ME5418 - Machine Learning in Robotics

Final Project

Guided Pushing of ball by 7-DOF Robotic Arm

By:
Anurag Roy - Group 27 - A0304443N

Introduction

This project involves developing and training a reinforcement learning (RL) model to control a **7-degree-of-freedom (7-DOF) robotic arm** in a simulated environment. The robotic arm is tasked with performing a pushing operation, where it **interacts with a ball and guides it toward a target location**. This scenario presents challenges such as precise motion control, dynamic environment interaction, and reward optimization, making it an ideal candidate for advanced RL algorithms.

Reinforcement learning is particularly well-suited for this task because it excels in environments where explicit programming of all possible scenarios and outcomes is infeasible. RL allows the agent to learn optimal behavior through interaction with the environment, leveraging trial-and-error to improve over time. This makes RL advantageous for robotic control tasks, where the state and action spaces are high-dimensional, and the environment is often complex and dynamic. Unlike traditional control approaches, which rely heavily on pre-defined rules and precise modeling, RL adapts to uncertainties and unanticipated challenges in real time.

By employing RL, this project not only enables the robotic arm to master intricate movements but also contributes to advancing autonomous decision-making capabilities in robotics. The application of RL to robotic tasks has long-term benefits, including improved adaptability, reduced reliance on task-specific programming, and the ability to generalize learned skills across similar tasks. In this context, successfully training a robotic arm to achieve its objective underscores the transformative potential of RL in automating complex physical interactions.

Conventional Algorithms

Arms with 6 or more DOF can move their end effector in any direction, but path planning still requires human effort. Algorithms like Rapidly-exploring Random Trees (RRT) or Probabilistic Roadmaps (PRM) can plan the robot's trajectory for tasks like hitting a golf ball. However, in dynamic environments, where the goal position and object types might change, (ball) these algorithms may struggle. Reinforcement learning (RL) is more suitable as it enables the robot to learn optimal behaviors through interactions and rewards. Model Predictive Control (MPC) can also be used to predict future states, but it becomes computationally expensive in high-dimensional spaces, such as a 7-DOF robotic arm operating in environments with changing conditions. Thus an approach with RL enables learning robotic arm movements that can adapt to changing conditions without the need for manual tuning like in conventional approaches and be less computationally expensive.

Environment and RL Cast

The environment for simulation and training of our 7-DOF was built and improved upon gymnasium-robotics's Fetch robot, by changing the area where the robot operates in a large area in which we have a golf ball that needs to be pushed towards the black marker considered the 'hole', with our cup-shaped end effector. The environment is simulated and rendered using *Mujoco Simulator* to bring forth an accurate representation of our task

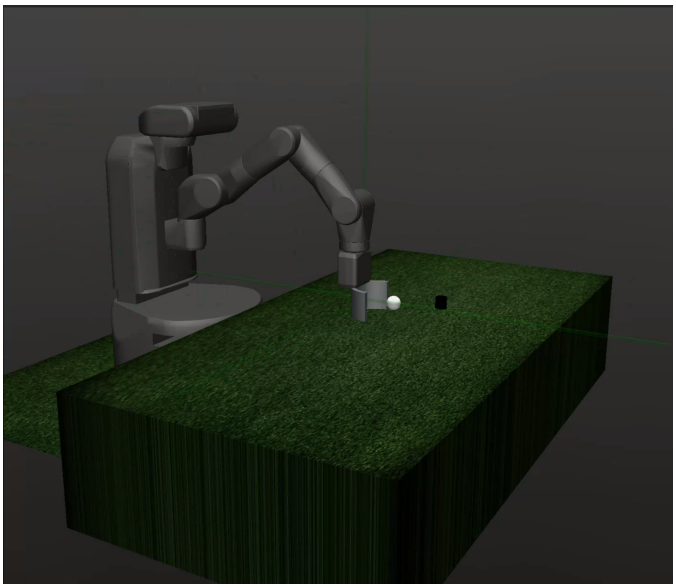


Figure 1 - 7 DOF Robot Arm pushing a ball towards hole marker in Mujoco simulator

Our previous report was not completely clear in presenting our RL Cast thus we have listed it down again. Our Observation space is a vector of 25 continuous numerical features representing the environment such as coordinates of end effector, position of ball and its relative position to the end effector, the ball rotation, the ball linear velocity, ball angular velocity, end effector velocity. Lastly there is also achieved goal and desired goal which are vectors of 3 continuous numerical features each.

Our Action Space is a vector of 4 continuous numerical features in which Three dimensions represent the Cartesian movement (dx, dy, dz) of the end effector within a bounded range and one dimension controls the opening/closing mechanism of the end effector.

Lastly our Reward Structure is a **dense** reward structure :

$$d = ||\, achieved\ goal - desired\ goal \,||_2$$

The Euclidean distance between achieved goal and desired goal

We also explored our training with a **sparse** reward structure, which will be compared later :

$$If\ d < distance\ threshold,\ reward = 0\ else\ reward = -1$$

Neural Network Architecture

Our Neural Network architecture is present in the **“PPO_network.py”** under the class *PushNetwork()*. Our neural network consists of multiple sub-networks that process the **observations** and **goal** inputs separately before combining the features to make predictions for both the **policy (actor)** and the **value function (critic)**. It utilizes a shared feature extraction approach followed by distinct branches for policy and value estimation.

Figure 2 - Neural Network Architecture

Our neural network architecture is present in our neural network report too, however we have made some changes which we like to discuss here.
First, an overview of our architecture:

1) The observation processing network, the input is an observation vector of size 24 from our *obs_dim* as per our observations requirement. We have a **linear layer** that transforms input from size 24 to 256, then **layer normalization** that normalizes the output of the linear layer to stabilize learning. Then **ReLU** activation to introduce non-linearity, and

then a **second layer** from size 256 to 128, with **second LayerNorm and ReLU.** Output size is 128.

2) The goal processing network, the input is a concatenation of *achieved_goal* and *desired_goal*. Each of them are of size 3 (*goal_dim)*, thus input size is 6. The layers are similar to ABOVE of *obs_net* and the output is a processed goal feature of size 128.

We would like to emphasize the separation of our observation and goal processing network as it reduces interference between unrelated features, improving representation quality and learning efficiency.
The change here also is that we added a **multi-headed attention layer** and according to our research it improves the ability of the network to focus on relevant observation-goal relationships dynamically.

3) Then the Policy Network (Actor), the input is the combined feature vector of size 256, with layers of **two linear layers with LayerNorm and ReLU**. The output layer transforms from 256 to *action_dim* (4), producing the action mean of size 4.
4) The Value Network (Critic), the input is the same combined features as the policy network, the layers are similar to the policy network but the **final output layer** transforms from 256 to 1. The 1 scalar value represents the value of the current state.

Again we emphasize the importance of distinct networks for policy and value functions as the actor focuses solely on learning the optimal policy and the critic evaluates the states without interference from policy gradients.
Another change we made to these two networks is that we added **residual blocks** for residual connection as it facilitates deeper architectures by mitigating our issue of vanishing gradients and encourages reuse of features learned in earlier layers. Our residual block has one fully connected layer with size of input and output of 256, and LayerNorm and ReLU.

Lastly in our forward loop, we process the inputs through the neural network and compute the action mean, action standard deviation and value state for our PPO algorithm to use in the next section.

Learning Agent

Our learning agent, also present in "**PPO_Network.py"** , defines a Proximal Policy Optimization (PPO) which was chosen for its stability in training with continuous observation and action spaces. PPO is effective and easier to implement compared to DDPG or TRPO and is inherently exploratory and generates stochastic policies. Furthermore, the clipped objective function in PPO helps prevent large policy updates which ensures smooth convergence and provides stability during training. The objective function is also responsible for maintaining balance between exploration and exploitation which is essential in Reinforcement Learning.

Our learning agent is still the same [Refer to Learning Agent report] as we believe PPO is the one of the optimal algorithms for our use case, thus only a short explanation is provided on how the algorithm works.

In our *PPOAgent()* class, we have our **Action selection** where the input is the *Observations*, *Achieved goal and Desired goal.* In this function we use our neural network's outputs *action_mean*, *action_std* and *value* for the current state, and then Gaussian distribution is created using *action_mean* and *action_std.* An action is sampled from the distribution, and the log probability of that action is computed. The function then returns *action* - the action sampled, *action_log_prob*- the log probability and value- Predicted value of the state.

Next, we have **Action Evaluation** where the input is the batch of observations, goals and actions. The network predicts the *action_mean, action_std and values*, and the gaussian distribution computes the outputs *action_log_probs*, *values* and *dist_entropy.*

Then **Loss Computation** where inputs are the batch data : *observations, actions, old log probabilities, returns and advantages,* and *PPO hyperparameters* like *clip_epsilon* and *value_loss_coeff.*

The **Policy Loss** is calculated by computing the ratio of new to old policy probabilities :

$$\text{Ratio} = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$$

The **Clipped Surrogate Objective** encourages the policy to stay within a small update region to maintain stability:

$$L^{CLIP} = E[min(ratio \cdot advantage, clip(ratio, 1 - \epsilon, 1 + \epsilon) \cdot advantage)]$$

The **Value Loss** is the mean squared error between predicted value and returns and the **Entropy Loss** adds a regularization term to encourage exploration.

Total Loss = $Policy\ Loss + c1 \cdot Value\ Loss - c2 \cdot Entropy\ Loss$**.**

Thus the outputs are total_loss, policy loss and value loss.

Training Process

Our training workflow is described in **"setup_ppo".py** to train our agent in our 7-DOF robotic arm pushing environment using the PPO algorithm.

1) **Registering the Environment** - For defining and interacting with our environment, we use the *gymnasium* library. This registers our custom environment *PushingBallEnv* with gymnasium and creates an instance of the environment and allows visualization
2) **Hyperparameters initialization** -

A) Learning Parameters

Learning rate for the optimizer - *learning_rate*, Discount factor for future rewards - *gamma*, and Generalized Advantage Estimation smoothing factor - *gae_lambda*

B) Optimization Parameters

PPO clipping parameter to ensure updates remain stable - *clip_epsilon*, Entropy coefficient for encouraging policy exploration - *entropy_coef*, Gradient clipping threshold - *max_grad_norm*

B) Training Parameters

Training episodes - *epochs* , Timesteps for policy update - *num_steps_per_update* , Number of PPO update epochs per batch - *ppo_epochs* , Batch size - *mini_batch_size*

3) **Interactions Storage** - We have a class called *RolloutBuffer()*, which serves as a temporary memory for storing the agent's interactions with the environment during an episode. Once the buffer is full, it is used to calculate advantages, returns, and gradients to update the neural network.

4) **Training Loop** -

Our training loop runs for a set number of episodes, and with each episode start, we reset the environment to its initial state.

Action selection (*select_action)*- Within the loop, the agent selects an action based on the current state (observation, achieved goal, desired goal) using the policy network. The network outputs the mean and standard deviation of the action distribution and an action is sampled from the distribution as mentioned before in the *Learning Agent* section.

Interaction (*env.step(action))* - The action is passed to the environment, and the agent receives the next state, reward and indication if the episode has ended (goal reached or maximum steps within an episode). This interaction data is stored in our buffer *RolloutBuffer().*

PPO Update - After a certain number of timesteps, the network is updated using this collected data to compute the cumulative discounted rewards for each state and the advantages, which is how much better the agent performed compared to the critic's value predictions. The advantages are normalized as large values can cause the gradient updates to overshoot, leading to unstable training and very small values may lead to insufficient updates, slowing down learning,

Batch optimization - The buffer data is divided into batches and the policy and value networks are updated iteratively. This ensures that the agent improves its policy while staying within an acceptable region to prevent drastic updates, trains the critic network to accurately predict state values and encourages exploration by penalizing low-entropy policies

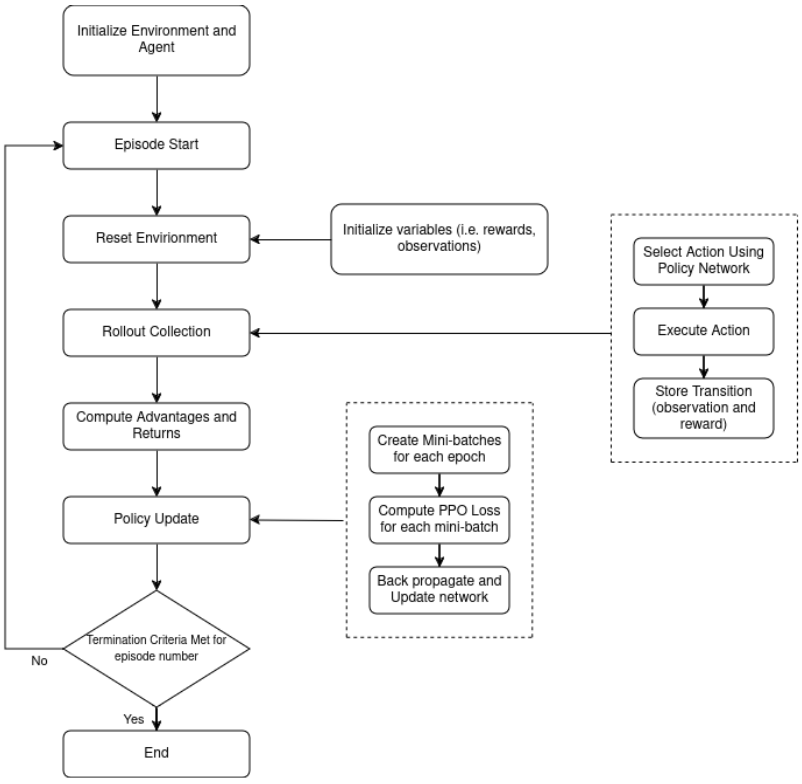


Figure 3 - Flowchart of Training Loop

Model Save and Visualizing Results - After all training episodes are completed, the trained policy network is saved for testing using *torch.save()*, and metrics such as Rewards per episode, policy loss, value loss and total loss are visualized.

Testing -All the testing is done in "**evaluated_model.py"** on our environment for a specified number of episodes, and various metrics such as episode rewards and number of steps are printed

Training Results

The model was trained in the below environment with the following value of parameters:

max_steps = 100 , **num_epochs** = 1000 , **batch_size** = 64 , **learning_rate** - 0.003 5. **num_steps_per_update** = 200, **ppo_epochs** = 10, **clip_epsilon** = 0.2 , **max_grad_norm** = 0.5, **entropy_coef** = 0.01

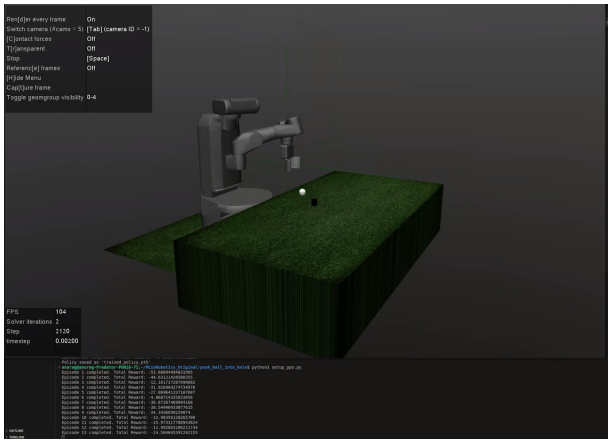


Figure 4 - Training Visualization(**PPOTrainingVideo.webm** in folder)

Hyperparameter tuning with different values and this is the most optimal set we have achieved even though there might be more optimal values. We experimented with 500, 1000 and 10000 episodes, and got the best results from 1000 episodes run. 10000 episodes showed us that the rewards kept on fluctuating around a small range after around the 1200 episode mark, thus we did not see the need to train beyond 1000 episodes. The training results are shown below using a dense reward structure for non-random positions of ball and hole :

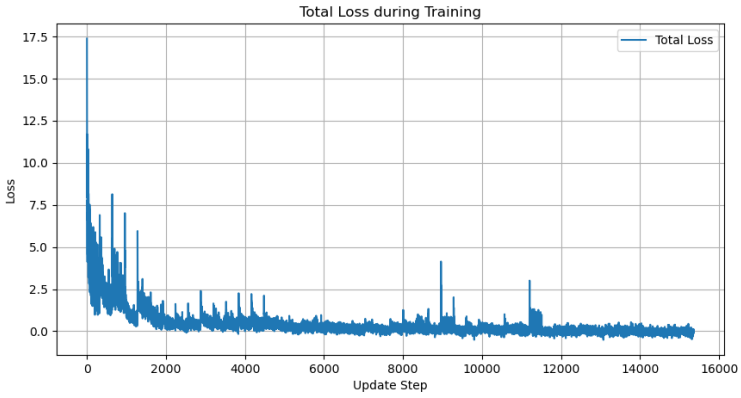


Figure 5 - Total Loss During Training

Figure 6 - Policy Loss During Training

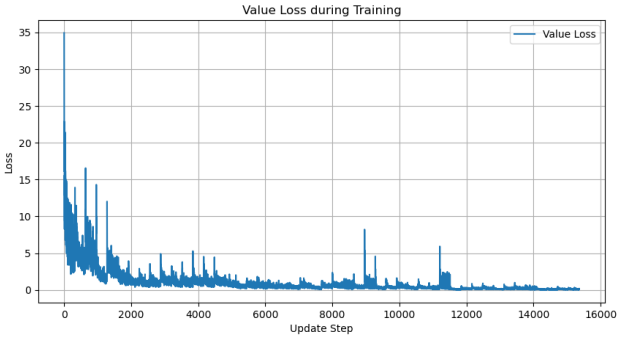


Figure 7 - Value Loss During Training

The **total loss graph** is a measure of the sum of policy loss, value loss with a weightage and entropy loss with a weightage. Our **policy loss graph** shows a fluctuation between positive and negative values as policy loss represents how the policy is improving relative to the old policy.Sometimes actions taken under the new policy result in higher rewards than the old policy (policy loss <0) and other times the opposite (policy loss >0).

The **value loss graph** decreases over time as it represents how well the value function is predicting the actual returns. Unlike the policy loss, the value loss has a clearer target: minimizing the **Mean Squared Error (MSE)** between predicted values and actual returns.

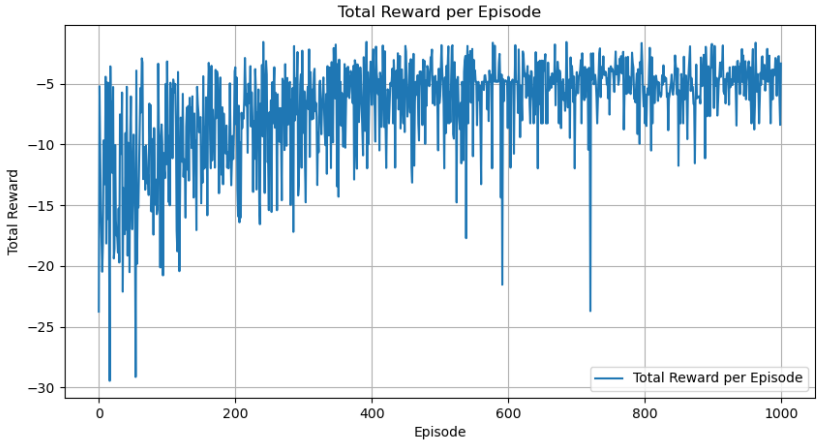
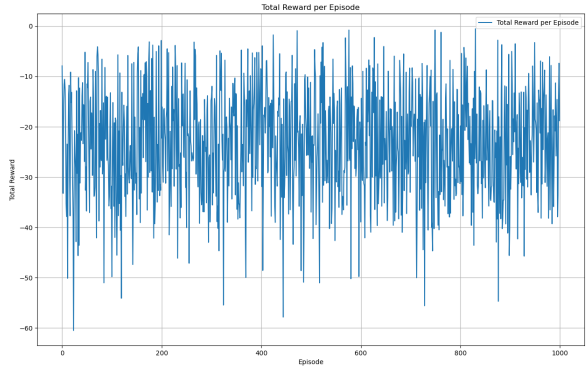


Figure 8 - Total Reward per Episode During Training

The **reward graph** shows the total rewards per episode, in which our reward is the Euclidean distance between achieved goal and future goal. The reward is added every step of the episode until either success or the max steps in the episode are reached.

The graph fluctuates a lot initially but steadily goes up as in the beginning the policy is initially exploring different actions, leading to variable rewards. As training progresses, the agent transitions towards exploiting the learned policy, causing less variability.Fluctuations persist later on but are smaller, as it is still exploring within some limits. These fluctuations could be reduced with future works of adjusting hyperparameters even more such as learning rate or advantage normalization. **This is the policy that is used for testing and comparing to stable-baselines.**

Exploration on Dense Reward with random positions and Sparse Rewards



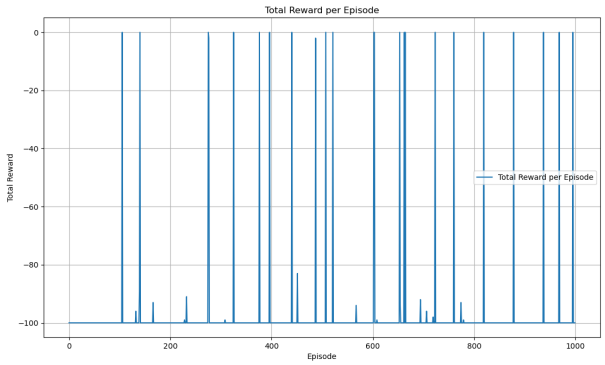


Figure 9 - Total Reward During Training for Random Positions **Figure 7 - Total Rewards for Sparse reward structure**

We have also decided to explore other reward structures for learning more on the differences. The above figure on the left shows the reward graph for **dense rewards with random positions of ball and hole** and shows a lot of fluctuation throughout the 1000 episodes showing that it requires a lot more time for training in order to first locate the ball and then move it to the goal. More steps might be required per episode too in order to provide more exploration.

The above figure on the right shows the sparse rewards structure where the reward is -1 every step if it is not within the threshold around the goal. We see that most episodes it stayed at -100 (due to 100 steps an episode), because it does not have information on the quality of its action since the reward doesn't change with any accept if it's at or near the goal. Research shows that this would ideally create a more optimal policy but requires more training time and incorporates other methods such as Hindsight Experience Replay.

Testing Model and Comparisons with Stablebaselines-3

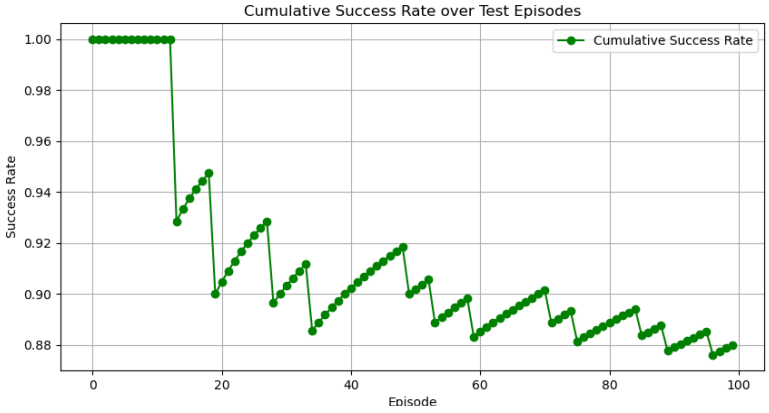


Figure 9 - Success rate over 100 episodes of reaching hole

The success rate on the left is our model evaluated on 100 episodes for non-random positions and shows a good success rate of 88% where the ball reached the hole in the file **“evaluate_model.py”**. The stablebaselines3 PPO model is evaluated in “**evaluate_model_sb3.py”** and the results showed similar success rates thus our model was

accurately modeled. The visualization (**FinalResultVideo.webm)** showed however that the movements of the robot arm with our model were quite fast and jerky as compared to the movements for stable_baselines, and this might cause problems if the policy was transferred to a real robot.

Reflection - Advantages of our RL Approach

- Unlike traditional controllers, RL operates within a model-free framework, learning optimal policies directly from interactions with the environment without explicit system calibration or knowledge of dynamics
- Excels at continuous control, reducing reliance on traditional controllers like PID or MPC.
- Leveraging neural networks, RL efficiently processes high-dimensional observation spaces and supports end-to-end training, simplifying pipeline design by mapping raw observations to actions without intermediate planners or controllers
- We moved to separating the networks for observations and goals in our architecture as it reduces interference between unrelated features, improving representation quality and learning efficiency.

Reflection - Challenges and Disadvantages of our RL Approach

- Movements of our robot arm are quite jerky and fast, and requires more insight in how to promote more smooth movements in order to emulate more realistic policies to be used in the real world.
- The robot arm after pushing the ball goes for very fast random positions as the reward structure does not care about after the task has finished which needs to be calibrated for real life use.
- There was a vanishing gradient issue where weight updates were very small and learning was not taking place, thus we used Residual blocks to solve the problem, however there is still instability in the training which has to be checked with more parameter tuning.

References

1) Base code for building the environment for 7-DOF arm with Mujoco simulator, in which we changed the environment for our particular use case of ball and hole. https://github.com/Farama-Foundation/Gymnasium-Robotics
2) Framework for code of Neural network and PPO agent www.neuralnet.ai/a-crash-course-in-proximal-policy-optimization/, github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/PPO/tf2,www.deeplearning.ai/the-batch/tag/proximal-policy-optimization/
3) Antão, L., Sousa, A., Reis, L. P., & Gonçalves, G. (2020). Learning to play precision ball sports from scratch: A deep reinforcement learning approach. *2020 International Joint Conference on Neural Networks (IJCNN)*, 1–8.