43008: Reinforcement Learning

Assignment -2

Student Name: Carlos Emiliano Mendoza Hernandez

Student ID: 26130551

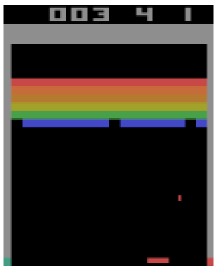
Introduction:

This report presents a comparative analysis of several Reinforcement Learning (RL) algorithms across a spectrum of control tasks, designed to evaluate their performance, efficiency, and implementation challenges. The scenarios were selected to represent distinct categories of RL problems: the foundational Pendulum classic control environment, which features a continuous state and action space; the vector-based CartPole environment, requiring a stable balancing policy; and the vision-based Atari game, Breakout, where the agent must learn to act directly from raw pixel data.

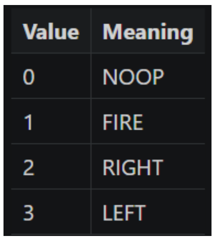
To solve these tasks, a range of algorithms was implemented. For the Pendulum task, a custom tabular Q-learning algorithm was developed, necessitating the discretization of the continuous environment. For the more complex CartPole and Breakout environments, industry-standard deep RL algorithms from the Stable Baselines3 library were employed: Deep Q-Learning (DQN), a prominent value-based method, and Proximal Policy Optimization (PPO), a state-of-the-art actor-critic method. The primary objective of these experiments is to investigate the performance trade-offs and sample efficiency of each algorithm when applied to an environment for which it is either well-suited or fundamentally limited. Ultimately, this report provides a practical analysis of how an environment's characteristics—from simple vectors to complex visual scenes—dictate the selection and success of different Reinforcement Learning strategies.

Scenario/Env. and Algorithm:

1a. Breakout



Breakout is a classic Atari 2600 game in the Arcade Learning Environment (ALE). An agent controls a horizontal paddle at the bottom of the screen and hits a bouncing ball to break a wall of bricks at the top. The episode proceeds until the agent loses all lives (or an environment-defined timeout). Scoring is by destroying bricks; different brick colours can award different point values.



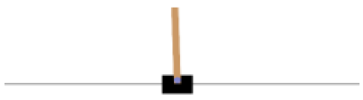
Value	Meaning
0	NOOP
1	FIRE
2	RIGHT
3	LEFT

State Space: The default state observation space *observation_space = Box(0,255, (210, 160, 3), np.uint)*, which are RGB frames with shape *(210,160,3)*.

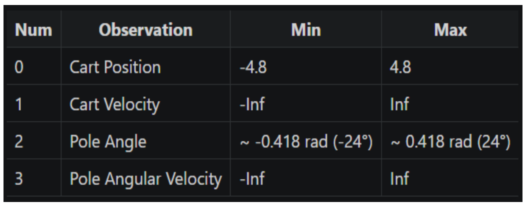
Action Space: Breakout has the action space of *Discrete(4)* with the table listing the meaning of each action's meaning.

Reward Structure: Game rewards are given for events like destroying bricks (and possibly other events). In ALE/Gym the reward for destroying a brick depends on the brick's colour (i.e. bricks are worth different integer scores).

1b. Cart Pole



The Cart-Pole environment is a classic control problem that serves as a benchmark in reinforcement learning. The scenario consists of a cart that can move horizontally along a frictionless track, with a pole hinged on its top surface. The system is unstable, and the pole will fall over if no action is taken. The goal for the agent is to learn a control strategy to balance the pole upright for as long as possible. The agent can apply a horizontal force to the cart, pushing it either to the left or to the right at each discrete time step.



Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

State Space: The observation is a *ndarray* with shape *(4,)* with values corresponding to the following positions and velocities:

Action Space: The action is a *ndarray* with shape *(1,)* which can take values *{0,1}* indicating the direction of the fixed force the car is pushed with.

- 0: Push cart to the left
- 1: Push cart to the right

Reward Structure: The environment uses a simple and dense reward signal to encourage the agent to prolong the episode. For every time step that the episode is not terminated, the agent receives a reward of +1. The objective is to maximize the cumulative reward, which is equivalent to maximizing the number of time steps the pole remains balanced.

Selected Algorithms

For this assignment, Deep Q-Learning (DQN) and Proximal Policy Optimization (PPO) have been selected. These algorithms represent two distinct and powerful families of reinforcement learning methods: value-based and policy-based, respectively.

For the Cart-Pole environment, as the state space is continuous, a simple Q-table is insufficient, necessitating the use of deep neural networks as function approximators. The appropriate feature extractor for both algorithms would be a simple Artificial Neural Network (ANN), also known as a Multi-Layer Perceptron (MLP), as the input is a low-dimensional vector.

In the case of Breakout, a Convolutional Neural Network (CNN) is the necessary feature extractor because the state is represented by the raw pixel data of the game screen. A CNN is designed to process such high-dimensional image data, automatically learning to identify crucial spatial features—such as the position of the ball, the paddle, and the bricks—which are essential for determining the optimal action.

A – Deep Q-Learning / DQN

DQN is a value-based, off-policy algorithm that revolutionized the field of deep reinforcement learning. It aims to learn the optimal action-value function, $Q^*(s, a)$, which represents the maximum expected future reward achievable from being in state s and taking action a.

DQN uses a deep neural network to approximate the Q-function. To stabilize training, it introduces two key mechanisms:

1. **Experience Replay:** The agent stores its experiences (state, action, reward, next state) in a replay buffer. During training, it samples mini-batches from this buffer, breaking the temporal correlation between consecutive samples and improving data efficiency.
2. **Target Network:** A separate, periodically updated "target network" is used to generate the target Q-values for the Bellman equation. This decouples the target from the online network, preventing oscillations and making training more stable.

DQN is highly effective in environments with discrete action spaces and can handle complex, high-dimensional state spaces. The Cart-Pole environment, with its discrete actions (push left/right) and continuous state vector, is a perfect use case for DQN. It serves as an excellent baseline for a value-based learning approach.

In the case of Breakout, it is a canonical benchmark for DQN: discrete action space + pixel inputs + sparse-ish scoring events (bricks broken). DQN is directly appropriate because it's designed to learn Q-values from raw images and historically obtained strong Atari scores.

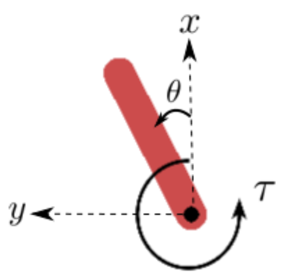
B – PPO

PPO is a policy-based, on-policy algorithm from the actor-critic family. It aims to directly learn a policy, $\pi(a|s)$, which maps states to a probability distribution over actions. It is known for its reliability, ease of implementation, and excellent performance across a wide range of tasks.

PPO improves upon older policy gradient methods by ensuring that policy updates are not too large, which could lead to performance collapse. It achieves this by using a clipped surrogate objective function. This objective function penalizes large changes to the policy, effectively keeping the new policy within a "trust region" of the old one, leading to more stable and reliable training. It uses two neural networks: one for the policy (the actor) and one for the value function (the critic), which estimates the value of a state.

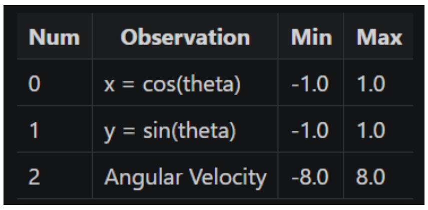
PPO is a state-of-the-art algorithm that is highly versatile, performing well in environments with either discrete or continuous action spaces. Selecting PPO provides a powerful point of comparison against DQN. As a policy-gradient method, it takes a different philosophical approach—directly optimizing the policy rather than deriving it from a value function.

2. Pendulum



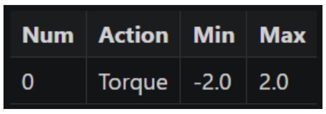
The task is the classic inverted-pendulum swing-up: apply a continuous torque at the pendulum's joint so the pendulum is swung upright and kept balanced (centre of gravity above the pivot). The pendulum starts in a random angle and small random angular velocity. The agent interacts with the environment in discrete time steps, choosing an action (torque) at each step, receiving the resulting reward, and transitioning to the next state.

State space: The observation is a *ndarray* with shape *(3,)*, representing the x-y coordinates of the pendulum's free end and its angular velocity.



Num	Observation	Min	Max
0	x = cos(theta)	-1.0	1.0
1	y = sin(theta)	-1.0	1.0
2	Angular Velocity	-8.0	8.0

Action space: The action is a *ndarray* with shape *(1,)* representing the torque applied to free end of the pendulum



Num	Action	Min	Max
0	Torque	-2.0	2.0

Reward structure: The reward function is defined as:

$R = -(theta^2 + 0.1 * theta\_dt^2 + 0.001 * torque^2)$

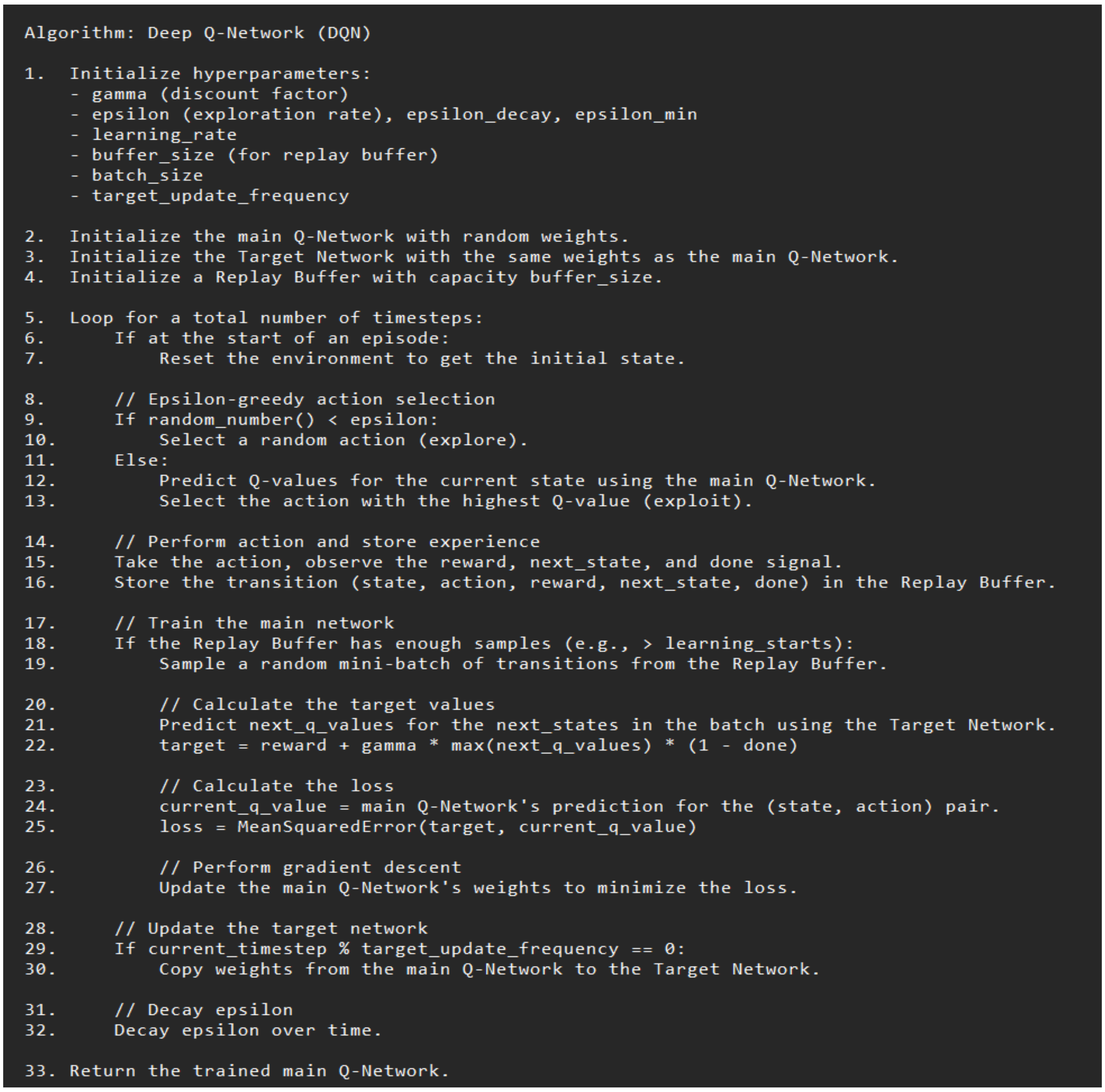
Where *theta* is the pendulum's angle normalized between *[-pi, pi]* (with 0 being in the upright position). Based on the above equation, the minimum reward that can be obtained is $-(pi^2 + 0.1 * 8^2 + 0.001 * 2^2) = -16.2736044$, while the maximum reward is zero (pendulum is upright with zero velocity and no torque applied).

Selected Algorithm: Q-learning was chosen because it is a foundational TD algorithm that clearly demonstrates the core principles of reinforcement learning: bootstrapping, exploration–exploitation, and iterative value estimation. While Pendulum has continuous dynamics, discretising the state and action spaces allows Q-learning to be applied in a straightforward manner, making it suitable for experimentation.

Q-learning is a model-free Temporal Difference (TD) control algorithm that estimates the action-value function $Q(s, a)$, which represents the expected return starting from state s, taking action a, and thereafter following the optimal policy. It is an off-policy method, meaning it learns the optimal value function independently of the agent's behaviour policy.

Diagram, Pseudocode and Flowchart:

DQN:



```
Algorithm: Deep Q-Network (DQN)

1.  Initialize hyperparameters:
    - gamma (discount factor)
    - epsilon (exploration rate), epsilon_decay, epsilon_min
    - learning_rate
    - buffer_size (for replay buffer)
    - batch_size
    - target_update_frequency

2.  Initialize the main Q-Network with random weights.
3.  Initialize the Target Network with the same weights as the main Q-Network.
4.  Initialize a Replay Buffer with capacity buffer_size.

5.  Loop for a total number of timesteps:
6.      If at the start of an episode:
7.          Reset the environment to get the initial state.

8.      // Epsilon-greedy action selection
9.      If random_number() < epsilon:
10.         Select a random action (explore).
11.     Else:
12.         Predict Q-values for the current state using the main Q-Network.
13.         Select the action with the highest Q-value (exploit).

14.     // Perform action and store experience
15.     Take the action, observe the reward, next_state, and done signal.
16.     Store the transition (state, action, reward, next_state, done) in the Replay Buffer.

17.     // Train the main network
18.     If the Replay Buffer has enough samples (e.g., > learning_starts):
19.         Sample a random mini-batch of transitions from the Replay Buffer.

20.         // Calculate the target values
21.         Predict next_q_values for the next_states in the batch using the Target Network.
22.         target = reward + gamma * max(next_q_values) * (1 - done)

23.         // Calculate the loss
24.         current_q_value = main Q-Network's prediction for the (state, action) pair.
25.         loss = MeanSquaredError(target, current_q_value)

26.         // Perform gradient descent
27.         Update the main Q-Network's weights to minimize the loss.

28.     // Update the target network
29.     If current_timestep % target_update_frequency == 0:
30.         Copy weights from the main Q-Network to the Target Network.

31.     // Decay epsilon
32.     Decay epsilon over time.

33. Return the trained main Q-Network.
```

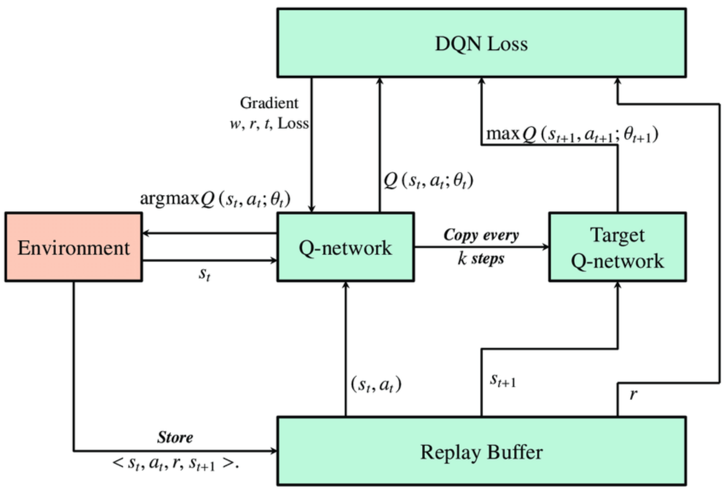
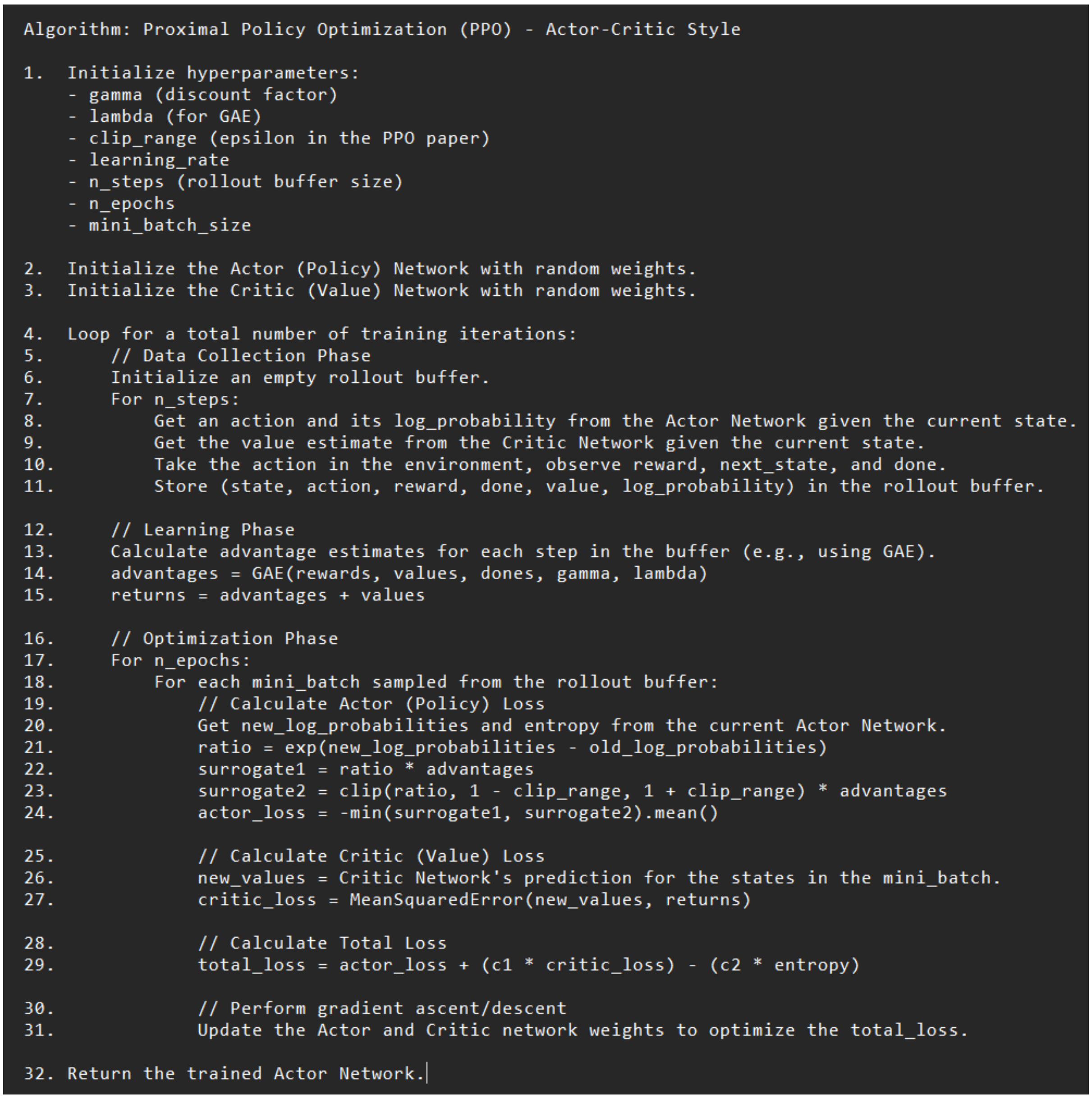


Figure 1. Source: https://www.researchgate.net/figure/A-Concept-of-DQN-Algorithm7_fig4_348444080

PPO:



```
Algorithm: Proximal Policy Optimization (PPO) - Actor-Critic Style

1.  Initialize hyperparameters:
    - gamma (discount factor)
    - lambda (for GAE)
    - clip_range (epsilon in the PPO paper)
    - learning_rate
    - n_steps (rollout buffer size)
    - n_epochs
    - mini_batch_size

2.  Initialize the Actor (Policy) Network with random weights.
3.  Initialize the Critic (Value) Network with random weights.

4.  Loop for a total number of training iterations:
5.      // Data Collection Phase
6.      Initialize an empty rollout buffer.
7.      For n_steps:
8.          Get an action and its log_probability from the Actor Network given the current state.
9.          Get the value estimate from the Critic Network given the current state.
10.         Take the action in the environment, observe reward, next_state, and done.
11.         Store (state, action, reward, done, value, log_probability) in the rollout buffer.

12.     // Learning Phase
13.     Calculate advantage estimates for each step in the buffer (e.g., using GAE).
14.     advantages = GAE(rewards, values, dones, gamma, lambda)
15.     returns = advantages + values

16.     // Optimization Phase
17.     For n_epochs:
18.         For each mini_batch sampled from the rollout buffer:
19.             // Calculate Actor (Policy) Loss
20.             Get new_log_probabilities and entropy from the current Actor Network.
21.             ratio = exp(new_log_probabilities - old_log_probabilities)
22.             surrogate1 = ratio * advantages
23.             surrogate2 = clip(ratio, 1 - clip_range, 1 + clip_range) * advantages
24.             actor_loss = -min(surrogate1, surrogate2).mean()

25.             // Calculate Critic (Value) Loss
26.             new_values = Critic Network's prediction for the states in the mini_batch.
27.             critic_loss = MeanSquaredError(new_values, returns)

28.             // Calculate Total Loss
29.             total_loss = actor_loss + (c1 * critic_loss) - (c2 * entropy)

30.             // Perform gradient ascent/descent
31.             Update the Actor and Critic network weights to optimize the total_loss.

32. Return the trained Actor Network.
```

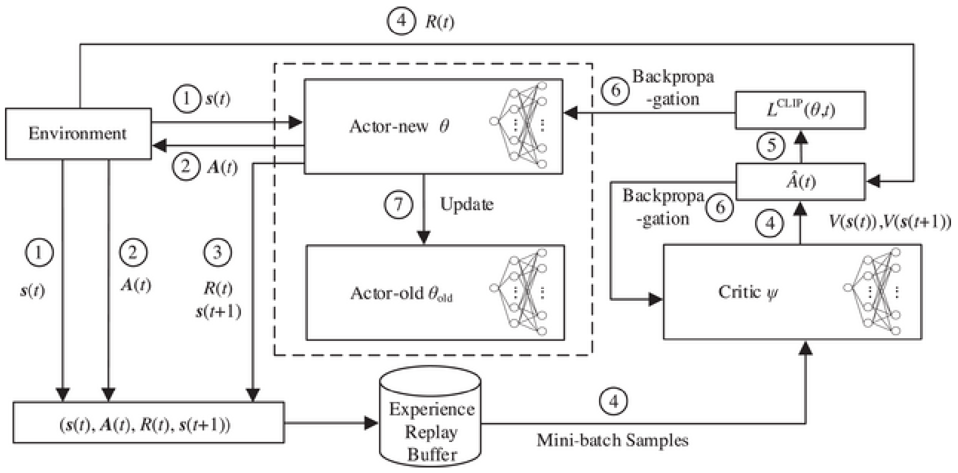
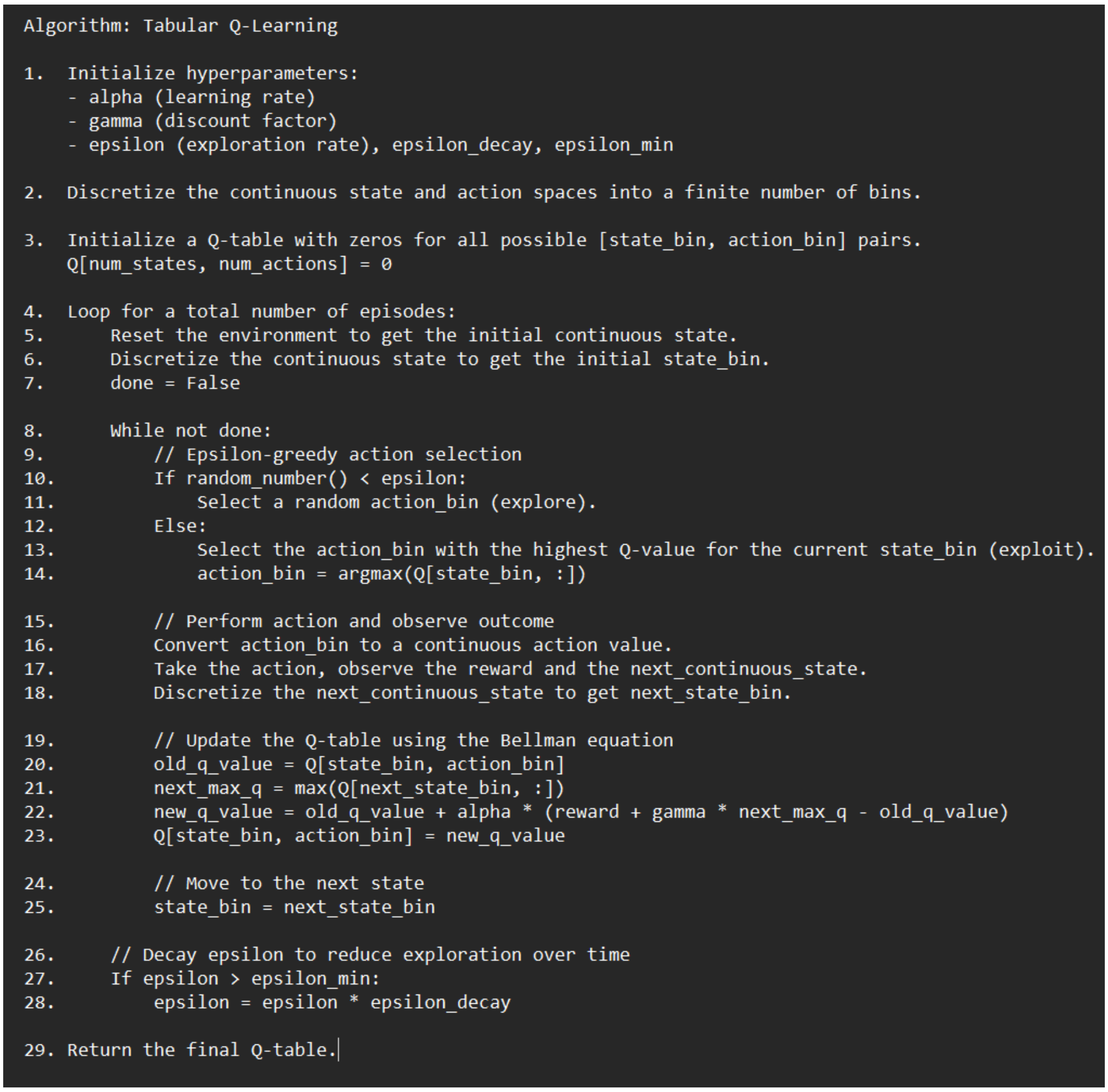


Figure 2. Source: https://www.researchgate.net/figure/Schematic-of-proximal-policy-optimization-PPO_fig2_376811584

Tabular Q-learning:



```
Algorithm: Tabular Q-Learning

1.  Initialize hyperparameters:
    - alpha (learning rate)
    - gamma (discount factor)
    - epsilon (exploration rate), epsilon_decay, epsilon_min

2.  Discretize the continuous state and action spaces into a finite number of bins.

3.  Initialize a Q-table with zeros for all possible [state_bin, action_bin] pairs.
    Q[num_states, num_actions] = 0

4.  Loop for a total number of episodes:
5.      Reset the environment to get the initial continuous state.
6.      Discretize the continuous state to get the initial state_bin.
7.      done = False

8.      While not done:
9.          // Epsilon-greedy action selection
10.         If random_number() < epsilon:
11.             Select a random action_bin (explore).
12.         Else:
13.             Select the action_bin with the highest Q-value for the current state_bin (exploit).
14.             action_bin = argmax(Q[state_bin, :])

15.         // Perform action and observe outcome
16.         Convert action_bin to a continuous action value.
17.         Take the action, observe the reward and the next_continuous_state.
18.         Discretize the next_continuous_state to get next_state_bin.

19.         // Update the Q-table using the Bellman equation
20.         old_q_value = Q[state_bin, action_bin]
21.         next_max_q = max(Q[next_state_bin, :])
22.         new_q_value = old_q_value + alpha * (reward + gamma * next_max_q - old_q_value)
23.         Q[state_bin, action_bin] = new_q_value

24.         // Move to the next state
25.         state_bin = next_state_bin

26.     // Decay epsilon to reduce exploration over time
27.     If epsilon > epsilon_min:
28.         epsilon = epsilon * epsilon_decay

29. Return the final Q-table.
```

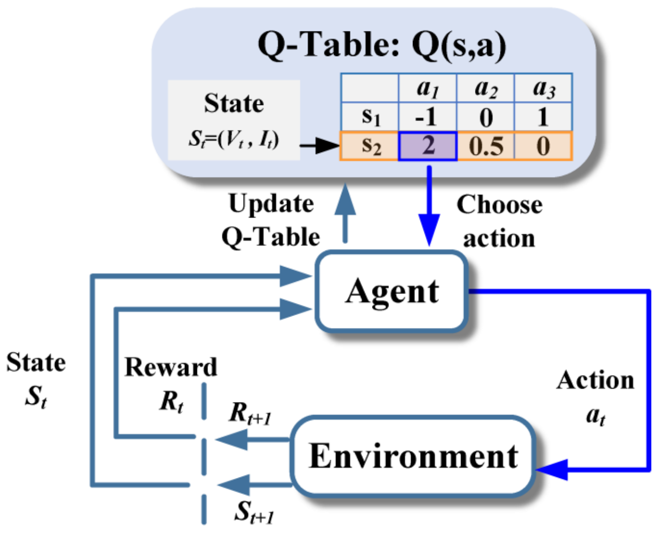
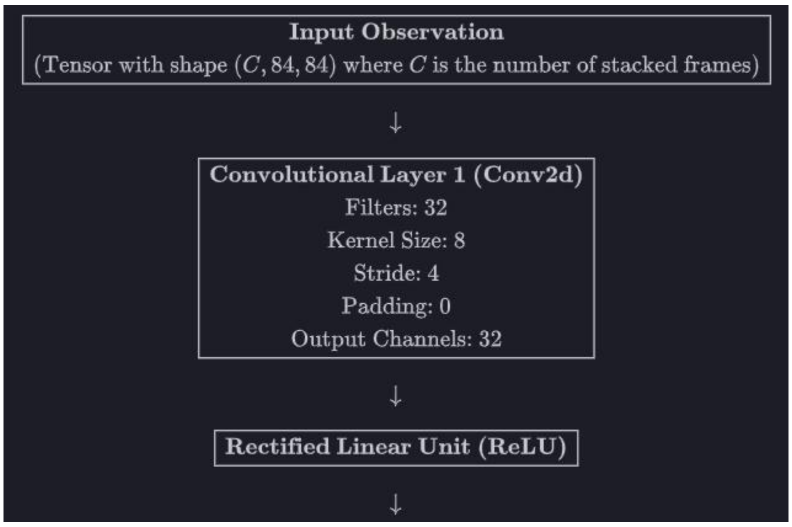
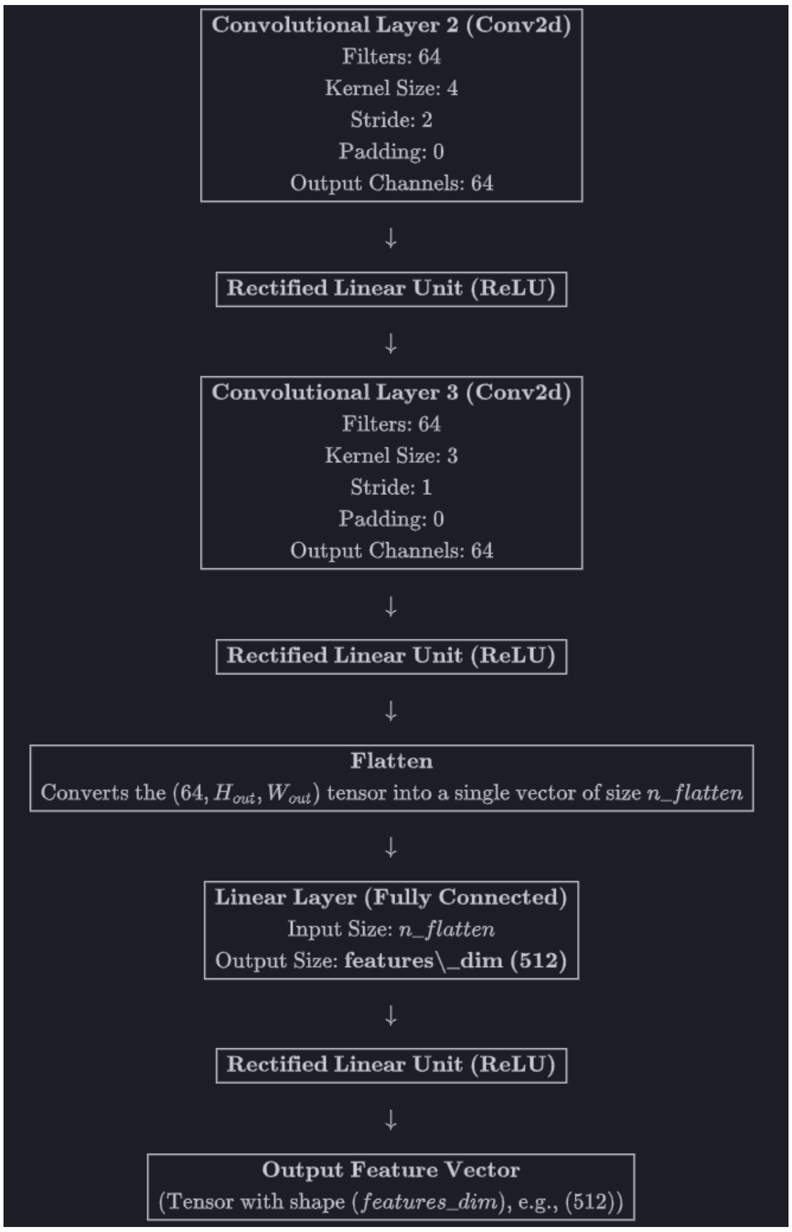


Figure 3. Source: https://www.mdpi.com/1996-1073/16/21/7286

Custom CNN flowchart:


Input Observation
(Tensor with shape $(C, 84, 84)$ where C is the number of stacked frames)
Convolutional Layer 1 (Conv2d)
Filters: 32
Kernel Size: 8
Stride: 4
Padding: 0
Output Channels: 32
Rectified Linear Unit (ReLU)


Convolutional Layer 2 (Conv2d)
Filters: 64
Kernel Size: 4
Stride: 2
Padding: 0
Output Channels: 64
Rectified Linear Unit (ReLU)
Convolutional Layer 3 (Conv2d)
Filters: 64
Kernel Size: 3
Stride: 1
Padding: 0
Output Channels: 64
Rectified Linear Unit (ReLU)
Flatten
Converts the $(64, H_{out}, W_{out})$ tensor into a single vector of size $n\_flatten$
Linear Layer (Fully Connected)
Input Size: $n\_flatten$
Output Size: **features_dim (512)**
Rectified Linear Unit (ReLU)
Output Feature Vector
(Tensor with shape $(features\_dim)$, e.g., (512))

Experimental results and discussion:

1a. Breakout

Experimental settings:

DQN:

The experimental setup utilizes the Atari Breakout environment, specifically the *BreakoutNoFrameskip-v4* version from the Gymnasium library. To enhance performance and stability, the environment is vectorized, running 4 parallel instances simultaneously. A frame stacking technique is employed, where each observation consists of a stack of 4 consecutive frames. This results in an input observation shape of (4, 84, 84) for the neural network.

The agent is based on the Deep Q-Network (DQN) algorithm and employs a custom Convolutional Neural Network (CNN) for feature extraction. This custom CNN architecture consists of three convolutional layers. The first layer has 32 filters with a kernel size of 8x8 and a stride of 4. The second layer has 64 filters with a 4x4 kernel and a stride of 2. The final convolutional layer uses 64 filters with a 3x3 kernel and a stride of 1. Each convolutional layer is followed by a Rectified Linear Unit (ReLU) activation function. The output from these layers is flattened and passed through a fully connected linear layer that maps the features to a final dimension of 512, also followed by a ReLU activation.

For the DQN agent's training, several key hyperparameters were configured. The learning rate was set to 0.0001. A replay buffer size of 100,000 transitions was used, with the learning process commencing only after this buffer was filled (learning_starts=100000). The model's weights are updated every 4 steps (train_freq=4), using a batch size of 32. The exploration strategy is an epsilon-greedy approach, where the value of epsilon linearly anneals from 1.0 to a final value of 0.01 over the first 10% of the total training steps (exploration_fraction=0.1).

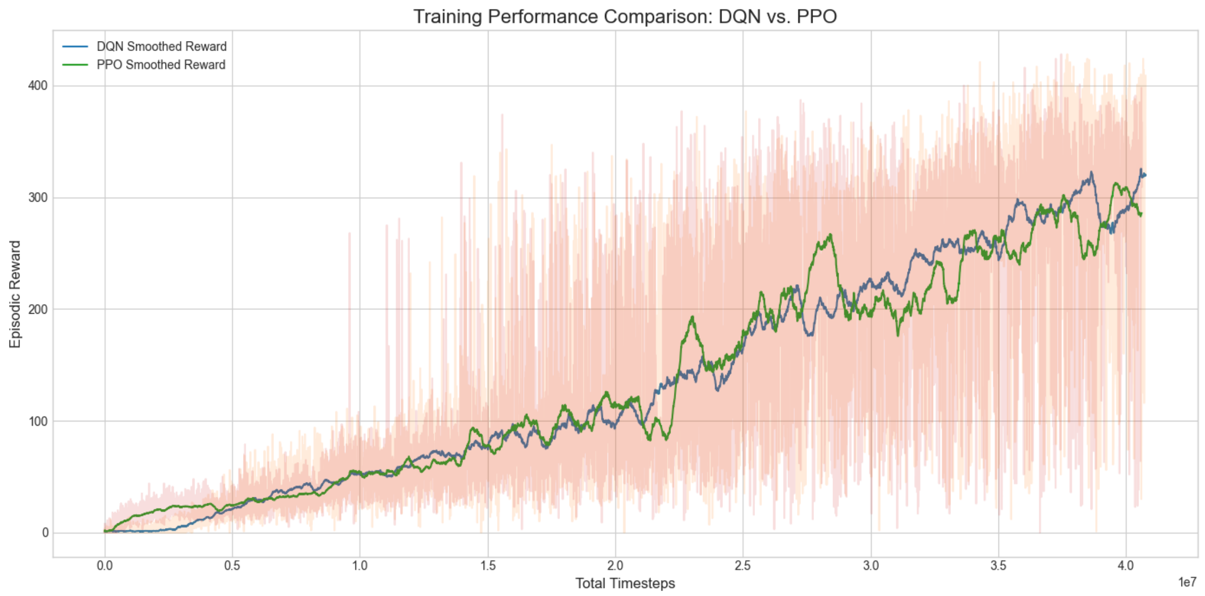
PPO:

For the second experiment, the Proximal Policy Optimization (PPO) algorithm was employed, using the same custom Convolutional Neural Network (CNN) architecture for feature extraction. The agent was trained in the *BreakoutNoFrameskip-v4* environment, which was vectorized to run 4 parallel instances with a frame stack of 4.

The PPO algorithm was configured with a learning rate of 2.5x10^(-4). For each policy update, data was collected for 128 steps (n_steps) from each of the parallel environments. The policy and value functions were then updated for 4 epochs (n_epochs) using mini batches of size 256. The discount factor (gamma) was set to 0.99, and the Generalized Advantage Estimation (GAE) parameter (lambda) was set to 0.95. The surrogate objective function utilized a clipping range of 0.1, and an entropy coefficient (ent_coef) of 0.01 was included in the loss function to encourage exploration.

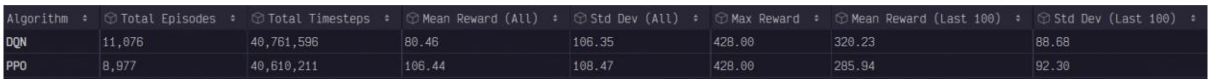
Experimental results:

Training Details:



As we can see here, for this specific environment, DQN showed to be more stable and to have a smoother reward training plot than PPO. It also got a better final reward for the last episodes.
https://github.com/emilianodesu/RLA2/tree/main/breakout/dqn/videos
https://github.com/emilianodesu/RLA2/tree/main/breakout/ppo/videos

Performance Evaluation:



Algorithm	Total Episodes	Total Timesteps	Mean Reward (All)	Std Dev (All)	Max Reward	Mean Reward (Last 100)	Std Dev (Last 100)
DQN	11,076	40,761,596	80.46	106.35	428.00	320.23	88.68
PPO	8,977	40,610,211	106.44	108.47	428.00	285.94	92.30

The most critical measure of performance, the final performance (**last 100 episodes reward**), show a better performance for DQN. Its final, trained policy was significantly better, scoring over 34 points higher on average than PPO's. This is the primary indicator of which algorithm ultimately "solved" the environment better. **(DQN: 320.23 vs PPO:285.94)**

As we can see on the **mean reward (all training)**, PPO was more sample-efficient. It achieved a higher average reward over the whole training process. This suggests that PPO learned a decent, "good enough" policy much faster than DQN. DQN likely spent a longer time at the beginning with a very poor policy before it started to learn effectively, which dragged its overall average down. (**DQN: 80.46 vs PPO:106.44**)

If we focus on the **standard deviation of the last 100 episodes**, The stability is very similar for both algorithms. A difference of less than 4 points on this scale is negligible. The initial assumption of PPO being noisier might be true visually, but the final policies are almost equally "noisy." High variance is expected in a stochastic game like Breakout, where a lucky or unlucky bounce of the ball can dramatically change the score. (**DQN: 88.68 vs PPO: 92.30)**

Regarding the **peak performance**, both algorithms could reach the exact same peak performance (**428.00**). This indicates that both methods explored the game effectively enough to find and execute a high-scoring strategy at least once. The key difference is that DQN's final policy was better at achieving high scores consistently.

Discussion:

Why did PPO learn faster initially? PPO is an on-policy algorithm with a clipped objective function that makes for very stable, incremental improvements. This often allows it to find a good policy quickly, as seen in its superior Mean Reward (All).

Why did DQN win in the end? DQN is an off-policy algorithm that uses a replay buffer. In a deterministic game like Breakout, this is a huge advantage. Once DQN discovers a highly successful strategy (like tunneling the ball up the side), it can store those "golden" experiences in its buffer and learn from them repeatedly. This repeated learning on optimal data allows it to fine-tune its Q-values to a higher degree of precision, leading to a superior final policy. PPO, being on-policy, must discard its experiences after each update and cannot reuse these golden moments.

Based on the performance metrics, the two algorithms demonstrate a clear trade-off between learning speed and final performance.

PPO proved to be the more sample-efficient algorithm, achieving a competent policy faster than DQN, as evidenced by its significantly higher Mean Reward.

However, DQN ultimately achieved a superior final policy. This is likely due to its replay buffer, which allowed it to repeatedly learn from high-quality experiences to master the game's strategy.

Both algorithms exhibited similar levels of variance in their final performance and reached the same peak score, indicating that for this Breakout task, PPO was the faster learner, but DQN was the better master.

1b. Cart Pole

Experimental settings:

DQN: For the CartPole training scenario, we employed the classic control environment CartPole-v1 provided by the Gymnasium library. We trained a Deep Q-Network (DQN) agent using a multi-layer perceptron (MLP) policy tailored for low-dimensional state spaces. The policy network consisted of two fully connected hidden layers with 256 neurons each, using the ReLU activation function. This deeper architecture was chosen to provide the model with sufficient representational capacity to learn stable Q-value approximations, while still being computationally efficient.

The DQN algorithm was configured with CartPole-specific hyperparameters optimized for stable and high-performance learning. We used a learning rate of 1×10^{-4}, which is lower than the default value to allow for smoother gradient updates and to avoid instability during Q-value estimation. The replay buffer size was set to 100,000 transitions to ensure diverse experience storage, and the agent began learning only after 10,000 steps, allowing the buffer to accumulate a representative set of experiences before the first gradient updates.

Training was performed every four environment steps, with a batch size of 64 sampled uniformly from the replay buffer. The target network was updated every 1,000 steps using hard updates (τ = 1.0), following the standard DQN formulation. We set the discount factor γ = 0.99, suitable for environments with delayed rewards. To balance exploration and exploitation, we used a linear epsilon decay strategy, where the exploration fraction was 0.1 of the total timesteps, and the final epsilon value was 0.01, allowing the agent to become increasingly greedy as training progressed. Additionally, we applied gradient clipping with a maximum norm of 10 to stabilize training. The model was trained for 1,000,000 timesteps to ensure convergence and robustness across different runs.

PPO: The policy and value function were parameterized using a two-layer MLP architecture, with 64 neurons per layer for both the policy (π) and value (vf) networks, and Tanh activation functions. This configuration mirrors the original PPO paper and provides sufficient expressiveness for solving CartPole efficiently while maintaining training stability.

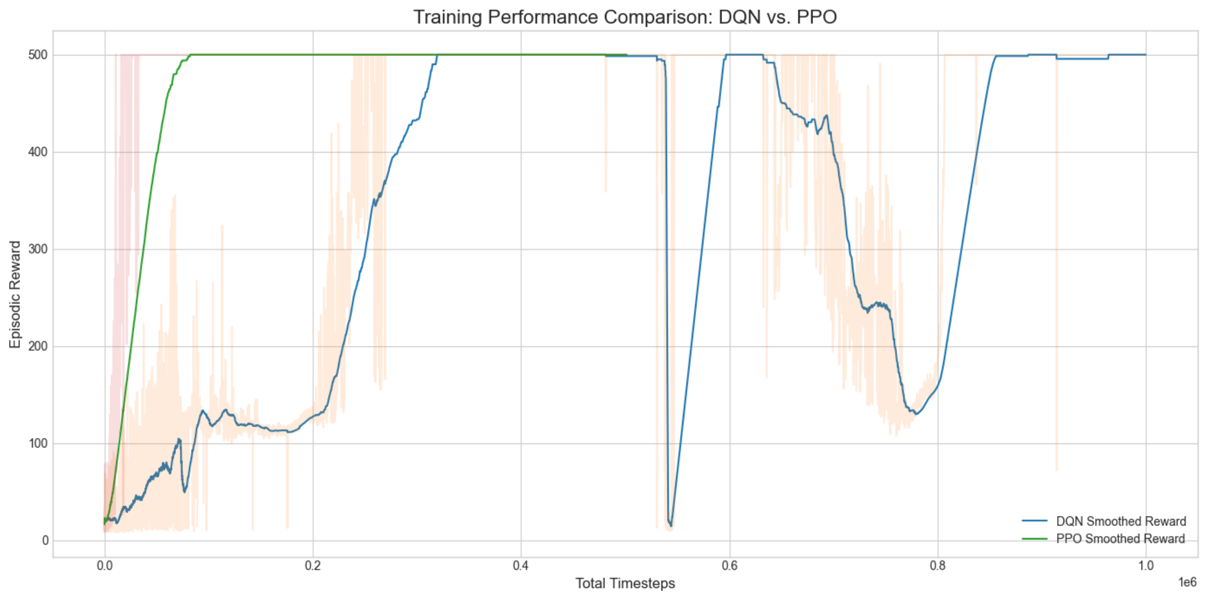
The PPO hyperparameters were chosen to maximize learning stability and efficiency. The learning rate was set to 3×10^{-4}, a well-established default for PPO that balances convergence speed with stability. Each policy update was based on 2048 environment steps (n_steps), producing a large batch of experience per iteration, which helps reduce gradient variance and improve training stability. The collected data was divided into minibatches of 64 samples, and the policy and value networks were updated for 10 epochs on each batch, ensuring effective use of collected samples. The discount factor was set to γ = 0.99, and GAE (Generalized Advantage Estimation) was used with λ = 0.95 to provide a low-variance but unbiased advantage estimator, which improves learning efficiency in PPO. We used a clip range of 0.2, following the standard PPO clipping mechanism to prevent destructive policy updates, and set the entropy coefficient to 0.0, as additional exploration incentives are unnecessary for CartPole. The value function coefficient remained at its default of 0.5, and gradient clipping with a maximum norm of 0.5 was applied to stabilize updates. The model was trained for 500,000 timesteps, which is typically more than sufficient for PPO to reach perfect performance (average reward of 500) on CartPole.

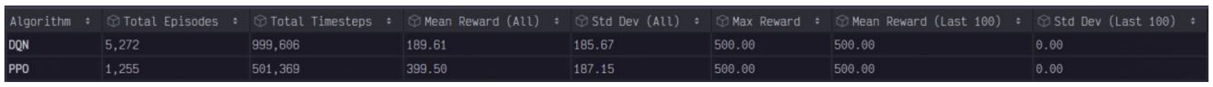
Experimental results:

Training Details:

https://github.com/emilianodesu/RLA2/tree/main/cartpole/dqn/videos

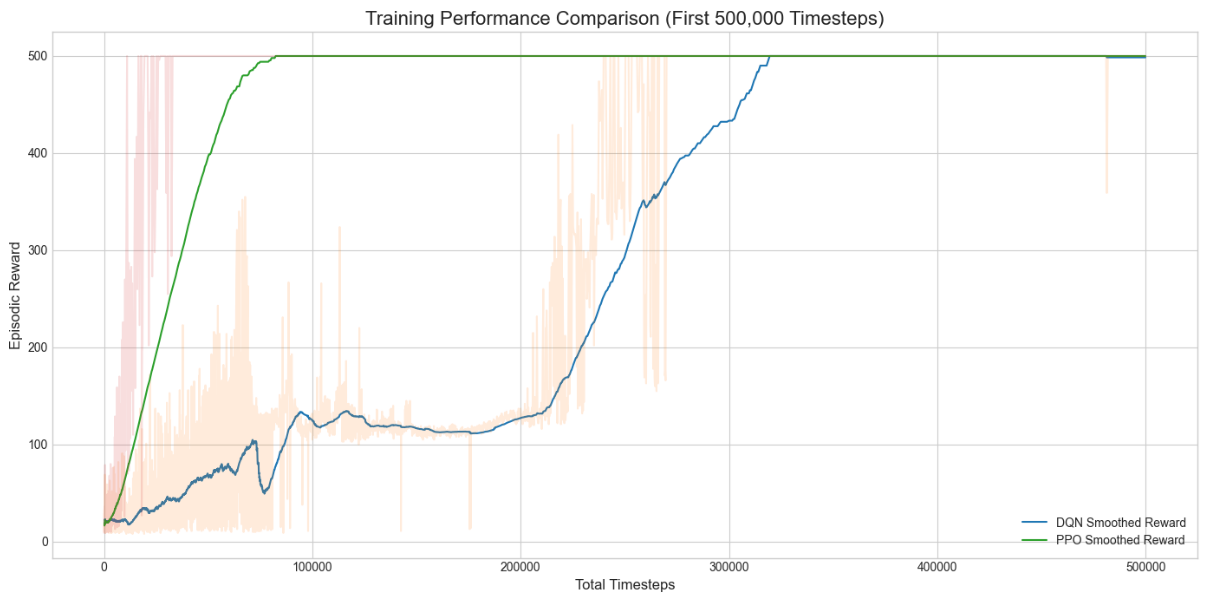
https://github.com/emilianodesu/RLA2/tree/main/cartpole/ppo/videos

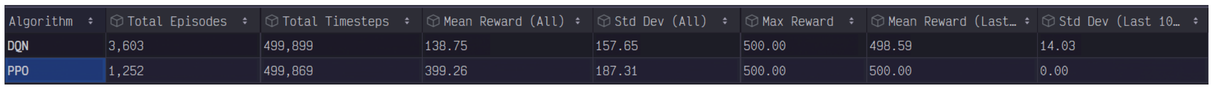




Algorithm	Total Episodes	Total Timesteps	Mean Reward (All)	Std Dev (All)	Max Reward	Mean Reward (Last 100)	Std Dev (Last 100)
DQN	5,272	999,606	189.61	185.67	500.00	500.00	0.00
PPO	1,255	501,369	399.50	187.15	500.00	500.00	0.00

We can zoom in on the first 500,000 steps as both algorithms could (nearly) solve the cartpole at this point. Let us notice that there is a clearly better-performing algorithm: PPO.





Algorithm	Total Episodes	Total Timesteps	Mean Reward (All)	Std Dev (All)	Max Reward	Mean Reward (Last…	Std Dev (Last 10…
DQN	3,603	499,899	138.75	157.65	500.00	498.59	14.03
PPO	1,252	499,869	399.26	187.31	500.00	500.00	0.00

Performance Evaluation:

Both algorithms ultimately solved the environment perfectly. A mean reward of 500 with a standard deviation of 0 means the agent is successfully balancing the pole for the maximum duration in every single episode. The key difference is how long it took them to reach this state of mastery.

We can see that PPO had already achieved a perfect final score (**Reward last 100 of 500**) in under 500,000 timesteps, DQN was close but not was yet perfectly consistent (**Std Dev last 100 of 14.03**). At 500,00 timesteps, for DQN, the agent was occasionally dropping the pole early. Both achieved perfect stability (0.00), but DQN took twice as long to get there.

Discussion:

Why was PPO so effective? PPO is an Actor-Critic method that directly learns a policy. For a control problem like CartPole, which requires continuous, fine-tuned adjustments, directly optimizing the policy is a very natural and efficient approach. The policy gradient updates allow PPO to smoothly "nudge" its strategy towards the optimal balancing act. Its stability, enforced by the clipped objective function, prevents it from taking dangerously large steps that would destabilize learning.

Why was DQN not so inefficient? DQN is a value-based method. It tries to learn the precise value of taking action 'left' or 'right' in every possible state. For a balancing problem, the difference in value between left and right can be very small and change rapidly, making the Q-function complex and difficult to learn accurately. The process of propagating rewards back through time (temporal difference learning) is also less direct than PPO's policy updates for this kind of task, leading to slower learning.

This dramatic difference in learning speed is reflected in the overall average rewards, where PPO (399.50) vastly outperformed DQN (189.61). The data clearly shows that PPO's Actor-Critic approach is exceptionally well-suited for the stable control policy required by CartPole, allowing it to learn faster and more consistently. DQN, while capable, proved to be far less efficient for this specific task.

2. Pendulum

Experimental settings:

For the Pendulum-v1 environment, we implemented a custom Q-learning algorithm adapted to handle continuous state and action spaces through discretization. The original environment provides a three-dimensional continuous state space $(\cos(\theta), \sin(\theta), \dot{\theta})$and a one-dimensional continuous action space corresponding to the torque applied to the pendulum. Since standard tabular Q-learning requires discrete state–action pairs, we applied discretization both to the observation space and the action space to make the problem tractable.

The state space discretization was performed using a custom Discretizer wrapper that transformed each continuous state dimension into a discrete index. Specifically, we divided each of the three state dimensions into 50 equally spaced bins, resulting in a discretized state space represented as a tuple of three integers, each indicating the bin index for one dimension. This effectively created a state space grid of size $50 \times 50 \times 50$, which allows the agent to learn approximate Q-values for different regions of the continuous space.

The action space was discretized separately into 25 equally spaced bins, spanning the original action space bounds of the pendulum environment. Each discrete action index corresponds to a continuous torque value, ensuring that the agent's selected actions remain valid within the environment. This discretization enables the use of ε-greedy exploration and standard Q-learning updates on a fixed set of discrete actions.

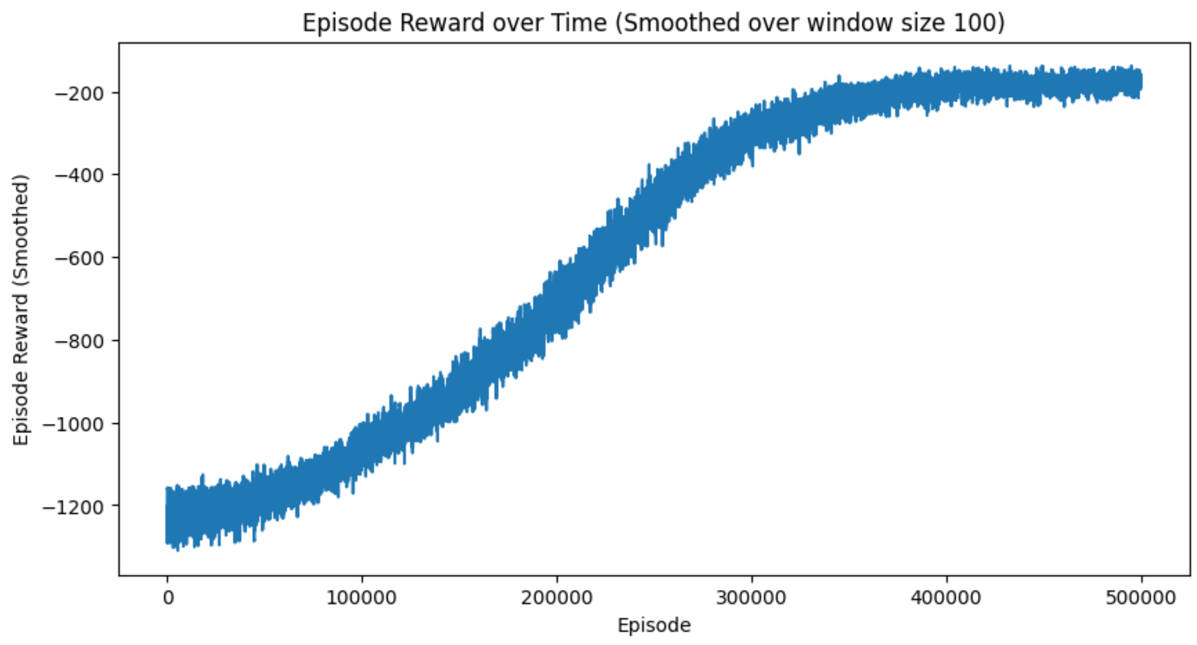
The Q-learning algorithm was configured with a learning rate (α) of 0.1, a discount factor (γ) of 0.99, and an ε-greedy exploration strategy. The initial exploration rate was set to ε = 1.0, encouraging purely random exploration at the beginning of training, and decayed linearly to ε = 0.01 over approximately 80% of the training episodes to gradually shift toward exploitation as learning progressed. The agent was trained for a total of 500,000 episodes, which was necessary given the large discretized state space and the complexity of the continuous control task. A default dictionary was used to represent the Q-table, with states stored as tuples of discrete indices and each entry containing a Q-value vector for the 25 discrete actions.

During training, at each step the agent selected actions according to the ε-greedy policy, executed them in the original continuous environment, received scalar rewards, and performed standard tabular Q-learning updates using the Bellman equation. This setup provided a simple but effective way to apply Q-learning to a continuous control environment, allowing the agent to approximate optimal policies by learning over a finely discretized representation of the state and action spaces.

Experimental results:

Training Details:

https://github.com/emilianodesu/RLA2/tree/main/pendulum/videos



The training results show a S-shaped learning curve, with episode rewards initially near the environment's baseline and gradually improving before plateauing around –200.

Performance Evaluation:

The curve shape suggests that the agent was indeed learning a more effective policy over time, but the performance eventually stabilized at a suboptimal level, rather than converging to near-optimal control (which would correspond to average episode rewards close to 0, the theoretical maximum for Pendulum).

The **mean episode reward of –595.39** over the entire training period reflects the fact that early training episodes were dominated by random exploration, as the agent began with ε = 1.0 and took uniformly random actions for a large number of episodes. This resulted in many trajectories where the pendulum quickly lost balance and the agent applied poorly directed torques, leading to large negative rewards. As ε decayed and the agent exploited its learned Q-values, the policy improved, producing the observed upward inflection in the curve. However, the performance plateau indicates that the policy never fully converged to an optimal balancing strategy, which is consistent with the limitations of using a tabular Q-learning approach on a continuous control problem.

Discussion:

A possible reason why the agent topped out at around –200, is that tabular Q-learning has no function approximation, so it cannot generalize between nearby continuous states; each discretized state is treated independently, making learning inefficient. Finally, the reward structure of Pendulum is dense and continuous, which often requires more advanced methods—such as actor–critic algorithms or deep Q-networks with continuous action representations to learn smooth control policies.

In summary, the S-shaped curve indicates some successful learning, as the agent improved from fully random behavior to moderately effective control. However, the plateau at –200 and the low overall mean reward highlight the inherent limitations of applying tabular Q-learning with discretization to a continuous control task. These results are expected: while discretization makes the problem tractable, it also imposes significant performance ceilings that are difficult to overcome without finer action resolution, better state generalization, or more sample-efficient algorithms.

Conclusion

The Pendulum experiment immediately highlighted the limitations of basic methods; applying tabular Q-learning to a continuous state and action space required discretization, which created an unavoidable performance ceiling due to the loss of information and inability to generalize between states. For the more complex tasks, the implementation shifted to deep learning with Stable Baselines3. The Breakout environment, being vision-based, necessitated a Custom CNN to process pixel data. Here, a fascinating trade-off emerged: PPO was more sample-efficient and learned a competent strategy faster, but DQN's use of a replay buffer allowed it to achieve a superior final policy by repeatedly learning from its best experiences. Conversely, in the vector-based CartPole environment, PPO's direct policy optimization was overwhelmingly more efficient, solving the task in half the time it took the value-based DQN, which struggled with the less direct learning approach. Overall, we successfully implemented and trained different gymnasium scenarios demonstrating a solid understanding of the Q-learning, DQN and Policy Search techniques.