Reinforcement Learning: CartPole, Breakout, and Pendulum

1 Introduction

This report presents the implementation and analysis of three gym environments, each solved using different reinforcement learning algorithms: Q-learning, Deep Q-Networks (DQN), and Proximal Policy Optimization (PPO). The first environment, Pendulum, was solved using Q-learning after adapting the environment to ensure compatibility with the algorithm. The second environment, chosen for its control challenge, is CartPole, which was addressed using both DQN and PPO with a custom Artificial Neural Network (ANN). Finally, I explored the Atari environment Breakout, employing both DQN and PPO, this time utilizing a custom Convolutional Neural Network (CNN) to effectively process image-based observations. The results highlight the effectiveness of these reinforcement learning techniques across different applications and demonstrate how they can be tailored to handle various settings. .

2 Pendulum Environment

The Pendulum environment consists of a pendulum that swings freely. The objective is to stabilize the pendulum in the upright position by applying a force at every step.

Observation Space

The state space is a three-dimensional continuous vector:

- **Cosine of the Angle** ($\cos(\theta)$): Represents the angle of the pendulum, bounded between $[-1, 1]$.
- **Sine of the Angle** ($\sin(\theta)$): Provides a complementary angle representation for calculating the direction. It is bounded between $[-1, 1]$.
- **Angular Velocity** ($\dot{\theta}$): The rate of change of the pendulum's angle, bounded within $[-8, 8]$ radians per second.

Action Space

The action space is continuous and consists of a single scalar action:

- **Torque** (u): A continuous value representing the force applied to the pendulum, bounded within the range $[-2, 2]$ Newton-meters.

Reward Structure

The reward in the Pendulum environment is defined as the negative cost of deviation from the upright position:

$$\text{Reward} = -(\theta^2 + 0.1 \cdot \dot{\theta}^2 + 0.001 \cdot u^2)$$

where θ represents the angle from the vertical, $\dot{\theta}$ is the angular velocity, and u is the applied torque. This reward structure penalizes both the angle's deviation from the upright position and the amount of torque applied, encouraging the agent to maintain stability with minimal movement. A reward closer to zero is achieved when the pendulum is near vertical with minimal angular velocity, indicating an optimally stable position.

End of the Simulation

The simulation in the Pendulum environment does not naturally terminate on its own, as the `done` flag is always set to `False`. However, the environment is configured to truncate each episode after a maximum of 200 steps. This truncation ensures that each episode has a finite length, allowing for periodic evaluation and consistent reward tracking across episodes.

3 Q-Learning Algorithm

3.1 Algorithm Description

Q-Learning is an off-policy Temporal Difference (TD) control algorithm that learns the optimal action-value function $Q(s, a)$ without requiring a model of the environment. In Q-Learning, the agent interacts with the environment by taking actions, observing rewards, and transitioning to new states. The Q-values are stored in a table and are updated based on the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

where:

- α is the learning rate, controlling the size of each update step,
- γ is the discount factor, indicating the importance of future rewards,
- r is the immediate reward after taking action a from state s,
- s' is the resulting state after taking action a, and
- a' represents possible actions in state s'.

The goal is to learn the optimal policy by selecting actions that maximize expected future rewards.

3.2 Pseudocode

Algorithm 1 Q-Learning Algorithm

```
1: Initialize Q-table $Q(s, a)$ arbitrarily
2: for episode in range(max_episodes) do
3:   Initialize state $s$
4:   for t in range(max_timesteps) do
5:     Choose action $a$ from state $s$ using an $\epsilon$-greedy policy based on $Q(s, a)$
6:     Take action $a$, observe reward $r$ and next state $s'$
7:     Update Q-value:
           $Q(s,a) \leftarrow Q(s,a) + \alpha \left( r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$
8:     Update state $s = s'$
9:     if done then
10:      Break
11:    end if
12:  end for
13: end for
```

4 Challenges in applying Q-Learning to the Pendulum Environment

Applying Q-learning to the Pendulum environment presents several challenges due to its continuous observation and action spaces, which make it incompatible with a standard Q-table.

4.1 Challenge 1: Continuous Action Space

In Q-learning, actions must be finite and discrete to map to specific Q-values in a Q-table. However, the continuous torque values in the Pendulum environment render the direct application of Q-learning impossible.

Solution: Discretization of Action Space To address this challenge, the continuous torque range was divided into a set of discrete values (specifically, I chose to divide the continuous space into 15 regions). This transformation allowed each action to map to a specific index in the Q-table, enabling the agent to make distinct torque choices.

4.2 Challenge 2: Continuous Observation Space

The observation space is formed by three continuous features. Again, I divided each of them into 15 regions. Thanks to these adjustments, the resulting Q matrix, while being four-dimensional, can still be effectively used for the Q-learning algorithms.

5 Implementation Approach

With both the action and observation spaces discretized, I implemented Q-learning as follows:

- **Q-table Initialization**: A 4-dimensional Q-table was created, where each entry corresponds to a unique state-action pair based on the discretized observations and actions.
- **Action Selection Using Epsilon-Greedy Strategy**: During training, actions were chosen using an epsilon-greedy policy. This strategy balances exploration (random actions) and exploitation (choosing the action with the highest Q-value).
- **Q-table Update Using Bellman Equation**: After each action, the Q-table was updated according to the Bellman equation, which adjusts the Q-value for the current state-action pair based on observed rewards and estimated future rewards.
- **Reward Tracking and Epsilon Decay**: I recorded the rewards per episode and decayed epsilon over time, allowing the agent to increasingly rely on learned Q-values rather than random exploration.
- **Periodic Saving of Q-table**: The Q-table was saved whenever the agent achieved a new best reward. This ensured training progress could be resumed from high-performance points.

Performance Comparison with a Random Agent

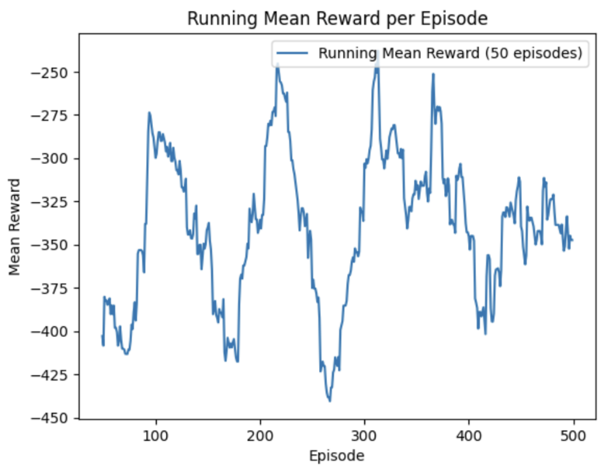


Figure 1: Q-learning agent

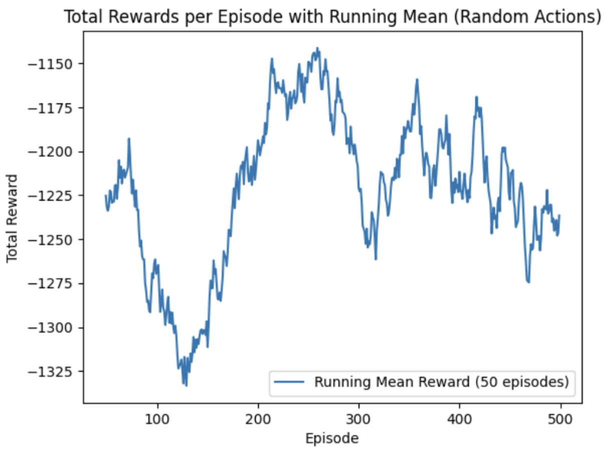


Figure 2: Random agent

To evaluate the effectiveness of the Q-learning algorithm in the Pendulum environment, I tested the trained Q-learning agent over 1000 episodes, comparing its performance with a random agent. The evaluation metric was the average reward per episode.

The results show a clear difference in performance:

- **Random Agent**: The average reward fluctuated between -1325 and -1150.
- **Q-learning Agent**: The trained Q-learning agent achieved a consistently higher average reward, ranging between -450 and -250.

These findings demonstrate that the Q-learning agent, trained through discretization of continuous spaces, significantly outperforms the random agent. The improvement in average reward indicates that the Q-learning agent has learned a policy that brings the pendulum closer to a stable, upright position more frequently and with better control.

This experiment illustrates that Q-learning, though originally designed for discrete environments, can be effectively applied to continuous environments like Pendulum by discretizing state and action spaces. This approach provides a simpler alternative to deep reinforcement learning methods, such as DQN, for managing continuous control tasks.

6 CartPole Environment

Scenario Description

The CartPole environment consists of a cart that can move horizontally along a track. Attached to the cart is a pole that pivots freely at one end. The goal is to balance the pole vertically by applying forces to the cart.

State Space

The state in CartPole is represented by a four-dimensional continuous vector:

- **Cart Position** (x): Horizontal position of the cart on the track, bounded within $[-4.8, 4.8]$ meters.
- **Cart Velocity** ($\dot{x}$): Velocity of the cart along the track, unbounded.
- **Pole Angle** (θ): Angle of the pole with respect to the vertical position, observed within $[-0.418, 0.418]$ radians.
- **Pole Angular Velocity** ($\dot{\theta}$): Rate of change of the pole angle, also unbounded.

Action Space

The action space is discrete, consisting of two possible actions:

- **Push Left (0)**: Apply a force to move the cart to the left.
- **Push Right (1)**: Apply a force to move the cart to the right.

Reward Structure

The agent receives a reward of $+1$ for every time step that the pole remains upright. The episode ends if:

- The pole angle exceeds a specified threshold (typically $\theta \geq 15^\circ$).
- The cart moves beyond the allowed track limits.
- The maximum episode length (usually 200 steps) is reached.

Ending Conditions

The episode ends if any one of the following occurs:

- **Termination**: Pole angle is greater than $\pm 12^\circ$ (± 0.2095 radians).
- **Termination**: Cart position is greater than ± 2.4 (the center of the cart reaches the edge of the display).
- **Truncation**: Episode length exceeds 500 steps (for the v1 version, that I used).

7 Deep Q-Learning (DQN)

7.1 Algorithm Description

DQN extends Q-learning with two stabilization techniques:

- **Experience Replay**: A replay buffer stores past experiences, and mini-batches are sampled randomly for training. This breaks correlations in the data, improving sample efficiency and stability.
- **Fixed Target Network**: A separate network Q' is periodically updated with the weights of the primary Q-network to calculate target Q-values. This technique reduces oscillations and stabilizes training.

The objective of DQN is to learn an optimal action-value function $Q(s, a; \theta)$, where θ represents the parameters of the neural network approximating the Q-values. Compared to traditional Q-learning, **Deep Q-Learning (DQN)** is better suited to environments with **continuous observation spaces** (though it still requires discrete action spaces). This capability arises from using a **deep neural network** to approximate Q-values, enabling DQN to handle high-dimensional state spaces effectively.

7.1.1 Pseudocode

Algorithm 2 Deep Q-Learning (DQN) Algorithm

```
 1: Initialize Q-network with random weights $\theta$
 2: Initialize target network $Q'$ with weights $\theta' = \theta$
 3: Initialize replay buffer $D$
 4: for episode in range(max_episodes) do
 5:   Initialize state $s$
 6:   for t in range(max_timesteps) do
 7:     Select action $a$ using $\epsilon$-greedy policy based on $Q(s, a; \theta)$
 8:     Take action $a$, observe reward $r$, next state $s'$, and done flag done
 9:     Store experience $(s, a, r, s', \text{done})$ in replay buffer $D$
10:     Sample random mini-batch of experiences from $D$
11:     Compute target Q-values for each experience:
          $Q_{\text{target}} = \begin{cases} r & \text{if done} \\ r + \gamma \max_{a'} Q'(s', a'; \theta') & \text{otherwise} \end{cases}$
12:     Perform a gradient descent step on $(Q(s, a; \theta) - Q_{\text{target}})^2$ with respect to $\theta$, using the mini-batch
13:     if t mod update_target_every == 0 then
14:       Update target network weights $\theta' \leftarrow \theta$
15:     end if
16:     Update state $s \leftarrow s'$
17:     if done then
18:       Break
19:     end if
20:   end for
21: end for
```

8 Proximal Policy Optimization (PPO)

8.1 Algorithm Description

Proximal Policy Optimization (PPO) is a policy gradient method that improves stability by using a clipped objective function to limit the policy updates. PPO aims to optimize the policy π_θ by maximizing an objective that includes a clipping mechanism, which prevents large deviations from the previous policy. The PPO objective is:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio of the new and old policies,
- ϵ is a hyperparameter that sets the clipping range, and
- $\hat{A}_t$ is the advantage estimate at timestep t.

PPO uses multiple epochs of mini-batch updates, making it both sample-efficient and stable.

8.1.1 Pseudocode

Algorithm 3 Proximal Policy Optimization (PPO) Algorithm

```
1:  Initialize policy parameters $\theta$ and value function parameters $\phi$
2:  for each training iteration do
3:    Collect a set of trajectories $\{\tau_i\}$ by executing policy $\pi_\theta$ in environment $E$
4:    for each trajectory $\tau_i$ do
5:      Compute the advantage estimates $A_t$ using the value function $V_\phi$
6:    end for
7:    for each epoch $k = 1, \ldots, K$ do
8:      for each minibatch of size $B$ from the collected trajectories do
9:        Compute the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$
10:       Compute the surrogate objective:
              $L(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$
11:       Update the policy parameters $\theta$ via gradient ascent on $L(\theta)$
12:       Update the value function parameters $\phi$ via gradient descent
13:     end for
14:   end for
15: end for
16: return Optimized policy network $\pi_\theta$ and value function $V_\phi$
```

9 Model Configurations for CartPole-v1 and comparisons

In this section, I compare the hyperparameters chosen for the Proximal Policy Optimization (PPO) and Deep Q-Network (DQN) models in the CartPole-v1 environment. Given my limited experience in deep learning, I opted for practical, well-documented settings for each algorithm. Additionally, I designed a simple custom neural network that performed effectively in both models. The following tables outline the settings for each model, along with my reasoning for specific choices.

PPO Hyperparameter	**Value**
Policy	`"MlpPolicy"`
Learning Rate	`lambda f: f * 0.001`
Batch Size	`256`
n_steps	`32`
n_epochs	`20`
Gamma	`0.98`
GAE Lambda	`0.8`
Clip Range	`lambda f: f * 0.2`
Entropy Coefficient	`0.0`
Policy Kwargs	CustomNetwork
Verbose	`1`

Table 1: PPO Hyperparameters

DQN Hyperparameter	**Value**
Policy	`"MlpPolicy"`
Learning Rate	`2.3e-3`
Batch Size	`64`
Buffer Size	`100,000`
Gamma	`0.99`
Target Update Interval	`10`
Train Frequency	`256`
Gradient Steps	`128`
Exploration Fraction	`0.16`
Exploration Final Epsilon	`0.04`
Policy Kwargs	CustomNetwork
Verbose	`1`

Table 2: DQN Hyperparameters

9.1 Explanation and Comparison of Key Hyperparameters

- **Learning Rate**: I used a decaying learning rate schedule in PPO (`lambda f: f * 0.001`) to avoid instability as the model converges. This helps to control update sizes, especially beneficial in policy-based methods like PPO. For DQN, I chose a fixed learning rate of `2.3e-3` to maintain a consistent update rate, as Q-learning benefits from a stable learning environment.

- **Batch and Buffer Size**: PPO's larger batch size (`256`) allowed each policy update to be well-informed by recent data, while I opted for no replay buffer to keep updates fresh and on-policy. For DQN, however, I set a smaller batch size (`64`) with a large replay buffer (`100,000`), enabling the model to sample diverse experiences and learn more effectively from past interactions.

- **Exploration Strategy**: DQN requires explicit exploration to balance learning and exploration; thus, I set the `exploration_fraction` and `exploration_final_eps` parameters to decay over time. This encourages broad exploration early on, helping the model avoid local optima, while gradually reducing randomness as it becomes more confident. PPO, in contrast, explores by its stochastic policy nature, so I didn't need to define exploration explicitly.
- **Gamma (Discount Factor)**: I used `0.98` for PPO, giving slightly more weight to immediate rewards, as this helps the model focus on short-term gains while still considering long-term stability. For DQN, I chose a slightly higher discount factor of `0.99` to prioritize future rewards, which is typical in value-based approaches that aim to maximize cumulative rewards.
- **Target Update Interval and Gradient Steps (DQN Only)**: DQN relies on stability enhancements like the target network and gradient updates. I set `target_update_interval` to update the target network every 10 steps, and `gradient_steps` to 128 for more in-depth training per update. This setup, combined with the replay buffer, helped DQN efficiently learn stable Q-values over time.
- **Clip Range and GAE Lambda (PPO Only)**: PPO includes `clip_range` and `gae_lambda` to manage policy stability and reduce the variance of advantage estimation, essential for on-policy methods. I set `clip_range` to start at 0.2 to avoid overly large policy updates, and `gae_lambda` to 0.8, balancing bias and variance for smooth policy learning.

9.2 Custom Network Architectures

Both models use custom architectures. Despite my limited experience in deep learning, this structure provided enough capacity to handle CartPole's dynamics effectively.

- **Hidden Layers**: Two fully connected layers of 256 units each.
- **Activation Function**: ReLU, adding non-linearity between layers.
- **Sequential Model**: The layers are combined into a sequential structure for straightforward processing.

Its relatively low complexity allowed for efficient training while adequately handling the environment's state-space dimensions.

9.3 Model Training and Performance evaluation

The model was trained for 100000 timesteps for PPO and 50000 for DQN (as in the previous one I only saw a marginal improvement after such number of steps) and with logs generated every 4 steps (`log_interval=4`) to track progress. I then run a simulation testing the final models and compared them with a random agent, to see wether the training was worth or not.

The performance results for the PPO, DQN, and Random agents on the CartPole-v1 environment are summarized as follows:

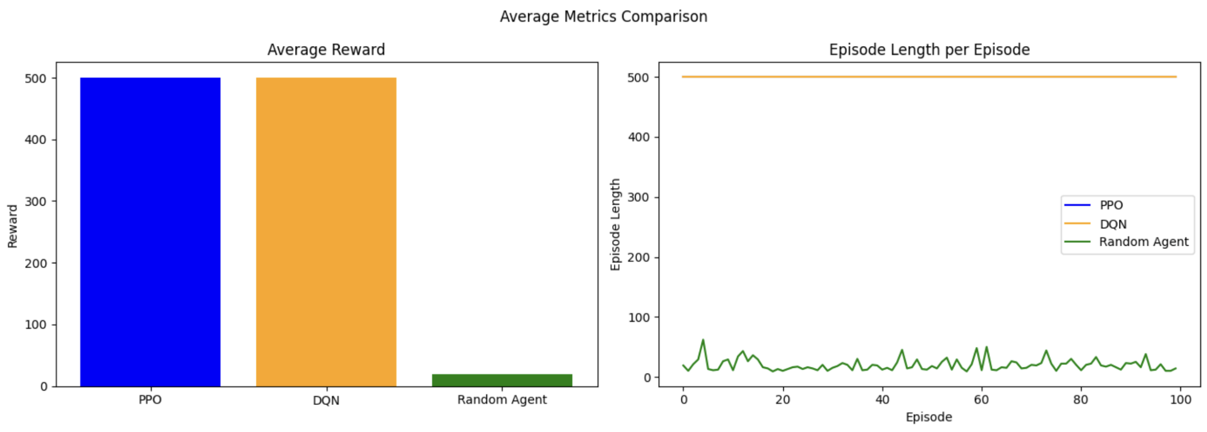
- **PPO:** Average Reward = 500.00, Average Episode Length = 500.00
- **DQN:** Average Reward = 500.00, Average Episode Length = 500.00
- **Random Agent:** Average Reward = 22.79, Average Episode Length = 22.79

9.4 Analysis

In this specific scenario, balancing a pole necessitates precise actions at every moment, leading to a stark contrast between the performance of trained models and that of a random agent. Both the PPO and DQN models achieved the maximum average reward of 500.00, consistently maintaining balance for the maximum episode length. This impressive performance illustrates the capability of both algorithms to learn and optimize policies in environments characterized by simpler state and action spaces like CartPole.

In stark contrast, the Random Agent exhibited significantly lower performance, averaging a reward of just 19.24, with an equivalent episode length of 19.24 steps. This outcome is not surprising, as the

Random Agent lacks any learning mechanism and relies solely on randomly chosen actions, resulting in inadequate control over the pole.



These results highlight the advantage of trained reinforcement learning agents over non-learning baselines, as well as the suitability of both PPO and DQN for tasks requiring stable control.

10 Breakout Environment

Scenario Description

The Breakout environment simulates the classic arcade game. The agent controls a paddle at the bottom of the screen and must hit a ball to break blocks positioned above. The objective is to break as many blocks as possible by bouncing the ball with the paddle.

State Space

The state in Breakout is represented by an 84×84 pixel grayscale image of the game screen, which captures:

- The position of the paddle.
- The position of the ball.
- The layout of the remaining blocks.

The state space is thus a high-dimensional space representing the visual information.

Action Space

The action space is discrete with four possible actions:

- **Noop (0)**: Do nothing.
- **Fire (1)**: Start the game or serve the ball if it is lost.
- **Move Right (2)**: Move the paddle to the right.
- **Move Left (3)**: Move the paddle to the left.

Reward Structure

The agent receives rewards based on the following events:

- **Block Destruction**: A positive reward is given when a block is destroyed.Its magnitude depends on the color of the block
- **Ball Loss**: No reward is given, but the episode may end if the ball is lost and no remaining lives are left.

Ending Conditioins

- **Game Over:** The episode ends when the player loses all lives, which occurs when the ball falls below the paddle.
- **Level Completion:** The episode may end if all bricks are destroyed, depending on the implementation.
- **Maximum Episode Length:** The episode terminates after reaching a predefined maximum of 1000 time steps.

10.1 Custom CNN Feature Extractor for Atari Breakout

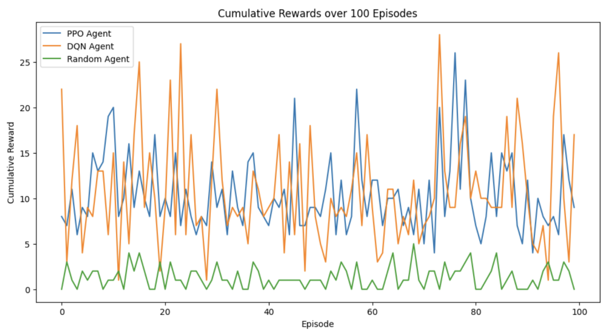
The main challenge in Atari environments is that the observation consists solely in the images displayed by the game. Therefore, it is essential to employ a Convolutional Neural Network (CNN) as a feature extractor to process the frames captured during gameplay. This approach allows for the extraction of meaningful information, which aids in the agent's learning process. I will now describe the custom CNN I implemented to solve this environment.

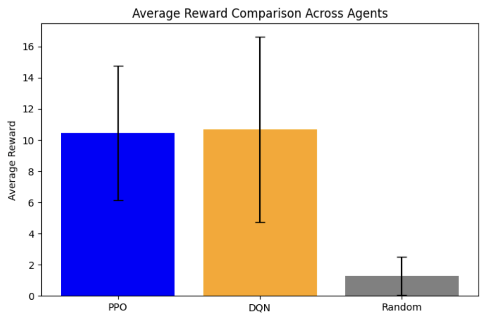
- **Convolutional Layers**: The CNN starts with three convolutional layers that progressively capture features from game frames:
  - Layer 1: 32 filters, 8x8 kernel, stride 4.
  - Layer 2: 64 filters, 4x4 kernel, stride 2.
  - Layer 3: 64 filters, 3x3 kernel, stride 1.

  Each convolutional layer is followed by a ReLU activation for non-linearity.
- **Fully Connected Layer**: What reuslts from the convolutions is flattened and passed through a fully connected layer (with 512 units) with ReLU, enhancing complex representations.
- **Output Dimension Calculation**:The CNN's output size is computed using a sample observation, ensuring compatibility with the fully connected layer.

This CNN structure effectively captures spatial and dynamic information, supporting informed decision-making for the agent.

10.2 Performance comparison and analysis





I trained both PPO and DQN with 100000 steps. Given the complexity of the scenario, it might have more of a longer training, but because of time constraints I did not manage to do it. The results are still pretty good and from the videos it is possible to see that the actions of the agents make sense. Let's now compaer them with a random agent

10.2.1 Random Agent

- **Rewards**: The Random agent's rewards are generally low, typically ranging between 0 and 5 per episode, with occasional higher values.
- **Interpretation**: The low rewards and high variance reflect the Random agent's lack of understanding of the environment, as expected due to its random action selection. The agent lacks any strategy to maintain the game state or consistently score points.

10.2.2 DQN Agent

- **Rewards**: The DQN agent performs significantly better than the Random agent, with rewards generally higher and often averaging between 10 and 15, with some episodes reaching 20 or more.
- **Interpretation**: The higher rewards indicate that the DQN agent has learned a somewhat effective policy for the environment. It shows an ability to score higher by learning strategies that improve its survival and scoring potential within the game, though its performance varies across episodes.

10.2.3 PPO Agent

- **Rewards**: The PPO agent's rewards are consistently moderate, typically between 7 and 15 per episode. Its performance is relatively stable, with a narrower range of values compared to the DQN agent.
- **Interpretation**: This indicates that the PPO agent has learned a stable but somewhat conservative policy for the Breakout environment. The narrower reward range suggests that the agent maintains consistent strategies that are effective but not necessarily as exploratory or reward-maximizing as DQN.

In summary, both DQN and PPO outperform the Random agent, with DQN achieving the highest peak rewards, albeit with more variance, while the PPO agent is more consistent but with slightly lower peak performance.

11 Conclusion

Through the implementation and analysis of Q-learning, DQN, and PPO across three distinct reinforcement learning environments—Pendulum, CartPole, and Breakout—I've gained significant insights into both the potential and challenges of each algorithm within specific contexts. My approach to reinforcement learning involved adapting algorithms to different environments, and the experiments highlighted each model's strengths, weaknesses, and adaptability.

- **Pendulum Environment**: Q-learning, a tabular reinforcement learning approach, was successfully applied to the Pendulum environment by discretizing its continuous state and action spaces. This adaptation allowed for a structured exploration and exploitation, resulting in an effective policy for maintaining the pendulum's upright position. Despite Q-learning's typical limitation to discrete spaces, this experiment demonstrated that a thoughtful discretization strategy can make it a viable solution for simple continuous environments, providing a lightweight alternative to more computationally expensive deep learning methods.
- **CartPole Environment**: Both DQN and PPO excelled in the CartPole environment, achieving near-optimal results with straightforward hyperparameters and a simple custom ANN. These results reinforce the suitability of DQN and PPO for environments with simple state-action spaces, where rapid and reliable decision-making is crucial. By comparing a random agent with DQN and PPO, I observed that model-based policies outperform purely random strategies in tasks requiring continuous stability. Furthermore, CartPole demonstrated the stability benefits of PPO's clipped objective and DQN's fixed target network in efficiently learning a control policy, with both models achieving comparable, high-level performance.
- **Breakout Environment**: Applying DQN and PPO in the Breakout environment underscored the challenges of high-dimensional, image-based observation spaces. By implementing a custom CNN to process visual data, I enabled both models to develop a basic level of strategic play, though performance was moderate given the complexity of the task. DQN's higher variance and occasional high rewards suggest it learned more aggressive strategies but lacked consistency. In contrast, PPO provided more stable, conservative performance. The experiment confirmed the need for extended training and complex network architectures in image-based reinforcement learning tasks, especially for dynamic, multi-faceted environments like Breakout.

In conclusion, these experiments underscore the adaptability of reinforcement learning algorithms across a range of tasks, from discrete and continuous control to high-dimensional observation environments. Each algorithm has specific advantages that make it suitable for particular types of reinforcement learning problems, emphasizing the importance of tailoring approaches based on the unique characteristics of each environment.