. $Precision = \frac{TP}{TP+FP}$: Precision measures our system's ability to simulate correct keypresses and only when needed to, since having high precision implies having a low rate of false positives.
3. $Recall = \frac{TP}{TP+FN}$: Recall measures our system's ability to simulate something when a user performs a pose since having high recall implies having a low rate of false negatives.
4. $F1 - score = 2\frac{Precision \cdot Recall}{Precision+Recall}$: The F1-score is the harmonic mean between Precision and Recall, and it allows us to find the sweet spot between them so we can have the best performing model.
5. $FPS = \frac{\#Frames}{seconds}$: Since inference time is of great importance to our project we also evaluate our Frames Per Second (FPS) to make sure our program is capable of running in real-time.

4.4 Pose Estimation Models Comparison

The first component in our pipeline we evaluated was the best pose estimation model for our task. For this reason, we used state-of-the-art pose estimation models capable of running

in real-time: MoveNet [3] (specifically MoveNet Thunder), BlazePose [12], and EfficientPose [22]. We decided to not use the popular pose estimation model OpenPose [23], since EfficientPose has proven to achieve better results on a public single person dataset as described in [22]. In addition, instead of using pose estimation, one might wonder if it's possible to classify the image directly and simulate the corresponding key from the pose appearing in the image (thus not needing to use feature engineering or another classifier). For this reason, we used transfer learning with the MobileNet image classification model [24] capable of running in real-time to act as a baseline for the pose estimation models. To test the pose estimation models we followed each model's API to extract keypoints, and to keep the playing field equal, we then applied the normalization feature engineering method on these points as described in Section 3.2 and passed them to a basic MLP network for classification. Testing the image classification model was straightforward as it directly predicts the pose. Finally, the accuracy received by using the 4 models on our first 6 test sets can be seen in Figure 3.

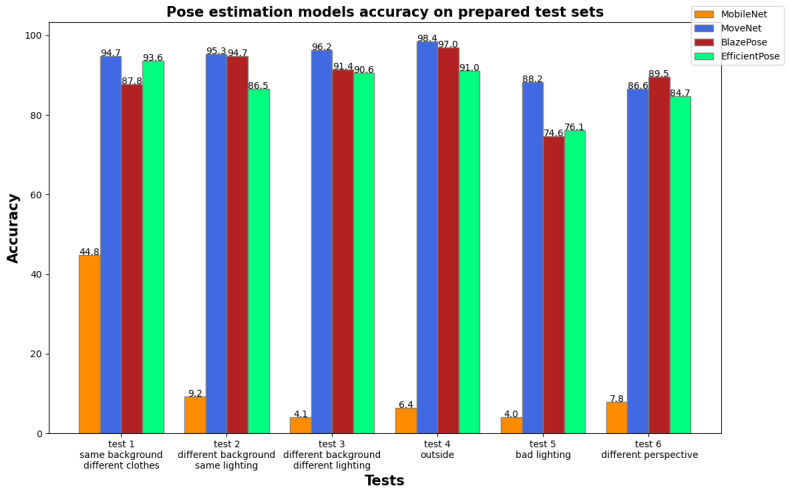


Figure 3: 3 pose estimation models and 1 image classification model accuracy score on the first 6 test sets

As we can see using MobileNet the image classification model results in only 4-9% accuracy on most tests, so out of 25 poses it is no better than a random guess, and why we do not use it in our pipeline. In contrast, using pose estimation models results in higher accuracy with MoveNet outperforming the rest (86%-98%), which is why we chose it as our default model.

4.5 Feature Engineering Method Comparison

The second component in our pipeline we evaluated was the best feature engineering method we can use, so we can improve the input of our classifier and achieve better classification. The different methods we tried are:

1. No feature engineering - not performing any processing to the keypoints produced by the pose estimation model, so we can show how using domain knowledge helps.

2. Normalizing and centering the points - as described in detail in Section 3.2, this is common practice with classifying poses and accounts for cases where a user is at different positions from the camera.
3. Normalizing + Euclidean distances - after normalizing (and centering) the points we also compute euclidean distances between them. We know distances between frames are not consistent, but we wanted to see their effect after normalization.
4. Normalizing + Angles - after normalizing the points, we compute angles between them, because we know that while the distance changes between frames, angles should theoretically remain similar as also suggested in [14].
5. Normalizing + Pairwise distances - as described in Section 3.2, after normalizing the points we compute the distances between each pair from 13 points, to see if the differences between joints improve the classification.

We show the accuracy of a basic MLP network on the validation set during training and using the same pose estimation model with different feature engineering methods. Results can be seen in Figure 4.

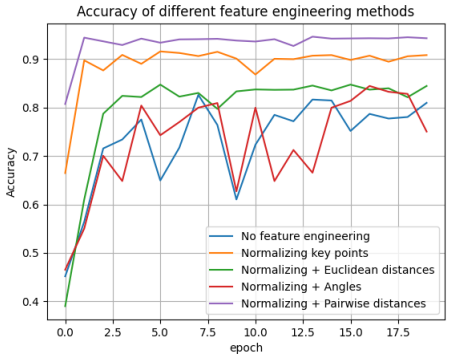


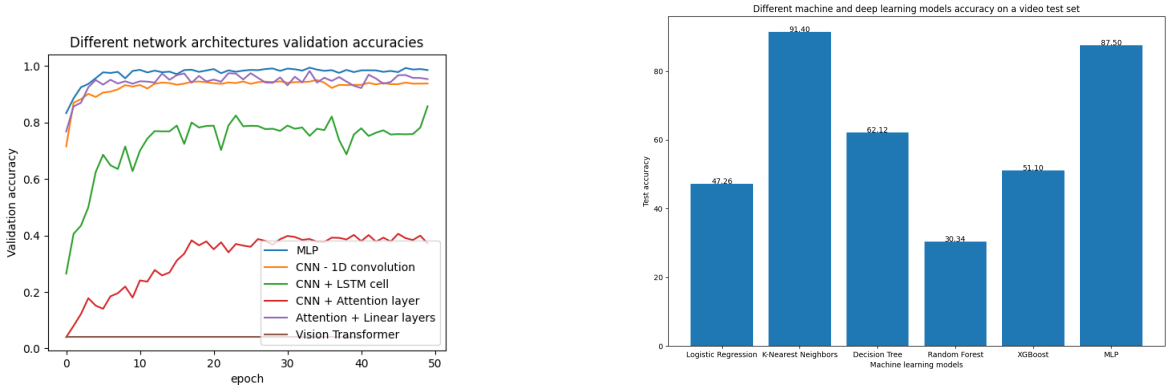
Figure 4: feature engineering methods accuracies on the validation set during training

We can see the effect of normalizing and centering the points as compared to not using any feature engineering. Additionally, we can see that while theoretically, angles should remain consistent, their effect is similar to not using any feature engineering at all. As for euclidean distances, while better than angles, they only worsen the performance of normalization. However using pairwise distances results in the highest validation accuracy of about 95% and improves upon only normalizing the points, so it is our default feature engineering method.

4.6 Classifier Comparison

The third component of our pipeline we evaluated was our classifier. While we initially had good results with only an MLP as suggested in [13], we wanted to increase our metrics scores by testing several machine and deep learning models. We tested and optimized with a grid search the following machine learning classifiers: Logistic Regression, K-Nearest Neighbors (KNN), Decision Tree, Random Forest, and XGboost since they were proven to classify numerical data well with KNN suggested in [14]. In addition, we used the following

deep neural network architectures: MLP, a network comprised of 1D-convolution (CNN for brevity), CNN with an LSTM cell, CNN with an attention layer, Attention with linear layers, and a type of vision Transformer [25]. We used these architectures since some were proven to work well with classifying yoga poses in [15] and also LSTM was suggested in [10] (while Attention layers weren't suggested we wanted to test it since it is often chosen over LSTM). The accuracy of the network architectures on the validation set, as well as the accuracy of the machine learning classifiers (with MLP) on test 13 can be seen in Figure 5.



(a) Deep learning models accuracies

(b) Machine learning models accuracies

Figure 5: Different machine and deep learning classifiers accuracies

We can see that while some network architectures succeed in classifying the different poses on the validation set like only using 1D convolution or using Attention with linear layers, the best performing architecture is the basic MLP network. However, when comparing the MLP to the machine learning classifiers, we can see that it achieves higher accuracy than all of them except the KNN classifier, yet they are pretty close.

This lead us to further investigate the differences between them in the rest of the test sets, and we saw that the KNN classifier reaches higher accuracy, precision, and recall at every test set. Thus the trivial course of action was to choose the KNN classifier as our default classifier, however, let's remember that the KNN classifier observes the training data at evaluation time to make a prediction, and its drawback is that the inference time grows linearly with the training data. So in a problem like ours, where we do not know how much data (poses) will be in advance, we cannot guarantee real-time using a KNN classifier.

Thus we wanted to find a solution that has a consistent inference time similar to the MLP, but improves upon it to achieve similar results to that of the KNN. This is where our neural network ensemble comes in. As described in Section 3.3, we use a sacking ensemble (performed better than other ensemble types as detailed in Section A.3), since it generalizes better than a single network, and the accuracy of the ensemble, the KNN, and a regular MLP on all 13 test sets are shown in Figure 6.

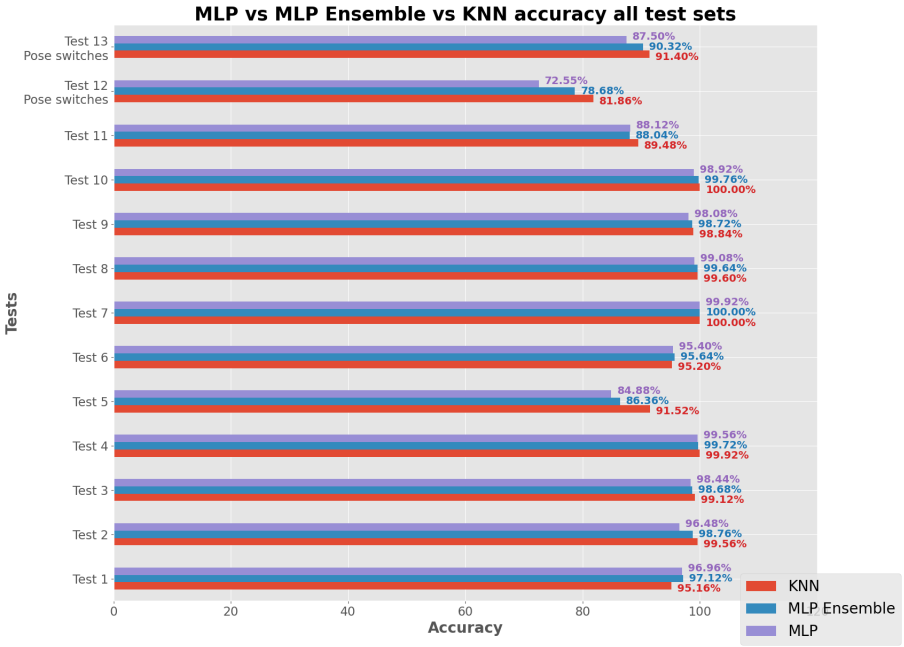


Figure 6: Comparison of the accuracy of the MLP, MLP Ensemble, and KNN on all custom test sets

We can see that the ensemble does better than a single MLP and is comparable to the KNN, having higher accuracy on some tests and lower on others. Specifically, both achieve more than 95% accuracy on most of the tests, where the only cases where both have difficulty reaching the 90% accuracy mark are tests where there is very bad lighting which makes it harder to detect keypoints on the body by the pose estimation model. To make the final decision between them, let's look at Table 1 which shows the accuracy, precision, recall, and FPS of both the ensemble and the KNN when the training data ranges from 10 to 500 images per pose.

	Accuracy		*Precision*		*Recall*		*FPS*	
	Ensemble	*KNN*	*Ensemble*	*KNN*	*Ensemble*	*KNN*	*Ensemble*	*KNN*
10 *images*	24.82	88.18	99.28	89.88	12.91	96.80	199.97	2029.58
100 *images*	88.48	91.92	93.52	92.80	93.04	98.23	199.76	469.53
200 *images*	86.8	91.98	91.91	93.057	92.68	98.003	194.25	245.02
300 *images*	90.32	91.4	92.43	93.06	96.65	97.26	197.40	167.73
400 *images*	89.56	91.8	92.66	92.6	95.42	98.37	199.50	128.79
500 *images*	89.2	92.1	92.05	92.88	95.72	98.37	202.04	103.68

Table 1: Comparison of the MLP Ensemble to the KNN with different number of training images per pose

We can see that the KNN has diminishing returns after 100 images per pose, while the ensemble has diminishing returns after 300 images per pose - which is why we ask for 300 images. Therefore the KNN is able to run in more than sufficient FPS with fewer images, however, it is evident that the more training images there are, the slower the KNN classifier gets, while the ensemble remains at about the same FPS. So given that we do not know how many poses there are in advance, we decided the default classifier to be the ensemble, as it achieves the best results (along with the KNN) with consistent FPS (unlike the KNN). In addition, we would like to stress that in most of our experiments the KNN was fast and capable of running in real-time, so if a user does not use many poses he could run the program with different arguments to use the KNN classifier or any other classifier that we presented.

4.7 Precision-Recall Tradeoff

As we stated in Section 3.4 we use a prediction queue to help our system not simulate unwanted keypresses. To determine the optimal queue size, we tested our evaluation metrics score using the same trained model with different queue sizes, and results are in Figure 7.

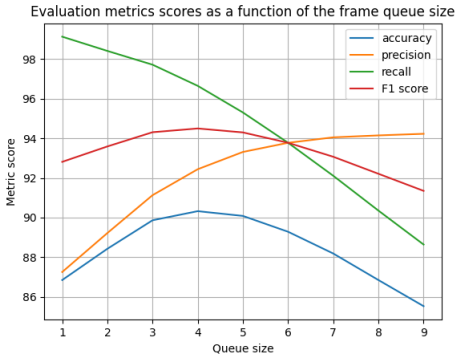


Figure 7: Evaluation metrics scores as a function of the prediction queue size

We can see that the size of the queue governs a tradeoff between precision and recall. The lower the size of the queue, the fewer frames need to have the same prediction, thus our system will simulate some keypress and as we described in Section 4.3 this means high recall. As the size of the queue increases more frames need to have the same prediction, making it harder for the system to simulate a keypress when there are pose switches and as explained in Section 4.3 this means higher precision. From the graph we can see that about 4 or 5 frames are optimal for the prediction queue in this case, however, that number could change from user to user, hence we let the user decide on this number and control the precision-recall tradeoff for what works best for him. Alternatives to the queue are discussed in Section A.2.

4.8 General Results

Now that we have our pose estimation model (MoveNet), feature engineering method (normalizing+pairwise distances), classifier (MLP Ensemble), and optimized queue size, we show

the results for all evaluation metrics when testing our final model on test 13. We believe it captures the general use case of our system the best. The model is tested on the same user it was trained on with the user performing poses in a well-lit environment that is different from the training data while also constantly changing poses. The results are in Figure 8:

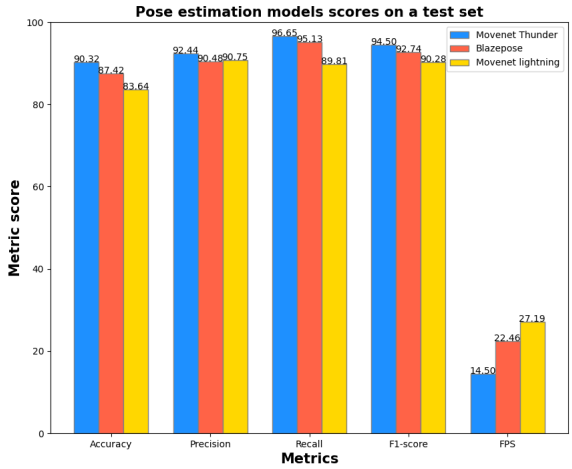


Figure 8: Evaluation metrics scores of our full model on test 13

We show the results of our system using our default pose estimation model MoveNet with its two variations - Thunder and Lightning as we offer both, but we also show the results using the BlazePose pose estimation model. We do so to show the generalizability of our solution. While we offer MoveNet (Thunder and Lightning) as a default model, our program can be run with BlazePose if it is run with different arguments. We can see that using MoveNet Thunder leads to the highest scores on all machine learning metrics passing 90% on all of them, so we can conclude it does a good job classifying poses correctly for simulating the correct keypresses and only when it neesds to, making it possible to play games with poses. However, we can see its FPS is the lowest, and while we find that more than 10 FPS is sufficient for playing games with poses, if it is not enough, a user could simply switch to MoveNet lightning with no need to train the model again and have more than 25 FPS at the cost of decreased accuracy. We can also see that there is a middle ground between MoveNet Thunder and Lightning, with BlazePose achieving better scores than Lightning yet lower than Thunder. So if inference time is insufficient with Thunder, but accuracy is not acceptable with Lightning, a user can choose to use BlazePose instead.

4.9 New Users and Poses

Finally, we would like to show the results of our system when training on one or more users and testing on others with different poses than the ones we had in mind. We trained our model on one, two, and three different users performing the G3D dataset poses and the accuracies received by testing on the 10 different subjects of the dataset are seen in Figure 9.

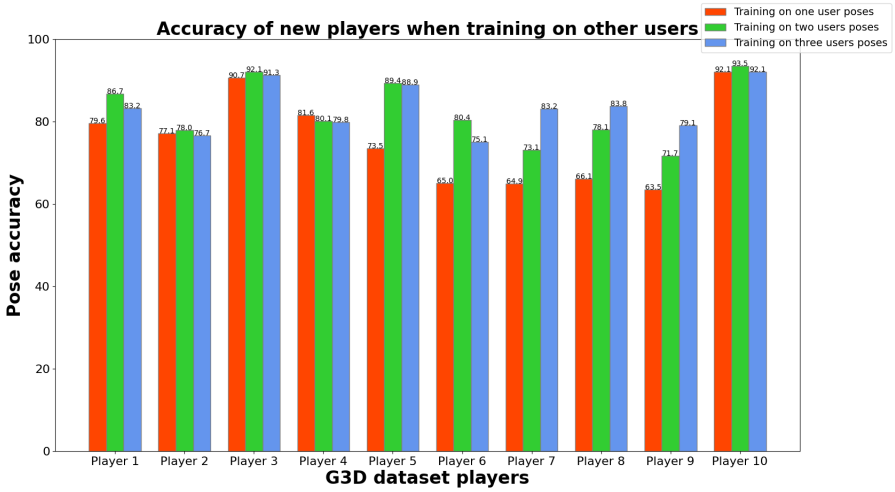


Figure 9: Accuracy of our model on G3D players when training on other users

From the graph, we can clearly see the accuracy decreased when testing on new users in comparison to our tests that focused on the same user (some players do have high accuracy because of similar body builds to the user that the model is trained on). This is expected since the goal is for the system to work on the same user, however, we can see that when the system is trained on two and three different users, the accuracy increases for almost all other test subjects reaching 77%-93%. Therefore we conclude that new and game-related poses work with our system and that it is best if the model is trained on the same user that is using it. However, if a user cannot do that, for example when a user wants his friend to use the system without having to train it again, it is preferable if the model is trained on several other users beforehand, so the accuracy for new users will be higher.

5 Conclusions and Future Work

In this project we created a human controller system that allows users to play potentially any game using body poses of their choice. We showed the use of our application in a demonstration video and when evaluating our solution on the poses we found most fitting, we managed to pass 90% accuracy, precision, recall, and F1-score on our general case all while running in real-time. We showed the robustness of our model to different environments when passing 95% accuracy on most of our test cases where very bad lighting scenarios have the biggest negative effect on performance. Finally, we showed robustness to new game-related poses from the G3D dataset, and the decrease in performance stemming from testing the model on new users it was not trained on. Our project can be used with any method we introduced and we hope our work can be extended with newer and improved pose estimation models, feature engineering methods, and classifiers, as well as using pose estimation models for more than one person, thus allowing for a two-player game experience. We also hope our work can act as a basis for future gaming-based deep learning applications.

References

[1] Shahbano Imran, “Videogame Interaction through Vision-based Input”, 2009 undergraduate Honors Thesis.

[2] Maria A Yala, “Plus+: Exploring Using the Body as a Game Controller”, 2019.

[3] Ronny Votel and Na Li, “Next-Generation Pose Detection with MoveNet and TensorFlow.js”, May17, 2021. https://blog.tensorflow.org/2021/05/next-generation-pose-detection-with-movenet-and-tensorflowjs.html (accessed Aug. 22, 2022).

[4] Dr Victoria Bloom, “Gaming Datasets”. http://velastin.dynu.com/G3D/G3D.html (accessed Aug. 22, 2022).

[5] Github, “Transfer Learning to play Pacman via the Webcam”, 2017. https://github.com/tensorflow/tfjs-examples/tree/master/webcam-transfer-learning (accessed Aug. 22, 2022).

[6] Charlie Gerard, “Playing Beat Saber in the browser with body movements using PoseNet & Tensorflow.js”, Oct. 24, 2019. https://dev.to/devdevcharlie/playing-beat-saber-in-the-browser-with-body-movements-using-posenet-tensorflow-js-36km (accessed Aug. 22, 2022).

[7] Github, “Emoji Ninja - a Peer2Peer fitness game leveraging TensorFlow’s PoseNet”, 2020. https://github.com/leungjch/posenet-game-peer2peer (accessed Aug. 22, 2022).

[8] J. Han, L. Shao, D. Xu and J. Shotton, "Enhanced Computer Vision With Microsoft Kinect Sensor: A Review," in IEEE Transactions on Cybernetics, vol. 43, no. 5, pp. 1318-1334, Oct. 2013, doi: 10.1109/TCYB.2013.2265378.

[9] Alex Kendall, Matthew Grimes and Roberto Cipolla "PoseNet: A Convolutional Network for Real-Time 6-DOF Camera Relocalization." Proceedings of the International Conference on Computer Vision (ICCV), 2015.

[10] S. Singh et al., "Action Replication in GTA5 using Posenet Architecture with LSTM Cells," 2021 2nd International Conference on Intelligent Engineering and Management (ICIEM), 2021, pp. 544-549, doi: 10.1109/ICIEM51511.2021.9445358.

[11] Gursimar Singh, Rohan Shekar, “Pose2Play”, 2021. https://devpost.com/software/pose2play#updates (accessed Aug. 22, 2022).

[12] Valentin Bazarevsky, Ivan Grishchenko, Karthik Raveendran, Tyler Zhu, Fan Zhang, and Matthias Grundmann. 2020. BlazePose: On-device Real-time Body Pose tracking. arXiv preprint arXiv:2006.10204(2020).

[13] Tensorflow, “Human Pose Classification with MoveNet and TensorFlow Lite”. https://www.tensorflow.org/lite/tutorials/pose_classification (accessed Aug. 22, 2022).

[14] ML Kit “Pose Classification Options”. https://developers.google.com/ml-kit/vision/pose-detection/classifying-poses (accessed Aug. 22, 2022).

[15] Kothari, Shruti, "Yoga Pose Classification Using Deep Learning" (2020). Master's Projects. 932.

[16] Jason Brownlee, “Stacking Ensemble for Deep Learning Neural Networks in Python”, Dec. 31, 2018. https://machinelearningmastery.com/stacking-ensemble-for-deep-learning-neural-networks/ (accessed Aug. 22, 2022).

[17] Hinton, G.E., Srivastave, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R.: Improving neural networks by preventing co-adaptation of feature detectors. In: arXiv:1207.0580 (2012)

[18] Lutz Prechelt. Early stopping-but when? Neural Networks: Tricks of the trade, pp. 553–553, 1998

[19] Microsoft, “Comparison of XInput and DirectInput features”, 2022. https://docs.microsoft.com/en-us/windows/win32/xinput/xinput-and-directinput (accessed Aug. 22, 2022).

[20] V. Bloom, D. Makris and V. Argyriou, "G3D: A gaming action dataset and real time action recognition evaluation framework," 2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2012, pp. 7-12, doi: 10.1109/CVPRW.2012.6239175.

[21] V. Bloom, V. Argyriou and D. Makris, "Hierarchical transfer learning for online recognition of compound actions", Computer Vision and Image Understanding, vol. 144, pp. 62-72, 2016.

[22] Groos, D., Ramampiaro, H. & Ihlen, E.A. EfficientPose: Scalable single-person pose estimation. Appl Intell 51, 2518–2533 (2021).

[23] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei, and Y. Sheikh. OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields. In arXiv preprint arXiv:1812.08008, 2018.

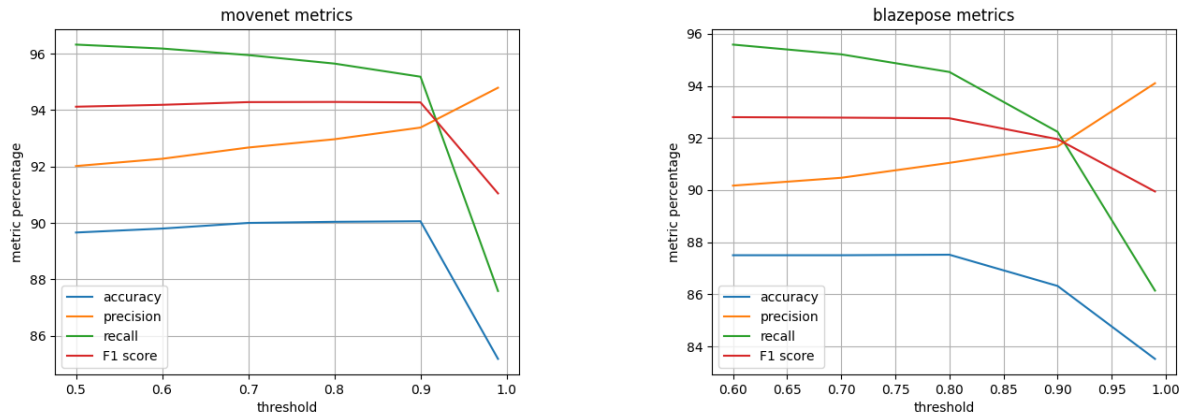
[24] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. CoRR, abs/1704.04861, 2017.

[25] Khalid Salama, “Image classification with Vision Transformer”, Jan. 18, 2021. https://keras.io/examples/vision/image_classification_with_vision_transformer/ (accessed Aug. 22, 2022).

A Appendix

A.1 Prediciton Threshold

As stated in Section 3.3 the network outputs a softmax vector with probabilities for every pose, and we take the argument with the highest probability. Nevertheless, if the network is not sure what pose to predict, then the softmax vector will hold low probabilities and we would not want to simulate a keypress. To find the optimal threshold size we ran our network on different threshold sizes, once again with different pose estimation models to see if there are differences, and the results can be seen in Figure 10.



(a) Using MoveNet

(b) Using BlazePose

Figure 10: Evaluation metrics score as a function of the prediciton threshold

We can see another take on the precision-recall tradeoff, however here it is clear that the optimal threshold size is 0.9 in both cases. In addition, it is reasonable to give the network a high threshold to pass, otherwise, any pose could be predicted and on average we find that this is the optimal threshold a prediction probability should pass.

A.2 Prediction Queue Alternatives

One might wonder if there are better alternatives to prevent simulating unwanted keypresses than the prediction queue and the prediction threshold.

One such alternative we tested was to create an extra pose that consisted of pose switches (where a user constantly changes between poses) as part of the training data. This way the network could learn what a pose switch is and when predicting it no keypress will be simulated.

A second alternative is a combination of the prediction queue and the extra pose to better avoid unwanted keypresses.

We measured the evaluation metrics of using neither (so only the base model should pass the prediction threshold of 0.9), using the prediction queue, using the extra pose, and using both, with results shown in Figure 11.

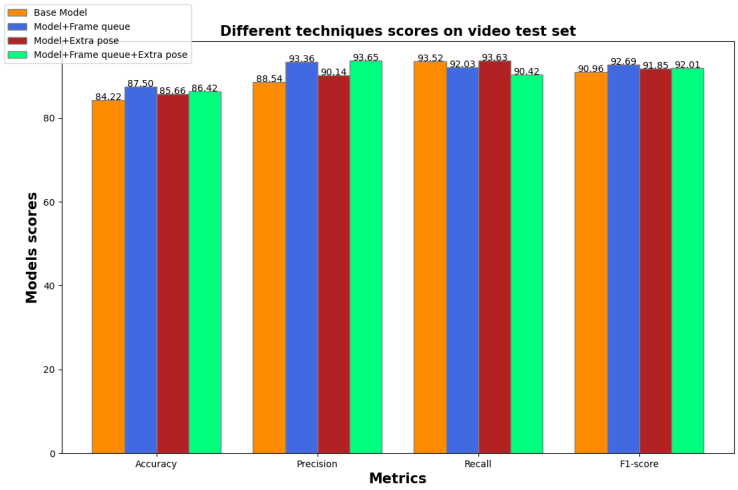


Figure 11: Comparison of prediction queue alternatives' evaluation metrics scores

We can clearly see that using at least some kind of method on top of the prediction threshold is useful since all metrics scores are higher in comparison to the base model. On top of that, we can see that both the prediction queue and the extra pose approaches work, and a combination of them works as well. However, we can see that just using the prediction queue results in the best scores and there is no merit in learning an extra pose. Additionally, we think it is confusing to ask the user for such a pose since it is not well defined what he should exactly do, and performing some movements between the poses could negatively affect his other poses as well.

A.3 Ensemble Alternatives

Deep learning neural network models are highly flexible nonlinear methods that learn via a stochastic training algorithm, and often suffer from high variance. A neural network ensemble trains multiple models instead of one to combine their predictions and reduce the variance of a single network, as well as generalize better.

There are different types of neural network ensembles. One is a model averaging ensemble where multiple sub-models are trained on the same dataset and their prediction is combined. In our case where we predict a pose, the prediction is the argument with maximum probability after summing the probabilities of each ensemble member.

A limitation of this approach is that each model has an equal contribution to the final prediction made by the ensemble. So a second ensemble is a weighted average ensemble, which weighs the contribution of each ensemble member by the expected performance of the model on a holdout dataset.

A further generalization of this approach is replacing the weighted sum model with any learning algorithm. This is the third ensemble called a stacking ensemble as described in Section 3.3. In stacking, the meta-learner takes the outputs of sub-models as input and attempts to learn how to best combine the input predictions to make a better output prediction.

We tested each method to find the best ensemble type with 3 MLP sub-models, as more resulted in an insufficient FPS. When we tried to find the best weights for the 3 sub-models in the weighted average ensemble using a grid search we got the weights $\begin{bmatrix} 0.0 & 0.0 & 1.0 \end{bmatrix}$. Namely, only one model is affecting prediction, so while it is the best out of the three, it is no different than the prediction of a single network. Nevertheless, we compared all three ensembles to find the optimal one for our problem, and the accuracy of the three ensembles on all test sets can be seen in Figure 12.

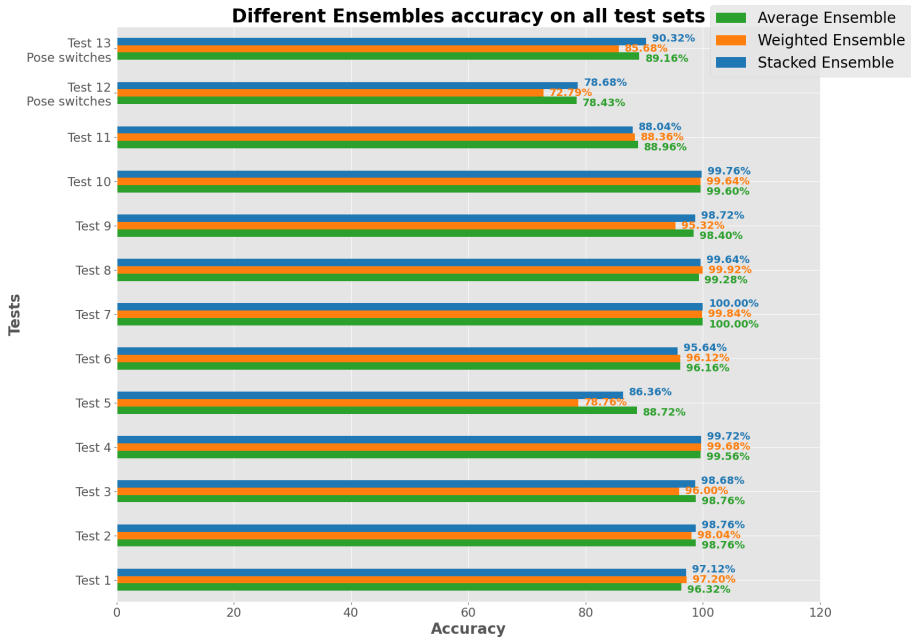


Figure 12: Accuracy of Average Ensemble, Weighted Ensemble, and Stacking Ensemble on all custom test sets

We can see that the weighted average ensemble has significantly less accuracy on some tests such as 5, 12, and 13 as expected from the high variance of a single network. In contrast, the average ensemble and the stacking ensemble both perform better and have similar accuracies. However, since the stacked ensemble performs better on tests 12 and 13 where the user performs pose switches, we find that the stacked ensemble of MLPs is more fitting as the default network architecture.

In addition, one may think that it is better to use a deeper network for better prediction instead of an ensemble. Albeit after testing many MLP architectures the best performing one was the same architecture we use as a sub-model for the ensemble - a fully connected neural network with two hidden layers and a dropout of 0.5 after each hidden layer.