# Machine Learning Project Presentation

Andrea Gennari, Alessio Maiola

## 1 The environment

The environment we chose is a simple 4-joint walker robot. The problem focuses on developing an efficient control strategy for the two-legged robot to walk across a slightly uneven terrain. The main objective is to achieve stable and continuous walking using reinforcement learning techniques.

### 1.1 Environment Setup

The bipedal walker robot is initialized in a standard upright position on an uneven terrain. The terrain consists of small variations in height to simulate a realistic walking surface. The simulation runs in discrete time steps, and each episode starts with the robot at a fixed starting point.

### 1.2 Reward Structure

The reward structure is designed to encourage stable and efficient walking. The robot receives positive rewards for forward movement and maintaining an upright posture. Penalties are applied for actions that lead to falling or inefficient movement patterns due to excessive torque usage.

### 1.3 Observation space

The observation space of the bipedal walker is quite extensive, consisting of 24 dimensions that provide a comprehensive description of the robot's state.
These dimensions include:

- **Position and velocity of the hull:** the (x, y) coordinates of the robot's central body (hull) and its linear velocities along these axes.
- **Angular orientation and velocity of the hull:** the angle of the hull with respect to the vertical axis and the angular velocity.
- **Joint angles and angular velocities:** the angles of each of the four joints (hips and knees for both legs) and their respective angular velocities.
- **Leg contact with the ground:** binary indicators (0 or 1) that signify whether each leg is in contact with the ground.
- **Lidar measurements:** distances measured by 10 lidar sensors that provide information about the terrain ahead of the robot. These measurements help the robot perceive and react to changes in the terrain.

## 1.4 Action space

The action space of the bipedal walker consists of continuous values representing the torques applied to the robot's joints. Specifically, it includes:

- **Upper joints (left and right):** two continuous values that control the torques applied to the left and right upper joints.
- **Lower joints (left and right):** two continuous values that control the torques applied to the left and right lower joints.

## 1.5 Challenges of the environment

The walker robot faces several challenges in the environment.

One major challenge is the uneven terrain. The ground is not flat and has slight bumps and dips, which means the robot has to constantly adjust its walking strategy to stay upright and move forward.

Another challenge is maintaining balance. Since the robot has only two legs, keeping its balance is naturally difficult, especially on surfaces that are not even. The robot needs to make continuous adjustments to avoid falling over.

Lastly, managing energy efficiently is crucial. The robot must use its energy wisely to keep walking for a longer time without falling. If the robot uses too much energy in its movements, it will not be able to walk steadily for a long period. Therefore, it has to find a way to move that uses the least amount of energy while still being effective.

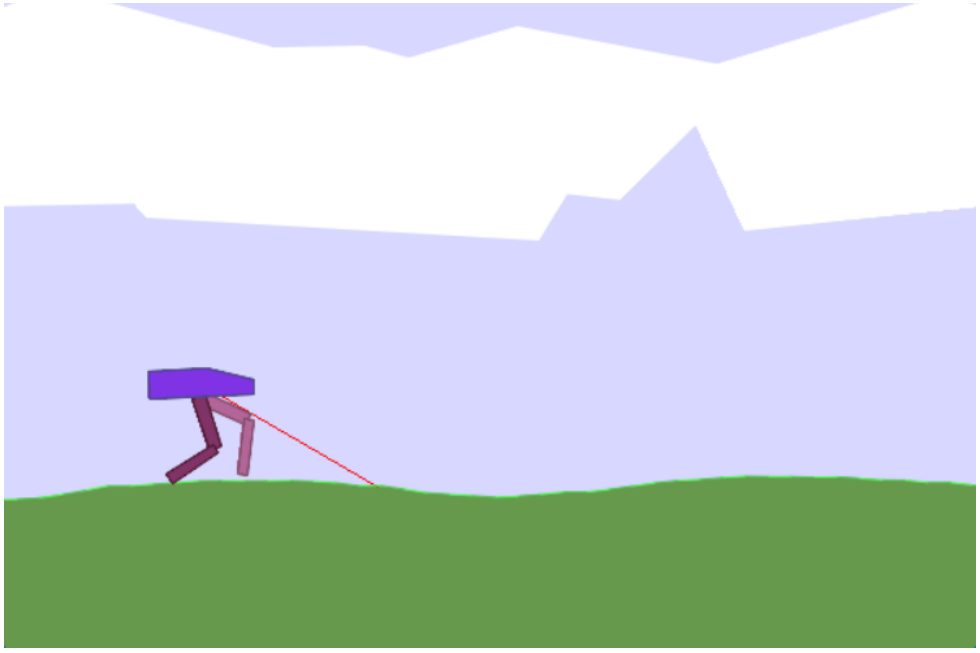

Figure 1: The bipedal walker robot in action

## 2 Solutions adopted

We will describe how we addressed this problem with the following Reinforcement Learning techniques: Q-Table, Deep Q-Network and Proximal Policy Optimization.

### 2.1 Q-Table

The Q-Table technique is based on the concept of Q-learning, which aims to find the optimal action-selection policy for an agent interacting with an environment to maximize cumulative reward. It is particularly effective for problems with environments where the number of states and actions is finite and the observation and action space are discrete.

Although both the observation and action spaces are finite, the greatest obstacle we dealt with was discretizing both of them, which makes the observation space huge (due to its high dimensionality) and also the action space difficult to explore.

In practice, by using 9 buckets for each dimension of both the observation space and the action space, we will need to explore $9^{24}$ states, where you can choose among $9^4$ actions, so visiting many times the same state is quite rare and, even when it happens, traversing extensively the action space is too expensive.

We chose an odd number of buckets to ensure that one bucket represents the value 0, as most of the dimensions in both the observation space and the action space have a symmetric range.

```
def discretizeState(self, state):
    if self.mode == 'uniform':
        discrete_state = np.round((state - self.env.observation_space.low) / (self.
            ↪ env.observation_space.high - self.env.observation_space.low) * (
            ↪ self.obs_buckets - 1)).astype(int)
    elif self.mode == 'range':
        discrete_state = []
        for i, val in enumerate(state):
            bucket_size = (self.env.observation_space.high[i] - self.env.
              ↪ observation_space.low[i]) * self.obs_buckets
            bucket_size = max(1, int(bucket_size))
            discrete_val = round((val - self.env.observation_space.low[i]) / (self.
              ↪ env.observation_space.high[i] - self.env.observation_space.low[i])
              ↪ * (bucket_size - 1))
            discrete_state.append(discrete_val)
    else:
        raise ValueError(f"Invalid mode '{self.mode}'. Supported modes are 'uniform'
            ↪  and 'range'.")
    return tuple(discrete_state)

def discretizeAction(self, action):
    discrete_action = np.round((action - self.env.action_space.low) / (self.env.
            ↪ action_space.high - self.env.action_space.low) * (self.act_buckets
            ↪ - 1)).astype(int)
    return tuple(discrete_action)

def undiscretizeAction(self, action):
    action = (np.array(action) / (self.act_buckets - 1)) * (self.env.action_space.
            ↪ high - self.env.action_space.low) + self.env.action_space.low
    return tuple(action)
```

Listing 1: Discretization Functions

To achieve effective discretization, we implemented specific functions for discretizing states and actions, as well as for converting discrete actions back into continuous actions. These functions are essential for managing the interaction between the agent and the environment in a discrete manner, enabling the application of Q-learning by mapping continuous states and actions to discrete values that the Q-learning algorithm can handle.

The ‘discretizeState‘ function converts continuous states into discrete buckets based on the chosen mode: ‘uniform‘ or ‘range‘. In the ‘uniform‘ mode, the function assigns the same number of buckets to each dimension of the state space, ensuring an even distribution across the entire range of each dimension. This is achieved by calculating the bucket index as a proportion of the state's position within the overall range of the observation space. In contrast, the ‘range‘ mode assigns a number of buckets proportional to the range of each dimension. This means that dimensions with a larger range will have more buckets, allowing for finer discretization where it is most needed.

Similarly, the ‘discretizeAction‘ function converts continuous actions into discrete buckets, while the ‘undiscretizeAction‘ function reverses this process, converting discrete actions back into their continuous counterparts.

## 2.2 Deep Q-Network

The Deep Q-Network (DQN) technique implements the concept of Q-learning using a Neural Network. This choice allows to avoid discretizing the observation space, but not the action space: the neural network will return a Q-Value for each possible action and it will be trained using the Mean Squared Error between the value returned by the network and the non-deterministic Q-Function update rule. We used a fully connected network as learning model for our algorithm.

In order to achieve better results, we applied a normalization technique to the observations before feeding them into the neural network, utilizing Welford's online algorithm.

```
class Normalizer:
    def __init__(self, num_inputs):
        self.mean = np.zeros(num_inputs)
        self.m2 = np.zeros(num_inputs)
        self.count = 0
    def update(self, x):
        self.count += 1
        old_mean = self.mean.copy()
        self.mean += (x - self.mean) / self.count
        self.m2 += (x - old_mean) * (x - self.mean)
    def normalize(self, x):
        eps = 1e-10
        mean = torch.tensor(self.mean).float().to(DEVICE)
        if self.count > 1:
            variance = self.m2 / (self.count - 1)
        else:
            variance = np.zeros_like(self.m2)
        stdev = torch.tensor(np.sqrt(variance) + eps).float().to(DEVICE)
        return (x - mean) / (stdev)
```

Listing 2: Welford's Normalization Algorithm

In addition to normalization, another critical component of our DQN implementation is the experience replay mechanism. This mechanism helps in breaking the correlation between consecutive experiences and stabilizes the training process.

```python
class ExperienceReplay:
    def __init__(self, buffer_size, batch_size=BATCH_SIZE):
        self.buffer = deque(maxlen=buffer_size)
        self.batch_size = batch_size

    def __len__(self):
        return len(self.buffer)

    def store_transition(self, state, action, reward, new_state, done):
        self.buffer.append((state, action, reward, new_state, done))

    def sample(self):
        sample = random.sample(self.buffer, self.batch_size)
        states, actions, rewards, next_states, dones = zip(*sample)
        states = torch.stack(states).to(DEVICE)
        next_states = torch.stack(next_states).to(DEVICE)
        actions = torch.tensor(actions).to(DEVICE)
        rewards = torch.tensor(rewards).float().to(DEVICE)
        dones = torch.tensor(dones).short().to(DEVICE)
        return states, actions, rewards, next_states, dones
```

Listing 3: Experience Replay Mechanism

The ‘ExperienceReplay‘ class is used to store and sample experiences from the agent's interaction with the environment. This helps in stabilizing the learning process by breaking the correlation between consecutive experiences.

Another essential part of the DQN is the Q-Network, which is a neural network used to approximate the Q-value function.

```python
class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size=HIDDEN_SIZE):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, action_dim)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Listing 4: Q-Network

The ‘QNetwork‘ class defines the architecture of the neural network used to approximate the Q-values. It consists of three fully connected layers with ReLU activation functions.

Unfortunately, despite avoiding discretizing the observation space, the performance achieved by our DQN implementation is not satisfactory due to the need to discretize the action space. This limitation often leads the agent to get stuck in equilibrium positions or local maxima. Discretization of the action space restricts the agent's ability to explore and exploit the environment effectively, especially in complex and continuous action domains. As a result, the agent may struggle to find optimal policies and tends to converge prematurely to suboptimal solutions.

## 2.3 Proximal Policy Optimization

The Proximal Policy Optimization algorithm (PPO) employs two neural networks to find the best policy in a Reinforcement Learning problem. The first network, called the actor, produces the mean value of a Multivariate Normal Distribution. In our implementation, the covariance matrix is a diagonal matrix. Therefore, we assume that each joint action has a fixed variance and that the action of each joint is independent of the others, as the covariances are all zero. When we choose an action, we sample from the distribution and compute the probability of that action.

In each iteration of the algorithm, a batch of episodes is collected. For each episode, we store the observed states, the chosen actions along with their probabilities, and the rewards, which are used to calculate the discounted rewards. The critic network is then used to determine the value function associated with the actor's choice. This value function is subtracted from the discounted rewards to derive the Advantage function. We then use the advantage to calculate the loss for the actor and the value function to determine the loss for the critic, and train the networks accordingly. Both networks are trained iteratively to improve their respective functions in approximating the optimal policy and value function.

We implemented the algorithm using a timestamp method, applying thresholds on the number of steps per episode and per batch. The only parameter is the total number of timestamps for the algorithm.

**Algorithm 1:** Proximal Policy Optimization: pseudocode algorithm

PPO

**Input :** Initial policy parameters $\theta_0$, initial value function parameters $\phi_0$

**Output:** Policy $\pi$

**for** $k \leftarrow 0$ **to** $|Iterations|$ **do**

Collect set of trajectories $T_k = \{\tau_i\}$ by running current policy $\pi_k = \pi(\theta_k)$

Compute discounted rewards $\hat{R}_e = \sum_{e=0}^{E} \gamma^e R_e$

Compute advantage estimates, $\hat{A}_e$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$

Update the policy by maximizing the PPO-clip objective using stochastic gradient ascent:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|T_k|E} \sum_{\tau \in T_k} \sum_{e=0}^{E} \min\left( \frac{\pi_\theta(a_e|s_e)}{\pi_{\theta_k}(a_e|s_e)} A^{\pi_{\theta_k}}(s_e, a_e),\ \sigma A^{\pi_{\theta_k}}(s_e, a_e) \right)$$

where $\sigma = \text{clip}\left( \frac{\pi_\theta(a_e|s_e)}{\pi_{\theta_k}(a_e|s_e)}, 0.8, 1.2 \right)$

Fit value function by regression on mean-squared error, typically via some gradient descent algorithm:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|T_k|E} \sum_{\tau \in T_k} \sum_{e=0}^{E} \left( V_\phi(s_e) - \hat{R}_e \right)^2$$

**end for**

The following code snippet contains the key aspects of our implementation of the algorithm, adapted from Eric Yang Yu's work [1].

We adapted the original code to enable to train the neural networks on GPUs, allowing the use of larger and deeper networks and enhancing computational efficiency.

In the following snippet of code, there is an implementation of the main aspects of the PPO algorithm. The main functions and their purposes are detailed as follows:

- **learn function**: This is the main function of the class, implementing the PPO algorithm.
  - It utilizes the **evaluate function** to derive the V-value provided by the critic network.
  - It calculates the current probability of the action being sampled in the multivariate normal distribution, based on the current parameters of the actor network.
- **rollout function**: This function derives a new batch of episodes.
  - It uses the **compute_rtgs function** to compute the discounted rewards for the observations in the batch.

All this information is necessary for the computation of the loss functions for both networks, which are trained in the final steps of the learn function.

```
class PPO:
    def __init__(self, policy_class, env, **hyperparameters):
        self._init_hyperparameters(hyperparameters)
        self.env = env
        ...
        self.actor = policy_class(self.obs_dim, self.act_dim).to(self.device)
        self.critic = policy_class(self.obs_dim, 1).to(self.device)  # ALG STEP 1
        ...

    def learn(self, total_timesteps):
        ...
        while t_so_far < total_timesteps:  # ALG STEP 2
            batch_obs, batch_acts, batch_log_probs, batch_rtgs, batch_lens,
              ↪ batch_rews = self.rollout()  # ALG STEP 3
            for ep_rews in batch_rews:
                rewards.append(np.sum(ep_rews))
            t_so_far += np.sum(batch_lens)
            i_so_far += 1
            self.logger['t_so_far'] = t_so_far
            self.logger['i_so_far'] = i_so_far
            V, _ = self.evaluate(batch_obs, batch_acts)
            A_k = batch_rtgs - V.detach()  # ALG STEP 5
            A_k = (A_k - A_k.mean()) / (A_k.std() + 1e-10)
            actor_losses_it = []
            critic_losses_it = []
            for _ in range(self.n_updates_per_iteration):  # ALG STEP 6 & 7
                V, curr_log_probs = self.evaluate(batch_obs, batch_acts)
                ratios = torch.exp(curr_log_probs - batch_log_probs)
                surr1 = ratios * A_k
                surr2 = torch.clamp(ratios, 1 - self.clip, 1 + self.clip) * A_k
                actor_loss = (-torch.min(surr1, surr2)).mean()
                critic_loss = nn.MSELoss()(V, batch_rtgs)
                actor_losses_it.append(actor_loss.item())
                critic_losses_it.append(critic_loss.item())
```

```
                self.actor_optim.zero_grad()
                actor_loss.backward(retain_graph=True)
                self.actor_optim.step()
                self.critic_optim.zero_grad()
                critic_loss.backward()
                self.critic_optim.step()
                self.logger['actor_losses'].append(actor_loss.detach())
            delta_t = self._log_summary()
            elapsed_times.append(delta_t)
            actor_losses.append(np.mean(actor_losses_it))
            critic_losses.append(np.mean(critic_losses_it))
        torch.save(self.actor.state_dict(), './ppo_actor.pth')
        torch.save(self.critic.state_dict(), './ppo_critic.pth')
        return rewards, actor_losses, critic_losses, elapsed_times

    def rollout(self):
        ...
        while t < self.timesteps_per_batch:
            ep_rews = []
            obs = self.env.reset()[0]
            done = False
            for ep_t in range(self.max_timesteps_per_episode):
                if self.render and (self.logger['i_so_far'] % self.render_every_i ==
                ↪  0) and len(batch_lens) == 0:
                    self.env.render()
                t += 1
                batch_obs.append(obs)
                action, log_prob = self.get_action(obs)
                obs, rew, done, _, _ = self.env.step(action)
                ep_rews.append(rew)
                batch_acts.append(action)
                batch_log_probs.append(log_prob)
                if done:
                    break
            batch_lens.append(ep_t + 1)
            batch_rews.append(ep_rews)
        batch_obs = torch.tensor(np.array(batch_obs), dtype=torch.float).to(self.
            ↪ device)
        batch_acts = torch.tensor(np.array(batch_acts), dtype=torch.float).to(self.
            ↪ device)
        batch_log_probs = torch.tensor(batch_log_probs, dtype=torch.float).to(self.
            ↪ device)
        batch_rtgs = self.compute_rtgs(batch_rews).to(self.device)  # ALG STEP 4
        self.logger['batch_rews'] = batch_rews
        self.logger['batch_lens'] = batch_lens
        return batch_obs, batch_acts, batch_log_probs, batch_rtgs, batch_lens,
            ↪ batch_rews
    ...
```

Listing 5: PPO implementation

As mentioned above, in our implementation actions are mapped onto a multivariate normal distribution, allowing for probabilistic exploration in the action space.

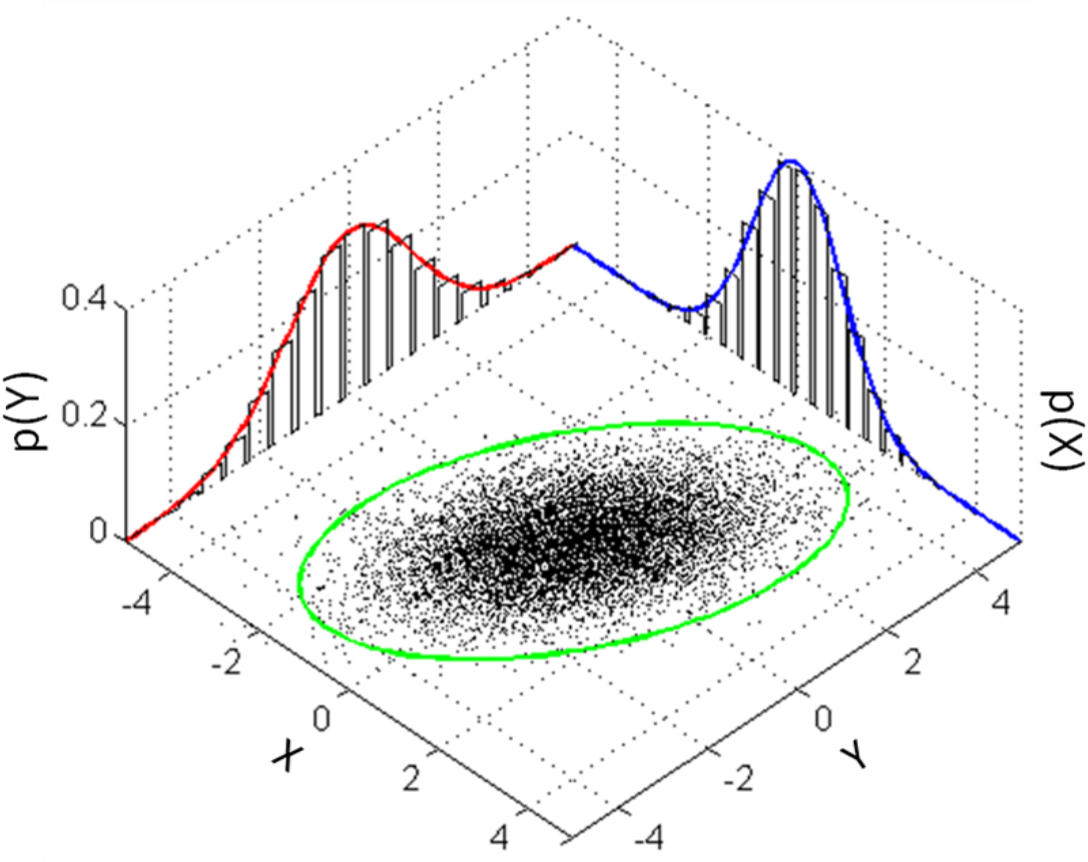


Figure 2: An example of multivariate normal distribution with $\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $\Sigma = \begin{pmatrix} 1 & \frac{3}{5} \\ \frac{3}{5} & 2 \end{pmatrix}$

In the example figure, we show a bivariate normal distribution. In our case, the distribution is four-dimensional. Importantly, the covariance matrix for our four-dimensional normal distribution is diagonal. This indicates that the four action variables are independent of each other, as each Gaussian distribution for the actions does not influence the others.

# 3 Results

## 3.1 Q-Table

After extensive experimentation with various hyperparameters, the optimal settings for our Q-Table based reinforcement learning model in the Bipedal Walker environment were identified as $\epsilon = 0.9999$, $\alpha = 0.01$, and $\gamma = 0.999$. Despite these efforts, the agent demonstrated significant limitations in its learning capability.

Due to the discretization of actions, the Q-Table approach struggled to effectively learn and perform the task. The agent primarily attempted to maintain balance but was unable to achieve meaningful walking behavior. The discrete action space was insufficient to capture the complex, continuous dynamics required for successful bipedal walking.

The use of the uniform mode, in particular, exacerbated these challenges. The uniform mode attempts to discretize the vast observation space evenly. However, this approach led to an huge number of discrete states, rendering the Q-Table intractably large and sparse. As a consequence, the agent struggled to generalize from its experiences. It failed to remain upright for more than a few seconds at a time, as the granularity of the state space prevented it from learning how to maintain balance and initiate walking.

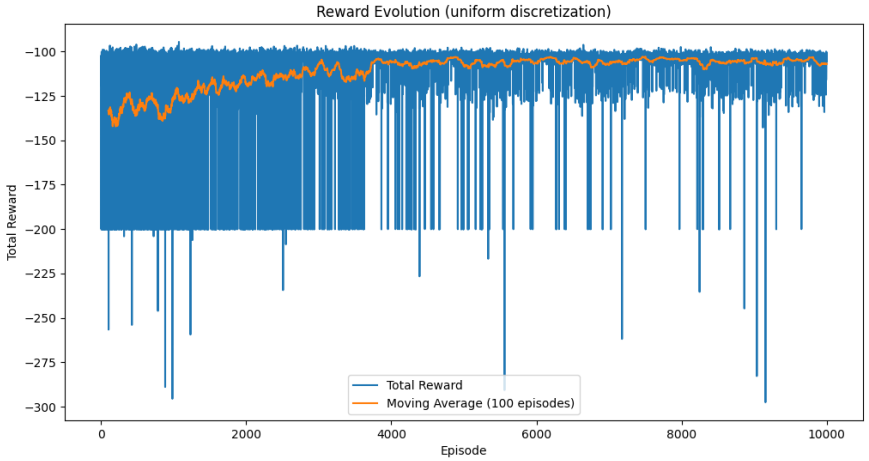


Figure 3: Q-Table learning rewards (uniform mode)

In contrast, the range mode provided a somewhat more promising, though still insufficient, alternative. By discretizing the observation space within specified ranges, this mode managed to reduce the dimensionality of the state space, but still not enough to enable efficient learning. The agent in the range mode was occasionally able to find stable positions and exhibit rudimentary forward movement. However, these movements were highly inefficient, resulting in very low rewards. The agent frequently became stuck in a balancing position, failing to make consistent progress and often becoming immobilized within the environment.

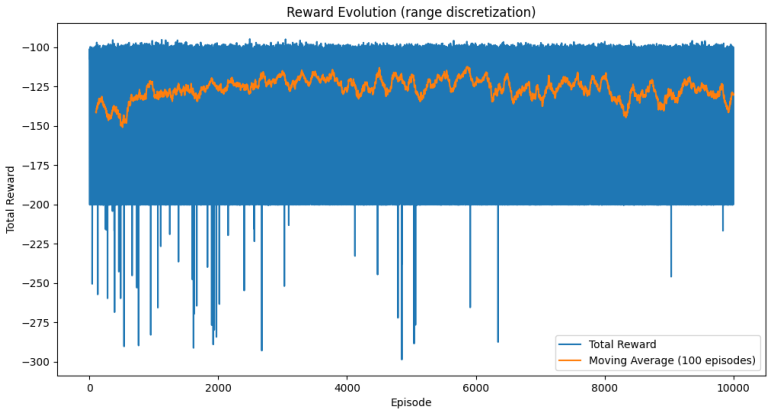


Figure 4: Q-Table learning rewards (range mode)

As a result, the uniform mode consistently receives a higher reward because the agent quickly falls down within a few seconds. In contrast, the range mode agent manages to progress further in some episodes but ends up with a lower overall reward due to the higher torque applied, receiving more penalties despite covering more distance.

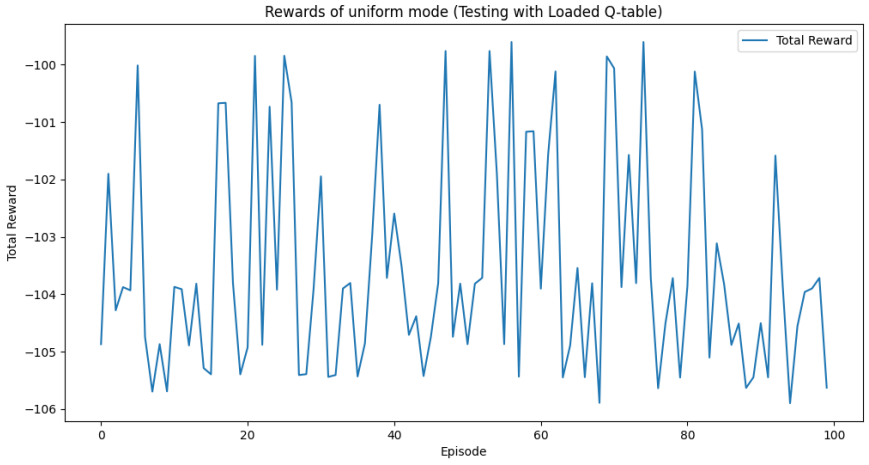


Figure 5: Q-Table testing rewards (uniform mode)

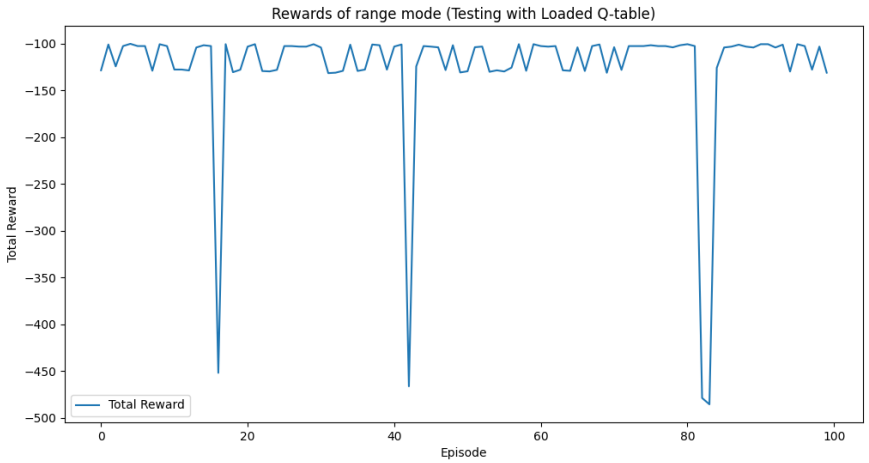


Figure 6: Q-Table testing rewards (range mode)

In conclusion, the Q-Table based approach proved inadequate for the Bipedal Walker task. The discretized actions severely limited the agent's ability to learn and execute the necessary movements, resulting in an overall poor performance. This underscores the need for more advanced methods, such as function approximation techniques, to handle environments with continuous action spaces.

## 3.2 Deep Q-Network

The optimal hyperparameters for the DQN model were found to be $lr = 0.001$, $\epsilon = 0.999$, $\alpha = 0.001$, and $\gamma = 0.999$. The comparison between normalized and non-normalized state inputs revealed a significant performance difference.

With state normalization, the DQN demonstrated superior learning efficiency and performance. The primary reasons for this improvement are:

1. **Improved Gradient Descent**: Normalizing input features stabilized the gradient descent process, ensuring balanced contributions from all features and preventing dominance by features with larger scales.

2. **Reduced Training Time**: The normalized states facilitated faster convergence, allowing the agent to learn effective policies more quickly due to a more uniform state space.

3. **Consistent Reward Scaling**: Normalization provided a consistent scale for rewards. When state inputs are normalized, the learning algorithm can better interpret the rewards it receives. This is because the normalized states ensure that no single feature dominates or skews the interpretation of the reward.

Quantitatively, the performance of the normalized DQN was comparable to that of the Q-Table approach. Moreover, the DQN faced challenges similar to those observed with the Q-Table model, depending on the chosen hyperparameters. Specifically, issues such as the agent getting

stuck in equilibrium points, encountering local minima, and falling almost immediately persisted. These challenges highlight that while normalization improved stability and efficiency, the underlying complexities of the Bipedal Walker environment continued to pose significant learning difficulties for both approaches.

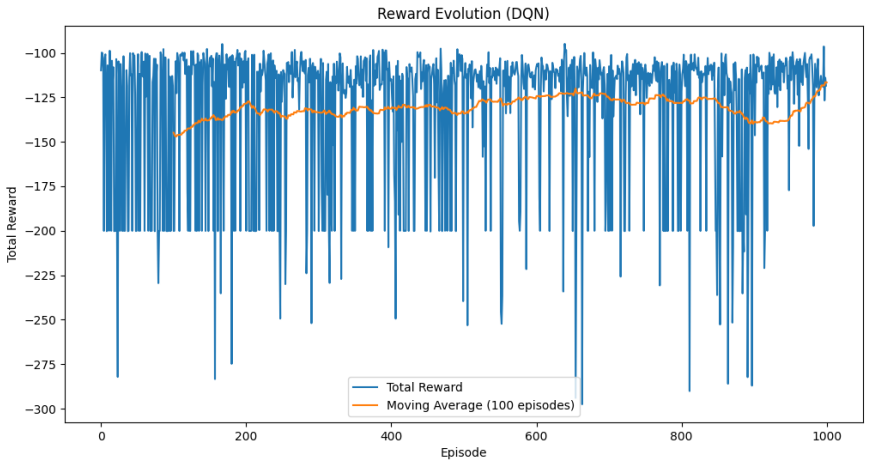


Figure 7: DQN learning rewards.

In any case, the DQN method requires discretization of the action space, which inherently makes the training process slower. This discretization limits the granularity of the actions the agent can take, contributing to the overall slower convergence and performance challenges.

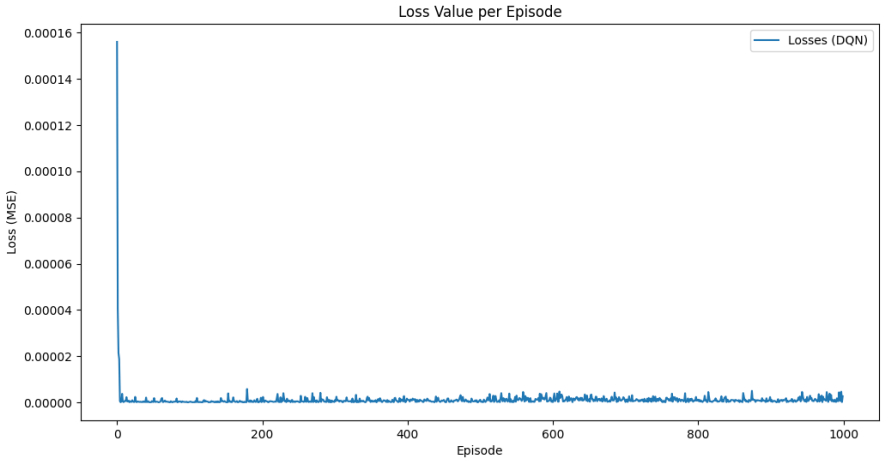


Figure 8: Learning curves showing the loss over training episodes for the DQN model.

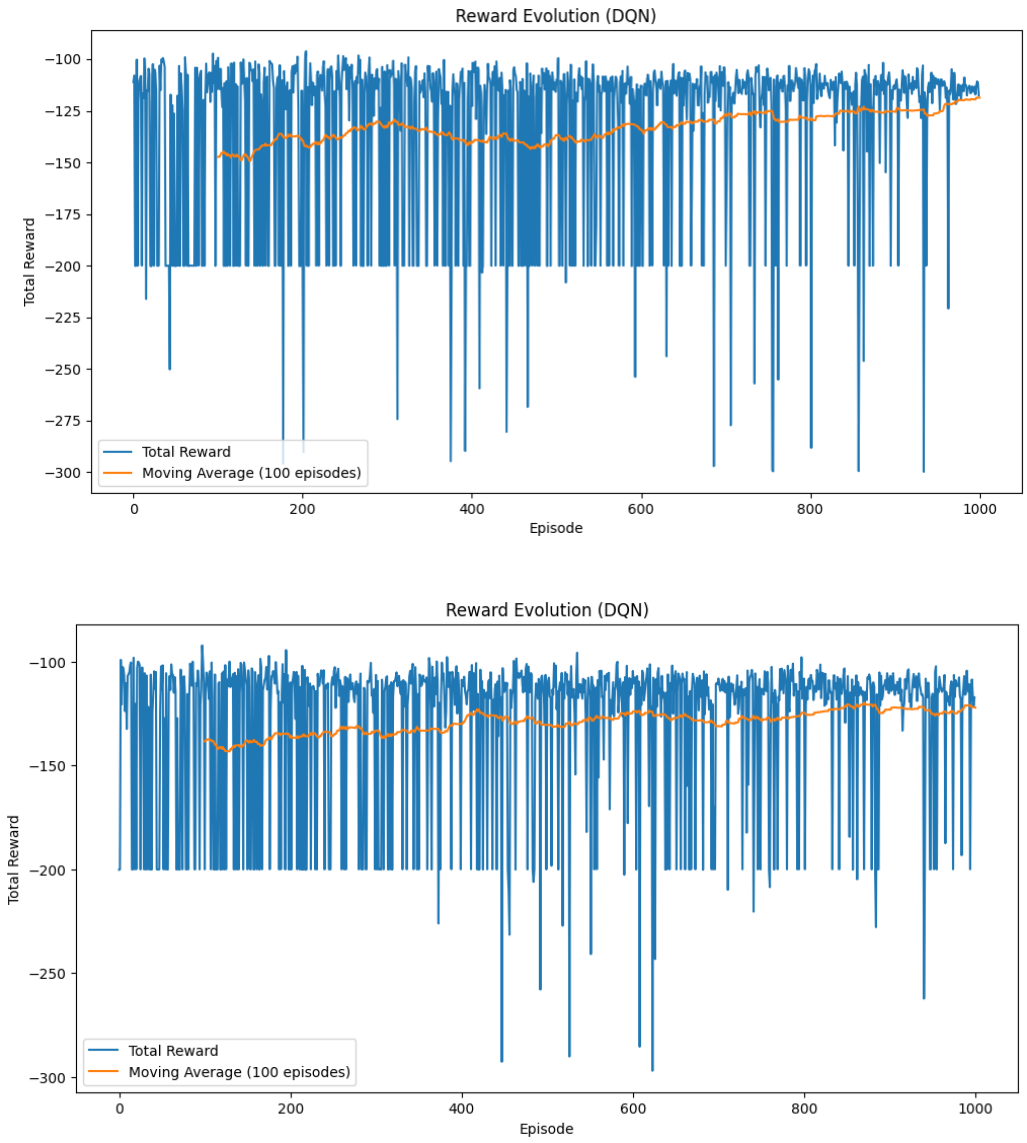


Figure 9: Comparison between Normalized (top) and Not Normalized (bottom)

In conclusion, state normalization enhanced the DQN's performance in the Bipedal Walker task, improving learning speed, stability, but not the overall reward.

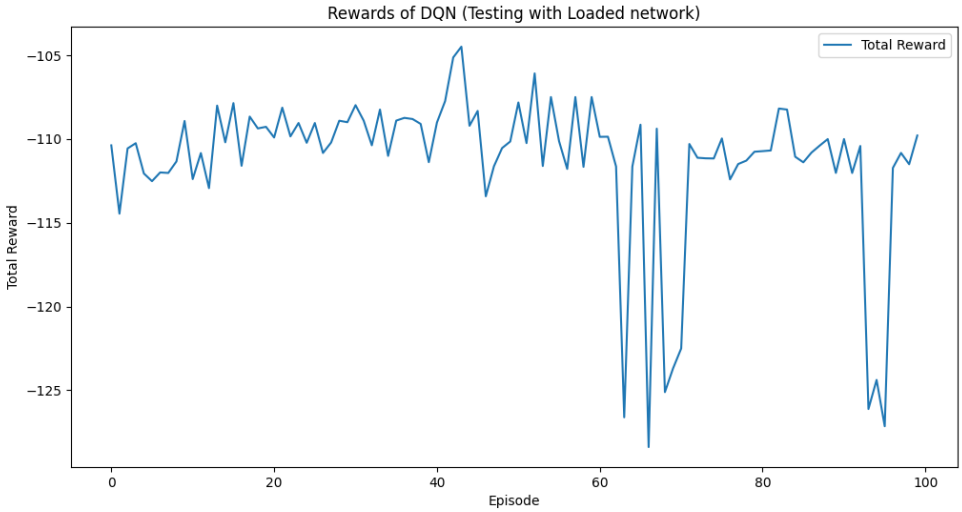


Figure 10: DQN testing rewards.

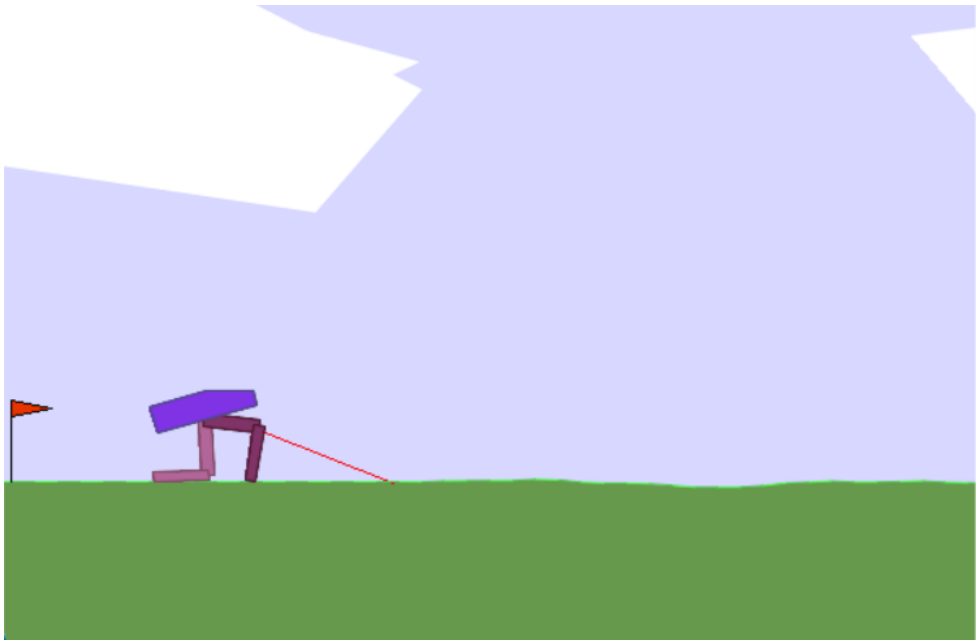

Figure 11: The bipedal walker stuck in the environment

### 3.3 Proximal Policy Optimization

For the Proximal Policy Optimization (PPO) model, the best-performing hyperparameters were identified as $lr = 0.001$, $updates\_per\_iter = 10$, and $\gamma = 0.99$, $ts\_per\_batch = 6000$, $max\_ts\_per\_ep = 2000$. Leveraging these settings, the PPO agent showcased remarkable proficiency in the Bipedal Walker environment, significantly surpassing the performance of other reinforcement learning algorithms tested.

The agent not only managed to maintain balance but also exhibited fluid and coordinated walking motions, frequently reaching the designated goal. This notable performance can be attributed to several critical factors:

1. **Stable Policy Updates**: The PPO algorithm's clipped surrogate objective played a crucial role in maintaining stability during policy updates. This mechanism effectively prevented large, destabilizing changes in the policy, thereby facilitating smoother and more consistent learning.

2. **Efficient Learning Dynamics**: The learning rate, $\alpha$, was meticulously tuned to ensure that the agent could make significant updates to its policy without succumbing to overfitting. This careful balance allowed for robust learning that was both efficient and resilient to noise.

3. **Effective Exploration Strategy**: PPO employs a well-balanced exploration strategy that allows the agent to effectively discover new actions while refining its policy based on previous experiences. This exploration is guided by a probabilistic approach that ensures the agent can explore diverse actions and scenarios, facilitating the discovery of optimal strategies and enhancing overall performance.

The evolution of the loss functions for both the critic and actor networks further demonstrates the agent's steady and continuous learning. Over successive iterations, the loss functions for the critic network, which evaluates the value function, decreased consistently, indicating that the network's estimates of state values were becoming more accurate. Concurrently, the actor network's loss function showed a downward trend, reflecting improved policy performance and more effective action selection. These trends confirm that the PPO algorithm effectively guided the agent toward learning better strategies, with both networks refining their performance over time. This continuous improvement underscores the stability and efficiency of PPO in facilitating progressive learning and adaptation.

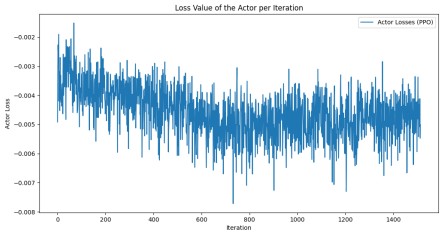


Figure 12: Evolution of the loss of the actor.

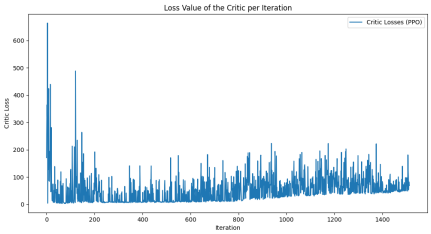


Figure 13: Evolution of the loss of the critic.

Figure 14: Evolution of the loss functions of both the actor and the critic network.

Quantitative analysis revealed that the PPO agent consistently achieved high rewards across numerous training episodes. The learning curves depicted a steep and sustained increase in performance, highlighting the agent's rapid acquisition of effective walking strategies. The agent's behavior was characterized by a smooth and stable gait, demonstrating its ability to adapt to the environment's dynamics and achieve the task objectives reliably.

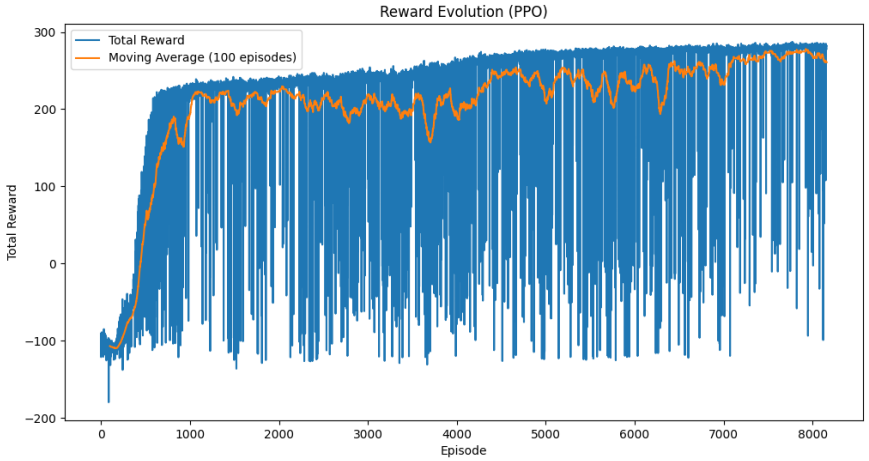


Figure 15: PPO learning rewards.

Furthermore, the PPO model's ability to generalize across different scenarios within the Bipedal Walker environment was particularly impressive. The agent not only adapted to varying terrains but also handled unexpected perturbations with remarkable agility and control. This adaptability underscores the robustness of the PPO algorithm when equipped with the optimal hyperparameters.

Overall, the Proximal Policy Optimization algorithm, fine-tuned with the specified hyperparameters, demonstrated exceptional performance in the Bipedal Walker task. The agent's rapid learning pace, stable policy updates, and consistent goal-reaching capability highlight the effectiveness of PPO for complex control tasks. This performance underscores PPO's strength in handling continuous action and state spaces, a significant advantage over traditional methods that require discretization. Unlike discretized approaches, PPO operates directly in the continuous space, avoiding the limitations and inefficiencies associated with discretization. By eliminating the need to discretize actions and states, PPO allows for more nuanced and precise control, enabling the agent to effectively manage continuous dynamics and adapt to varying conditions. The observed results strongly support the use of PPO as a viable and powerful method for training agents in environments that demand high levels of coordination and adaptability, particularly in scenarios where continuous decision-making and smooth policy transitions are crucial.

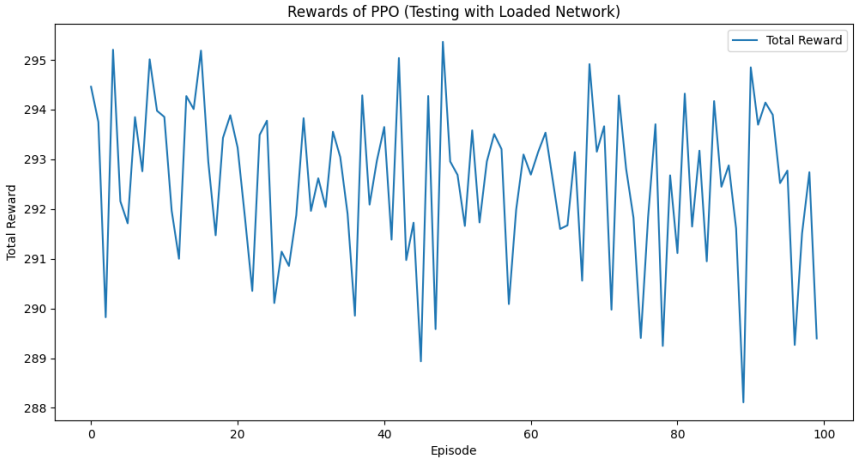


Figure 16: PPO testing rewards.

As shown in the figure above, the cumulative reward of our PPO-trained agent approaches 300, which is the score required to solve the environment. Other agents trained with PPO were also capable of reaching and exceeding a score of 300. However, these agents were more prone to falling, resulting in episodes with significantly lower rewards.

# 4 Final considerations

At the onset of this project, we anticipated that the Q-Table method would be unsuitable for this environment due to the challenges of discretization and high dimensionality, which could heavily impact RAM usage.

Initially, we were hopeful that DQN would be a breakthrough. However, it quickly became evident that our network struggled to effectively approximate the Q-function in both normalized and denormalized modes due to the high dimensionality of the discretized action space, which included $11^4$ possible actions.

Fortunately, Proximal Policy Optimization (PPO) addressed the issues associated with discretizing observation and action spaces, successfully solving the environment. PPO produced agents capable of consistently scoring 300 points with adequate training and demonstrating a full bipedal walking style.

Despite this success, high-scoring agents still exhibited instability in certain states, leading to occasional falls. Our proposed agent, however, showed greater resilience. By bending one leg backward, it slightly reduced its movement speed but significantly improved stability, enabling efficient and stable walking.

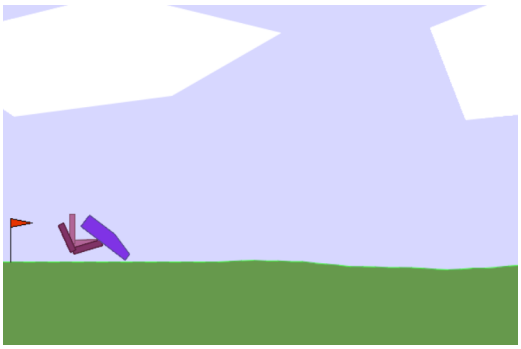

Figure 17: Agent using Q-Table.

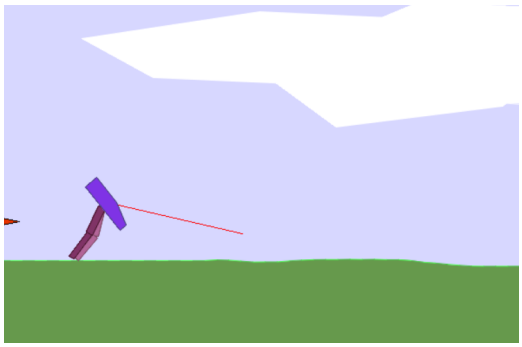

Figure 18: Agent using DQN.

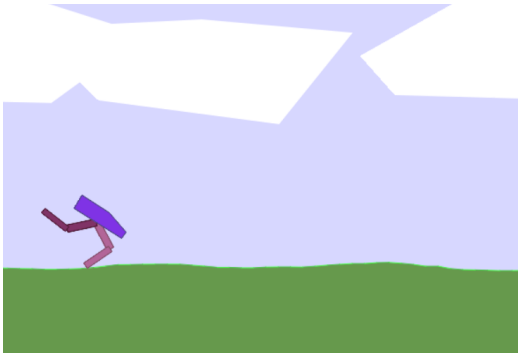

Figure 19: Agent solving the environment.

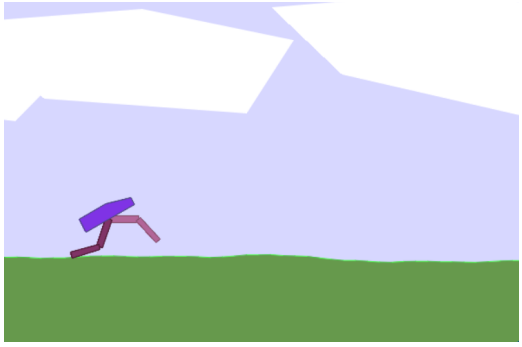

Figure 20: Our proposed agent.

Figure 21: Comparison of agents.

# References

[1] Eric Yang Yu, *Medium Article on PPO Implementation with PyTorch*, Medium, 2020.
`https://medium.com/analytics-vidhya/coding-ppo-from-scratch-with-pytorch-part-1-4-613dfc1b14c8`