RL-Course 2024/25: Final Project Report

Emre Güçer

February 25, 2025

1 Introduction

Reinforcement Learning has been widely applied to solve various tasks, particularly in environments where states can be defined and associated with rewards. This report is for using Reinforcement Learning to teach the agent how to play air hockey. To check the implementation of this project, refer to the following GitHub repository: `https://github.com/gucere/RL`

For readers who do not know air hockey game, it is "a table game in which two players attempt to score points by using a handheld paddle to shoot a plastic disk across a surface with minimal friction into the opponent's goal" [2]. Figure 1, located at the end of Section 1.1, provides a rendered image of the game environment to aid visual understanding.

1.1 Hockey Environment

The project's baseline is defined by the *hockey/hockey_env.py* file. It handles environment creation, rendering, player and puck movements, reward evaluation, and scorekeeping. Additionally, it utilizes the *Box2D* library to simulate realistic 2D physics, ensuring accurate and dynamic interactions within the game.

To facilitate the agent's learning across various aspects of gameplay, the Hockey Environment includes three distinct training modes.

1. **"NORMAL":** A full game practice mode to simulate the experience of playing a full game.
2. **"TRAIN_SHOOTING":** A game mode to practice shooting.
3. **"TRAIN_DEFENSE":** A game mode to practice defending an upcoming attack.

These training sessions require an opponent. The environment provides 3 choices for this part as well:

1. **"HumanOpponent":** Allows the user to play against the agent.
2. **"BasicOpponent" Weak Mode:** Simple and less difficult opponent.
3. **"BasicOpponent" Strong Mode:** More complex and harder opponent.

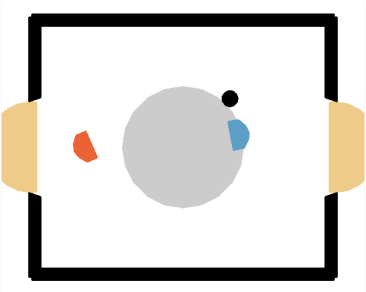

Figure 1: Rendered image of hockey environment [1]

1.2 Main Task of the Project

The primary objective of this project was to develop a Reinforcement Learning agent capable of playing air hockey within the provided hockey environment. This process can be broken down into these key steps:

1. **Selecting and Implementing the RL Model:** The first step involved choosing an RL algorithm and implementing it to serve as the agent's decision-making system. *(Covered in Section 2)*

2. **Modifying the Implemented Algorithm:** The second step involved modifying the original *TD3* implementation to enhance stability, efficiency, and generalization. Making algorithmic modifications was a project requirement to enhance the agent's learning process and optimize performance. These adjustments were designed to address gradient instability, improve weight initialization, and reduce computational inefficiencies. *(Covered in Section 2)*

3. **Implementing Training and Testing Pipeline:** The next step was to develop a training pipeline that iteratively evaluates the agent's performance, saving an improved version whenever it surpasses previous results. The best-performed model is retained for further evaluations, refinements, and participation in the tournament. *(Covered in Section 3)*

4. **Training the Agent:** The agent was trained within the hockey environment to improve its gameplay performance. *(Covered in Section 3)*

5. **Hyperparameter Tuning & Experimenting:** The final step was to optimize the hyperparameters and to experiment with different configurations to improve the agent's learning efficiency and overall performance. *(Covered in Section 3)*

2 Method

2.1 Reinforcement Learning Algorithm Selection

I chose *Twin Delayed DDPG (TD3)* as my base algorithm, because we covered *DDPG* in class, making it easier to understand compared to other alternative algorithms. I based my algorithm to Scott Fujimoto, Herke Hoof, and David Meger's "Addressing Function Approximation Error in Actor-Critic Methods" paper and their GitHub repository [4, 5].

While implementing *TD3*, I made several modifications to improve stability, efficiency, and generalization in training. Additionally, making modifications to the base algorithm was one of the requirements

of this project. These changes were introduced to address gradient instability, network initialization, and computational inefficiencies. Below, I outline the key adjustments I made to the original *TD3* implementation and explain their impact on performance.

2.2 Modifications to *TD3*

2.2.1 LeakyReLU Instead of ReLU

I replaced *ReLU* activation functions with *LeakyReLU* in both the Actor and Critic networks. The primary reason for this change was to prevent dead neurons, a common issue with *ReLU*. In standard ReLU, neurons receiving negative inputs produce zero gradients, causing them to stop learning. In contrast, *LeakyReLU* allows a small gradient flow for negative inputs, ensuring continuous updates even for negative activation values. This modification improves stability and performance.

2.2.2 Normalization of Layers

I applied *Layer Normalization* to both the Actor and Critic networks to enhance stability in training. This normalization technique stabilizes the optimization landscape, reduces sensitivity to hyperparameter choices, and maintains consistent activation statistics throughout training.

Additionally, by improving stability, Layer Normalization accelerates convergence. Since training time and computational power are limited, this change was also implemented to enhance efficiency.

An alternative approach is *Batch Normalization*, which normalizes across the batch dimension. However, unlike Batch Normalization, *Layer Normalization* normalizes across features, making it more suitable for small batch sizes.

2.2.3 He Initialization for Better Weight Distribution

To improve training stability and optimize weight distribution, I added *He Initialization*. *He Initialization* is specifically suitable for *ReLU*, *Leaky ReLU*, and their variants. It accounts for the non-linearity of rectified activations and the asymmetry introduced by *Leaky ReLU*, ensuring that the variance of activations remains consistent across layers. This mitigates issues such as vanishing or exploding gradients, which can hinder training.

An alternative initialization method is *Xavier Initialization*. However, this approach is not ideal for my code since *Xavier Initialization* assumes a symmetric activation distribution, whereas my code uses a non-symmetric activation, *Leaky ReLU*.

2.2.4 Weight Decay in Adam Optimizer

To improve the stability and generalization of the *TD3* agent, I incorporated weight decay in the Adam optimizer using its built-in *weight decay* feature. According to PyTorch's documentation [3], weight decay functions as an *L2 penalty*, discouraging large weight magnitudes and helping to prevent overfitting. By applying weight decay, the network parameters remain well-regularized, leading to smoother updates and better generalization.

2.2.5 Gradient Clipping

To prevent exploding gradients and stabilize training, I incorporated gradient clipping in the *TD3* algorithm. Exploding gradients occur when weight updates become excessively large, leading to unstable training dynamics and divergence. I set the default clipping threshold to 1.0 while allowing future users to modify this value as needed.

2.2.6 Efficient Use of Torch Tensors for GPU Optimization

To maximize computational efficiency and reduce training time, I optimized the use of Torch tensors to delegate computations to the GPU, leveraging its superior processing power. Additionally, I avoided unnecessary transfers between the CPU and GPU, as frequent transfers can significantly slow down the training process. Before making this change, my code was running with around average of 2.5 episodes per minute. After making this change, it started running with the speed of 10 episodes per minute.

2.2.7 No Explicit Save and Load Methods

In the original implementation by Fujimoto et al. [4, 5], the save and load methods were included directly within the *TD3* class. In my version, I relocated these methods to the training script instead. This change does not affect the training process, but it enhances the separation between the core algorithm and the training pipeline, particularly for model saving and loading.

3 Experimental Evaluation

This section presents the implemented training methodology of the *TD3* agent in the air hockey environment, and the agent's performance against weak and strong "Basic Opponents"

3.1 Customizability and Adaptability of the Training Pipeline

The training pipeline was designed to be highly customizable for future experiments and modifications All numeric variables can be re-edited according to performance. All of the parameters in this section are customizable by the user.

The training process for the modified *TD3* agent was structured to optimize its performance in the air hockey environment. Training was conducted over 30205 episodes, but due to the time limit it failed to reach 80000 episodes before tournament started, with each episode consisting of a maximum of 251 steps, since it is max limit of steps in the environment. The agent interacted with the environment using continuously occurring up, down, left, and right actions to acquire rewards based on match performance.

3.1.1 Agent Parameters

The agent was trained using the modified *TD3* algorithm with tunable hyperparameters. The following parameters were used in the experiments:

- **Hidden Layer Dimensions:** 256, 256
- **Discount Factor (γ):** *0.99*
- **Target Network Update Rate (τ):** *0.005*

- **Policy Noise:** 0.2
- **Noise Clipping:** 0.5
- **Policy Update Frequency:** 2
- **Gradient Clipping:** 1.0

3.1.2 Memory Parameters

The agent used a replay buffer to store past experiences and sample mini-batches for training. Replay Buffer's size is for reserving memory for a certain number of transitions. A larger size would allow recording more transitions; however, it also leads to increased power consumption and required time. The memory parameters were as follows:

- **Replay Buffer Size:** 100,000 transitions
- **Batch Size:** 32

3.1.3 Training Parameters

The training setup was structured to allow the agent to progressively learn from interactions with different opponents. A higher exploration rate allows the agent to do more random actions for more exploration; however, the agent should be exploiting as well so that it will mix current best actions with alternative actions to find a new best action combination. Since the agent will progressively get better, it should focus more on exploring first, and then on exploiting as time passes.

To assess the agent's learning progress, evaluation was performed every 100 episodes as a regular check-up or when the reward would be deemed as worth investigating, which is when the reward is at least higher than the previous best reward minus 0.5. Then that episode will be investigated for the inputted number of games. Lastly, the promising episode's quality should be investigated multiple times since the agent may get lucky or unlucky at that episode and wrongly assign rewards for that episode. A higher number of evaluations are more accurate; however, they will consume more time per episode. If it is found to be better than the current best, then it replaces the current best. Using a low value can cause an episode to be valued incorrectly and it can either fail to save a good episode or save a bad episode. This error is mentioned in 4.1.

The following configurations were used for training:

- **Training Mode:** "NORMAL" (Full-game)
- **Opponents:** *Weak Opponent* and *Strong Opponent*
- **Episodes Against Weak Opponent Before Switching:** 1,000
- **Total Training Episodes:** 80,000
- **Episode Length:** 251 steps
- **Exploration Noise:** Initial value of 0.2
- **Exploration Decay:** 0.999999

- **Minimum Exploration Noise:** 0.0001
- **Number of Games:** 200

3.1.4 Additional Parameters for Rendering and Evaluating the Best Agent

After completing some training user can choose to render their agent to see its progress. In that case, they have to use:

- **Whether To Run Best Model (Boolean):** To run the best model.
- **Whether To Render (Boolean):** To render games of the best model.
- **Number of Games:** To choose the number of games the best model will play.

3.2 Methodology of Training and Evaluation

The training process aimed to track the agent's learning progress and determine the best-performing model for the final deployment.

The process starts by getting parameters from the user. Then if the script was used before, it loads the saved best agent and memory buffer or changes the size of the memory buffer depending on input.

The next step is to start training the agent. For the inputted number of times, it will train the agent against the weak opponent. When the total number of training passes that number, it will start alternatively training against both weak and strong opponents.

When there is an episode that has a promising reward, a reward higher than the previous best minus 0.5, or the episode is at a checkpoint, every 100th episode is a checkpoint, that episode gets evaluated by the inputted number of games to get a more accurate reward value for that episode. If that value is better than the best agent, it gets to be saved over the best agent.

Additionally, at every 5 iterations, progress gets saved so that when the script gets to be run again, it can continue from where it stopped.

3.3 Performance Against the Basic Opponent

This section is for showing the performance of the current best agent against provided *Basic Opponent*s. The ideal way to test my agent's performance is to make it play against Weak and Strong opponents for a certain number of games. Since results may differ, I set my agent to play against each *Basic Opponent* 100 times and provided the results.

Testing against weak opponent:

- Agent success (number of games with positive reward) count: 29
- Average reward: -5.43
- Success rate (Percentage of games with positive reward): 29.0%

Testing against strong opponent:

- Agent success (number of games with positive reward) count: 23
- Average reward: -6.95
- Success rate (Percentage of games with positive reward): 23.0%

Overall Performance:

- Average reward across all opponents: -6.19
- Average success rate (Percentage of games with positive reward): 26.0%

```
user_variables = {
    # train and run at the end
    "whether_to_train": False, # Boolean to train
    "whether_to_run_best_model": True, #Boolean to test the current best model
    "whether_to_render": False, # Boolean to render test
    # training settings
    "total_episodes": 80000, # Number of episodes to run
    "batch_size": 32,
    "episode_length": 251, # Depth of an episode (Hockey env has max depth 251)
    "exploration_noise": 0.2,
    "min_exploration_noise": 0.0001,
    "exploration_decay": 0.999999,
    "train_against_weak_n_number_of_times": 1000,
    # running settings
    "number_of_games": 100, # Number of games to test an episode's quality or to render - a high number
    "opponents": [h_env.BasicOpponent(weak=True), h_env.BasicOpponent(weak=False)], # list of opponents
    # TD3 settings
    "discount": 0.99,
    "tau": 0.005,
    "policy_noise": 0.2,
    "noise_clip": 0.5,
    "policy_freq": 2,
    "hidden_dim_1": 256,
    "hidden_dim_2": 256,
    "learning_rate": 1e-5,
    "weight_decay": 1e-5,
    "grad_clip": 1.0,
    "leaky_relu_grad": 0.01,
    "memory_limit": 100000 # memory limit to allocate for the task
}
```

(a) Used Values

```
PS C:\Users\emreg\Downloads\RL> & C:/Users/emreg/AppData/Local/Programs/Python/Python310/python.exe
ckey-env-master/train.py
Loaded memory buffer with 12066 transitions (Limited to 100000)
Loaded agent from models/best_actor.pth and models/best_critic.pth

Agent will play 100 games against both Basic Opponents.

Testing against weak opponent:
Against weak opponent:
Agent success (number of games with positive reward) count:  29
Average reward: -5.43
Success rate (Percentage of games with positive reward): 29.0%

Testing against strong opponent:
Against strong opponent:
Agent success (number of games with positive reward) count:  23
Average reward: -6.95
Success rate (Percentage of games with positive reward): 23.0%

Overall Performance:
Average reward across all opponents: -6.192202752788684
Average success rate (Percentage of games with positive reward): 26.0%
```

(b) Results of 100 Games

Figure 2: Performance Against *Basic Opponents*

4 Discussion

4.1 Hyperparameter Tuning

Hyperparameter tuning is a key part of machine learning. Not using the right hyperparameters can lead to requiring too much time and computational time or suboptimal results.

Having a suboptimally high batch size, episode length, hidden dimensions or number of games for testing each episode are examples of requiring too much time and computational time.

On the other hand, having a low batch size, low episode number, low episode length, low hidden dimensions, not having exploration decay, or not having a reasonable exploration rate are the common causes of suboptimal results.

In addition to these mentioned reasons for suboptimal results, my training code has one additional flaw. Not checking the quality of an episode a high number of times, which is having a low number_of_games, can lead to suboptimal results as well. In the case that an episode is not checked thoroughly, it may mislabel the quality of that episode and give it a wrong reward value. This problem occurred during

my training process. I was able to find the problem and update the number of checks for each episode; however, I fixed the problem late, and this caused my agent to have a bad result in the tournament. Possible solutions to this problem are mentioned in 4.2.

4.2 Future Improvements

The ideal way to counter overwriting a good agent due to mislabeling problem would be to have a high number of comparisons, which is having a high number_of_games, before saving an agent as a new best agent and saving old versions of the best agents and comparing them before deciding which is the best.

This idea can be applied to all parameters and forcing a certain range on them. In the case that the user would provide a value out of bounds, the script could automatically assign a value and print a message to indicate it or stop the process before starting and indicate which values are out of bounds.

Lastly, since there are too many variables., users may forget to provide a value and the script may crash due to missing a variable. To prevent this, I used optional variables for most variables of most methods. This can be completed and tested more in-depth to choose more optimal default variables for all variables.

References

[1] Georg Martius. Hockey Environment - GitHub Repository. Accessed: February 26, 2025.

[2] Merriam-Webster. Air Hockey - Definition & Meaning. Accessed: February 24, 2025.

[3] PyTorch Documentation. torch.optim.Adam, 2024. Accessed: February 26, 2025.

[4] Scott Fujimoto. TD3, 2018. Accessed: February 25, 2025.

[5] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. Accessed: February 25, 2025.