Hand Pose Imitation

By L N Saaswath, Somnath Sendhil Kumar, Atul Kumar, Yash Garg

ABSTRACT

Our project is about training a CNN based model and using it to control a robotic hand simulation and imitate the pose of a real hand in the input image. We used Reinforcement Learning to do so as traditional methods would require us to provide ideal joint angles or we would need 3D positions of a few points on our hand to get the Inverse Kinematics of the system, which needless to say are either difficult to fetch or to set up. So we have used an **OpenAI** Environment (**gym-handOfJustice**) which was set up by us with **PyBullet** as the physics simulation engine and tried training with **SAC** (**Soft Actor-Critic**) with a custom Reward Function defined as *negative of the sum of the absolute difference between masks of robotic and the real hand images*.

HAND IMITATION

Human hands have a variety of muscular movements and are very complex in nature, making a robot learn from these may assist in many applications.

In this Age Of Robotic Automations having a robot for automating a certain task is easy and even combining a lot of small challenging logical tasks can be done with Machine Learning very easily, but that requires supervision for it to train or advanced machinery to track few points on our hand for the robot to learn from it. So what we aim to make is a simpler approach to do it with only an image feed of our hand and reward-based learning.

OUR APPROACH

We tried to keep the process of building and training a Robotic arm as minimal as possible. We used Reinforcement learning as a means of supervision to the model i.e. We defined a Reward Function which correlates to an error in the alignment of both the hands. Using the traditional methods to train the robotic arm by Machine Learning would need us to provide the ideal set of Joint angles or sophisticated techniques to get

3D Coordinates of a few points on our hand to get a reference. We used traditional colour segmentation to obtain a mask of our hand and the robotic hand which is used in the Reward Function, which can be defined as *negative of the sum of absolute difference of the masked images*.

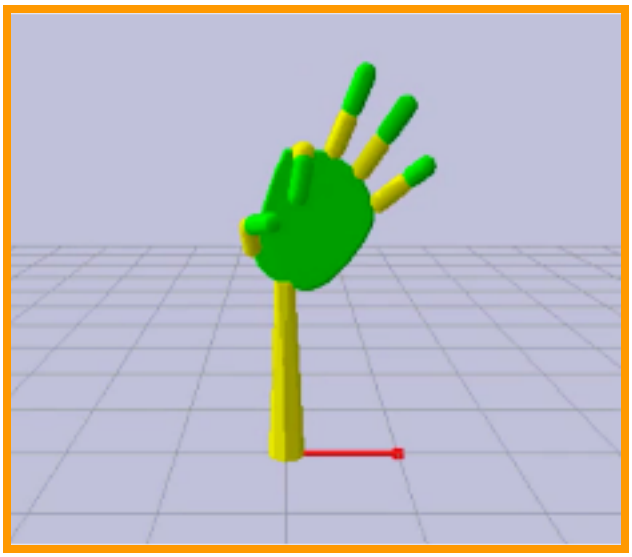
$$r(RealHand, a) = -\sum |Robotic_{mask}(a) - RealHand_{mask}|$$

This function is not in any way a representation of the error in the real 3D hand pose but rather the error in the projections of the hand poses, which on analysing is not a bad representation of a 3D hand model as a hand has constraints to move freely which would mean a particular projection can only be achieved with a particular (or a very small set of) pose(s).

KEY CONCEPTS USED

Environment and PyBullet (physics simulation engine)

We set up our environment in the OpenAI gym as this would help in having a simple interaction between the environment and the user. We have used PyBullet for all the physics simulation, We have used a Hand Model with 12 Joints to give us a basic representation of a hand. We used two bones instead of three in a finger as the topmost phalanx has a very little contribution to the net pose of a hand and which would exponentially increase the number of states. We used basic and realistic constraints for the joints so that it can imitate simple hand poses.



We set up our environment in OpenAI under the name of HandOfJustice. In our environment, we have a continuous Action Space of the 12 joint angles of the Robotic hand. Our Observation space is two images appended on the *axis=1*, the Observation is the image of the hand pose to be imitated and the image of the robotic hand trying to imitate it. The Reward Function is the same as defined above that is *negative of the sum of absolute difference of the masked images*

Ex: here the resolution of the image is 256x256

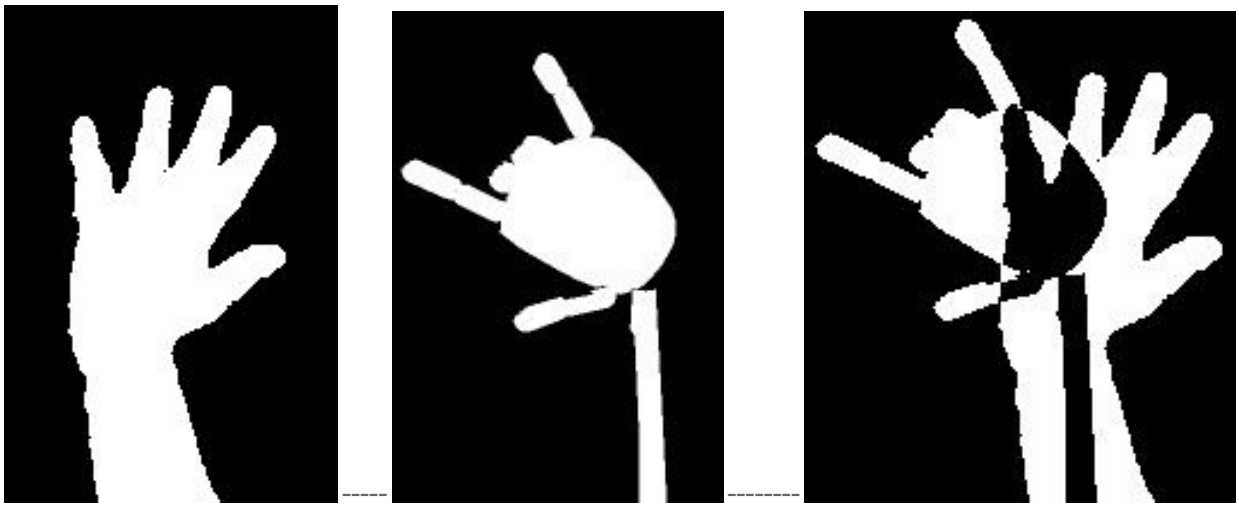
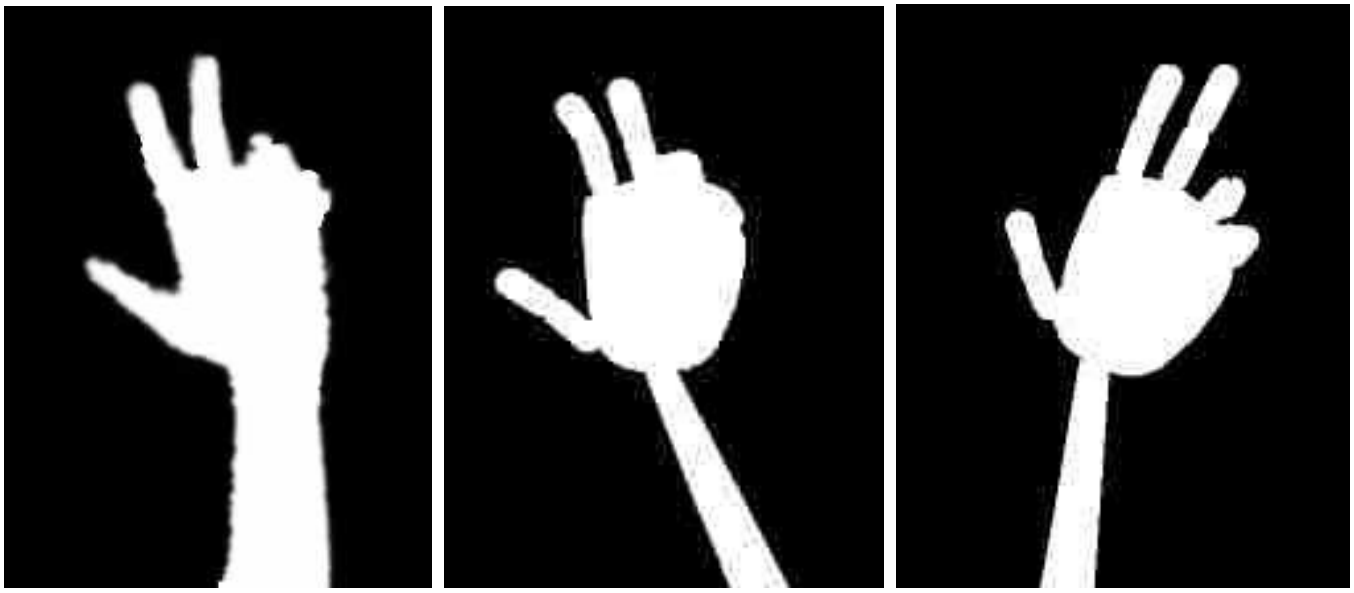


Image1 Image2 Resultant image

Reward = - sum of all points in the resultant image = -9468

The episode would last as long as the absolute reward in a state is greater than a certain value of epsilon, where epsilon is to check if the robotic arm is close enough to given hand pose and also to compensate any error in masking the hands. The value of epsilon determines the rate at which an episode ends and how accurate the model is. For training we would decrease the value of epsilon periodically to increase exploration for a higher reward at a time when it's trained well enough to get a similar orientation to that of our hand very easily which would have been the previous terminal state, thereby leading to exploration when it's more likely to reach the optimal policy.



The real hand to be imitated epsilon=60 epsilon=550

These are the terminating state for the input on the left and the respective epsilon values

Above you can see how value of epsilon can determine the different terminal states so training with a decreasing value of epsilon helps the model even learn the finger positioning with greater accuracy. (These values are for 56x56 image)

There more parameters for the environment which can be seen on its [GitHub page]

Image processing

The image processing technique was a fairly simple one using **OpenCV**.

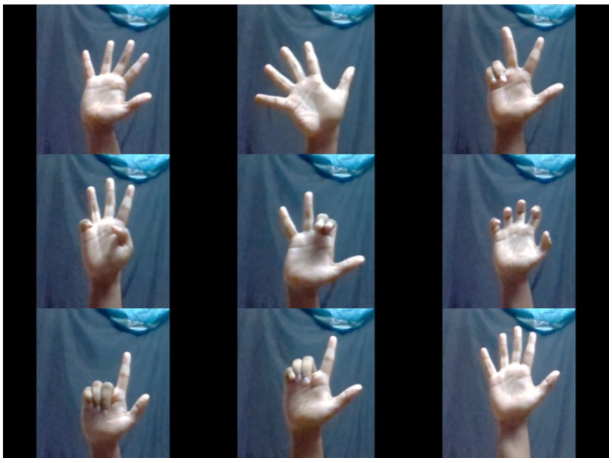
Once the images were loaded they were cropped out to an automated Region of Interest (RoI) based on the occurrence of a hand in the region . The region where we place our hand is rectangular in nature. So, we padded the rectangle to make it a square.

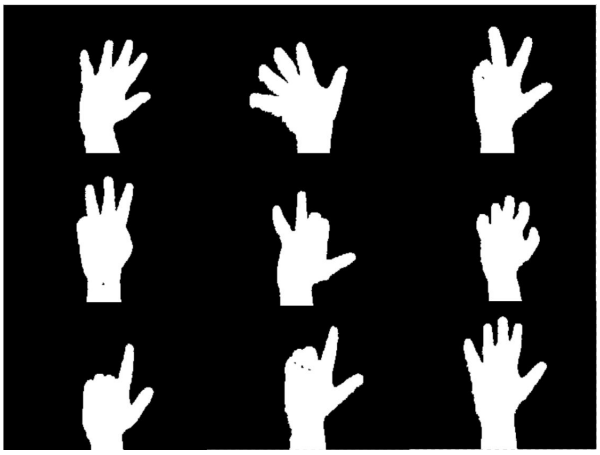
The images of the hand were then loaded into 56x56 resolution and stored into a training space. And ended up collecting a set of around **50000 images** of hand poses. Though the environment supports whatever resolution one wants to train or run on it which has to be specified by passing the resolution parameter to it.

The BGR images were converted to LAB colour-space for masking skin colour as LAB has orthogonal control over light with respect to color segmentation. With a suitable range for the skin, the hand was masked. The image was then dilated to fill in the black spots in the hand mask.

The resulting image was passed through a gaussian blur to smoothen outer areas of the mask.

The directory of images was passed as an object of ***cv2.VideoCapture(<-directory->)*** as this gives the user a lot of flexibility to use any manner of Stream of Images for training or using the environment efficiently.





The main limitation of traditional color segmentation is that it doesn't generalize from hand to hand .. and even the lighting conditions or camera module can bring in a lot of variance to the image, Hence we included a parameter in our environment called **preprocess** which takes in a function pointer which should threshold the hands in the images that the user want to train or run the environment with, if the user wants to.

Reinforcement learning

So our main goal was to train a CNN which could output a set of Joint angles for joints of the robotic arm with only an image of the robotic and the real arm. So to make our lives easier we choose to do it with Reinforcement learning as it would reduce our burden of providing supervision directly.

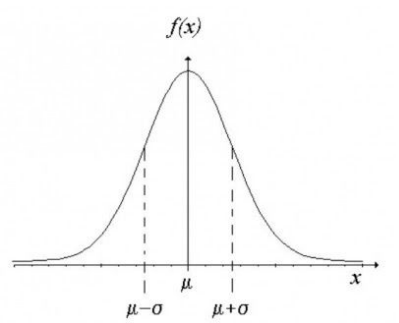
For training, we used a CNN which was trained with **Actor-Critic**, A State Of the Art RL technique, which backpropagation through the CNN based on the Return (G) for action (a) taken in a state (s) with respect to a baseline (bn) in this case it's the expectation of the return for the same state (s) which is the Value function for that state (V(s))

Some Theory about the Actor-Critic method (Though this might not be necessary to be presented here).

In this approach we have two models, the Actor and the Critic .In layman term the Actor takes an action and then is judged with a baseline which is given by the Critic and then improves on it.

Diagram of a basic Actor Critic Model

Actor Model



Here as we are using continuous Action Space, we predict the normal distribution of each joint angle with Mu and Sigma where Mu denotes the mean and Sigma denotes the variance of the distribution. The Actor Model is evaluated based on the **TD error** which is the *Return (G(s)) obtained for the state by taking a certain action (a) minus the expectation of return (E(r)) in that state.*

TD error

$$\delta = r(S_t, a) + \gamma * V(S_{t+1}) - V(S_t)$$

Which is the most fundamental evaluation in RL, but the objective of Actor Model is

rather proportional and not equal to TD error. So the actual gradient of the objective of the Actor Model is

$$\nabla_{\Theta} J(\Theta) = E[\nabla_{\Theta} log(\pi_{\Theta}(s, a)) * \delta_{\pi}]$$

Giving a nice update of

$$-log(N(a|\mu(S_t), \sigma(S_t)))$$

To the Actor Model for it to back propagate and update all the weights

Critic Model

Critic Model on the other hand estimates the **Value function** for a state or even called the Advantage function for a state which determines the Expectation of returns for that state. Here the Critic is evaluated based on the return that the agent got by going through all the steps till the current state with respect to the predicted Expectation. We use Mean Squared Error as the cost function for the Returns till this state .

We did use the a pre-trained CNN for both Actor and Critic models, the **MobilenetV2** as it can't be expected from a Reinforcement learning algorithm to completely train a CNN to even do even a basic Object detection as the extra head room to search would allow the RL algorithm to end up searching in an infinitely big State Space for a long long time....

CONCLUSION

To sum it up our expectation from the project was more than what it performs but the experience of learning something new filled it. The vision we had for the model was to have instantaneous reflex to our changes , but ended up being a process which takes few steps to reach the endpoint which is not a problem till it takes steps somewhere along the path but when, it randomly moves away from the path and comes back to the path of end point. The problem of instantaneous reflex can be fixed with either keeping gamma as a smaller value or making the Reward function even a function of no of steps taken. Due to the accuracy that can be achieved with this method,it is not worth spending a lot of time perfecting it. Bringing me to its biggest problem that is, our model is a function of epsilon which means reducing it and training would make the model explore for higher reward, but by reducing the value of epsilon substantially the error

that was allowed for bad masking or any other kinds of dissimilarity between the hands is gonna become a uncertainty for the model to explore, a good example of this is for a finger till the movement performed is significant it can be tracked and would be able to make the simulation to go to a fairly closer to terminal state else the movement would go unnoticed.

For the future work we would love to play with the behavior of the model on how it is exactly affected by the value of epsilon and how it proceeds on further exploration.

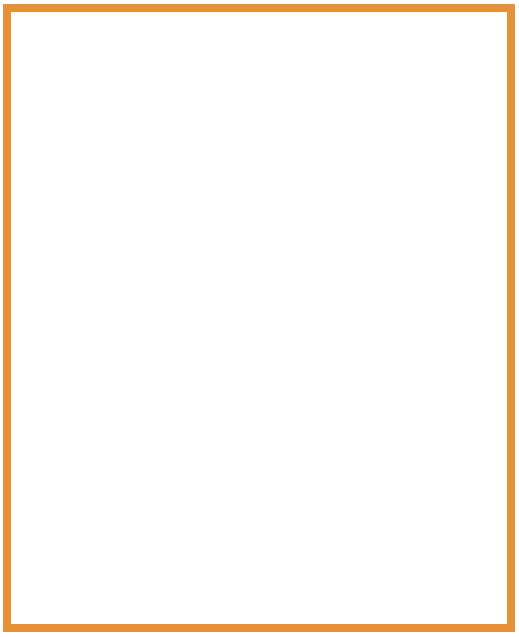
We would also like to use PID controllers rather than setting the jointAngles, maybe then the output would be stable and would be closer to a real world application. But this was actually the Backup Idea for our primary idea which is suffering from a lot of errors and could not be completed in time, Hence thinking a lot less about this project.

The whole project is hosted at The Github page to see the implementation and the technical description about the project.

REFERENCE

Actor Critic For Continuous Action Space ,by Andy Steinbach

OpenAI Environment with PyBullet , by Mahyar Abdeetedal

Thank  You