Learning to Play Atari Pong: A Study of Reinforcement Learning Techniques

David Fuentes Insa

30 de juny de 2025

Resum– Aquest projecte té com a objectiu establir una base sòlida sobre l'aprenentatge per reforç, entenent-ne els fonaments, aprenent a implementar-lo a través de diversos algoritmes i explorant com pot aplicar-se als videojocs. Amb l'augment de la importància de la intel·ligència artificial en múltiples sectors, l'aprenentatge per reforç destaca com un paradigma potent per entrenar agents a prendre decisions mitjançant la interacció i l'experiència. L'estudi comença amb una anàlisi de conceptes teòrics com els processos de decisió de Markov, els entorns, les recompenses, les polítiques i els agents. S'estudien, implementen i avaluen algoritmes de RL tabulars i aproximats aplicats a agents individuals en entorns de videojocs. Els videojocs s'utilitzen com a entorns interactius i controlats per observar el comportament dels agents i el rendiment dels algoritmes. Finalment, l'estudi aprofundeix en tècniques d'entrenament multiagent, significativament més complexes que l'entrenament d'un sol agent.

Paraules clau– Aprenentatge per reforç, intel·ligència artificial, aprenentatge d'un sol agent, aprenentatge multiagent, videojocs, aprenentatge profund, atari pong

Abstract– This project aims to get a solid base of Reinforcement Learning, understanding the foundations, learning how to implement it across multiple algorithms and how can be applied to videogames. As AI becomes increasingly important across many industries, RL stands out as a powerful paradigm for training agents to make decisions through interaction and experience. The study begins with an analysis on theoretical concepts such as Marko Devision Processes, enviroments, rewards, policies and agents. Tabular and approximate RL single-agent algorithms are studied, implemented and evaluated, in videogames enviroments. Videogames are used as controlled, interactive enviroments to test learning behavior and algorithm performance. The study then proceeds to dive into multi-agent training techniques, much more complex than training a single-agent.

Keywords– Reinforcement Learning, Artificial Intelligence, Single-agent RL, Multi-agent RL, Videogames, Deep learning, Atari Pong

---

✦

---

1 INTRODUCTION

REINFORCEMENT learning is one of the most promising approaches of artificial intelligence, especially in the videogames sector. Meanwhile other AI approaches use supervised training with labeled data, RL agents learn by interacting with the enviroment throught tri-

al and error, and improve based on rewards or penalties for taking actions.

The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics —trial-and-error search and delayed reward— are the two most important distinguishing features of reinforcement learning [1].

In 2013, the company DeepMind released a paper regarding how their RL models could surpass a human expert playing Atari 2600 classic games. The model beat the hu-

---

- E-mail de contacte: David.FuentesI@autonoma.cat
- Menció realitzada: Computació
- Treball tutoritzat per: Jordi Casas Roma
- Curs 2024/25

man expert in 3 of 6 games tested [2]. In 2015, they released a more extended paper tested on 49 games. The model got to the level of a human professional [3]. The most impressive achievement is AlphaGo, the model that beat the European Go champion Fan Hui by 5 games to 0, and later defeated the world champion Lee Sedol [4]. That marked a historic milestone artificial intelligence.

Videogames provide an ideal enviroment to test the models because of the clear objectives, controlled actions and a closed world. However, reinforcement learning applications extend not only in videogames, also in real life problems such as robotics, autonomous vehicles, healthcare, resource management, and much more.

2 OBJECTIVES

The main objective of this project is fully understanding SARL and what can be acomplished with it using existing libraries but also how it works from zero. The next five objectives define the goal by the end of the project.

1. Learn RL Fundamentals: Study core RL concepts such as Markov Decision Processes (MDPs), value functions, and policy optimization. Implement simple algorithms like Monte Carlo or Dynamic Programming to build a solid base.

2. Learn RL Advanced Methods: Such as Temporal Difference learning (Q-learning and SARSA) and deep RL approaches like DQNs.

3. Implementing some algorithms from zero, without using pre-built libraries.

4. Implementing image based DQNs that are more complex and harder to train.

5. Experiment with self-play, multi-agent training in a videogame: Using a ping pong enviroment, train the agents from both sides of the game.

3 METHODOLOGY AND PLANNING

The methodology chosen is Kanban, an agile approach that has a clear visualization of the workflow, the evolution is continuous and is very flexible.

The tool used for the implementation of Kanban is Clickup[1], a free tool for project management.

The planning is divided in five phases of three or four weeks each.

- Phase 1: Reinforcement learning fundamentals and the State of Art (10/03/25 - 19/04/25)

  – Task 1: Research and document core RL concepts (agents, environments, states, actions, rewards)

  – Task 2: Study tabular methods (Dynamic Programming, Monte Carlo, Q-learning, SARSA)

  – Task 3: Study approximate methods (DQN)

  – Task 4: Document the State of Art

[1]https://clickup.com/about

- Phase 2: Enviroment setup and algorithms (18/04/25 - 25/05/25)

  – Task 5: Study gymnasium library

  – Task 6: Implement approximate methods

  – Task 7: Implement tabular methods

  – Task 8: Document the implementations and tests

- Phase 3: Deep Q-Network (DQN) applied to image-based game (04/05/25 - 25-05-25)

  – Task 9: Select enviroment and study the DQN algorithm

  – Task 10: Implement and train the agent

  – Task 11: Test the algorithm

  – Task 12: Document the implementations and tests

- Phase 4: Self-play (2025-05-20 - 2025-06-14)

  – Task 13: Study petting zoo library

  – Task 14: Implement the training and testing algorithm

  – Task 15: Train the agents implemented

  – Task 16: Test the model, and evaluate the models performance

  – Task 17: Document the implementations and tests

- Phase 5: Final inform and conclusions (2025-06-14 - 2025-06-30)

  – Task 18: Document the and results conclusions

  – Task 19: Final inform

  – Task 20: Project presentation

4 STATE OF ART

4.1 Reinforcement Learning vs other types of learning paradigms

In reinforcement learning, there is an agent which learns while interacting in the environment and is given some feedback such as rewards or penalties in order to guide the behavior. In supervised learning, the model is trained on a data set having the inputs along with the correct outputs such that the model is able to learn the direct mapping of the inputs to the targets. Unsupervised learning is operating on the data where there are no labels, and the goal is to discover the patterns or the structure in the input. Self-supervised learning creates its own labels from the data itself, allowing the model to learn useful representations without manual annotation. Semi-supervised learning combines a small amount of labeled data with a large amount of unlabeled data to improve learning performance when labeling is expensive or time-consuming.

4.2 Reinforcement Learning Fundamentals

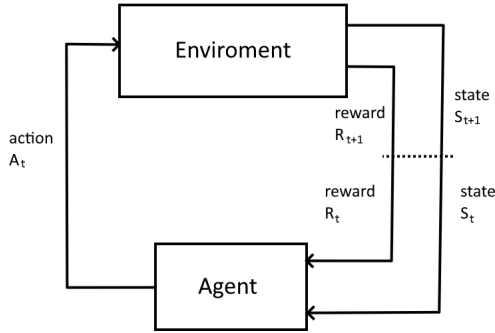These are the basic components a reinforcement learning problem has, illustred in Figure 1

Fig. 1: Enviroment agent schema

• An agent — it's the entity that makes decisions. It's objective is to learn a policy ($\pi(a|s)$) that maximizes the accumulated reward. It can be a robot, a player in a videogame, etc. • Enviroment — responds to the agent actions. It provides the states and the rewards, and changes based on the agent actions. • States — Represent the actual enviroment situation by the perspective of the agent. • Actions — The possible decissions an agent can do in a determinate state s. Choosing an action makes a transition to a new state. • Rewards — Numeric values that indicate the immediate utility of an action. The agent must maximize the rewards at long term. An immediate reward r_t is a reward obtained in the time t. The return G_t, is the future rewards sum with a discount factor γ, which determines how much the agent values future rewards. If its near 0, the agent only cares for immediate rewards. If it's near 1, the agent will appreciate much more the future rewards.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

• Policy — Defines the agent behavior. It's the strategy the agent follows. $\pi(a|s) = P(A_t = a \mid S_t = s)$

Example: What would these components be in a reinforcement learning problem applied in the videogame of Space Invaders? The agent, is the spaceship that moves horizontally and shoots the aliens. The enviroment, would be the aliens, the visual scenary, the alien bullets and all the game interface like lives, points. The state could include: position of the agent spaceship, position of all the aliens, position of every bullet, the lives remaining and the image frame. The actions: move left, move right, shoot, doing nothing. The rewads: +1 for shooting an enemy, -1 for loosing a live and 0 for not doing anything for example. The policy: If an enemy is just up the agent, shoot.

4.3 Epsilon greedy policies

Epsilon-greedy is a simple strategy to balance exploration and exploitation. With probability ϵ, the agent chooses a random action (exploration), and with probability $1 - \epsilon$, it chooses the best-known action (exploitation). This helps the agent discover better actions over time while still using what it has learned.

4.4 Markov Decision Process

A MDP is the math model that formalizes a RL problem. It is defined by a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of all possible actions, $P(s'|s, a)$ is the probability of transitioning to state s' when taking action a in state s, $R(s, a)$ is the expected immediate reward received after taking action a in state s, and $\gamma \in [0, 1]$ is the discount factor.

The Markov property states that the probability of transitioning to the next state depends only and only on the current state and action, not on the sequence of previous states and actions:

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots)$$

4.5 Bellman equation

The Bellman equation allows us to calculate the value of each state in the state table, assuming we are following a given policy π. In others words, it tells us how good each state is when the agent behaves according to that policy. By applying the Bellman equation to every state, we can iteratively fill in or update the entire value table.

This is useful because, in reinforcement learning, we usually don't know beforehand how good a state is we estimate it based on the rewards the agent receives and the expected values of future states. The Bellman equation makes this estimation possible, step by step, using experience and a fixed policy.

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right]$$

4.6 Tabular methods

Ideal when the enviroment is small and manageable. State-action values can be stored in tables.

4.6.1 Dynamic Programming

Based on a simple idea, if we know exactly how the enviroment works (what will happen with a probability when we take an action in a specific state), we can calculate which decisions are better in long term. This is very limited, because in the real world we don't know complete knowledge of the enviroment [5].

There are three algorithms used in DP: • Policy evaluation: Calculates how good is the current policy using Bellman equation in every state using that specific policy. • Policy iteration: Firstly uses policy evaluation and then improves the current policy choosing the best actions based on the calculated values. • Values iteration: This method updates the values of every state with a simplified Bellman equation, to get the optimal policy.

4.6.2 Monte Carlo

Does not need to know the model of the enviroment. Learns based on experience. This means the agent does not need to know how the environment behaves or the probabilities of state transitions. Instead, it learns purely from experience by interacting with the environment and observing the

results. Over time, by accumulating many episodes of experience, the agent can estimate the long-term value of states or state-action pairs based on actual returns received. This makes Monte Carlo methods especially useful in complex or unknown environments where modeling the dynamics is difficult or impossible.

The monte carlo algorithms works like this: The agent follows a policy [6]. Plays an episode till the end. At the end of the episode, calculates how much has won in total from every state. Updates the value estimates. Improves the policy, to take better actions in the next episodes

What is an episode? A sequence of states starting from one state to the end state when a condition is met.

There are two main objectives in Monte Carlo algorithms. • Prediction: Follow a specified policy and evaluate how good the policy is. • Control: To learn the best actions the agent can do, keep adjusting the policy.

And there are two approaches on how to update what the agent learnt. • First-visit: Updates the value of the state only the first time it appears on an episode. • Every-visit: Updates the value of state every time it appears.

4.6.3 Temporal Difference

Learn from previous experience and don't need to end an episode like Monte Carlo algorithms. They are ideal for tasks that do not have an end (like maintaining the temperature of an oven that is always on, or to move a character in a videogame on an open world without a clear objective). Combines ideas from Monte Carlo and Dynamic Programming [7].

TD algorithms update the value of a state based on future estimates. When the agent changes the state and gets a reward, the state value gets immediately updated. This update is based on the **temporal difference (TD) error**, which measures the difference between the current Q-value and a new estimate based on the reward received and the next state's predicted value. It is the core learning signal used to adjust the Q-value closer to reality.

There are two main algorithms based on TD: • SARSA — means State-Action-Reward-State-Action. To learn, must know the state, the chosen action, the reward obtained, the next state and the next chosen action. Learns from what you are doing, even if that is not the best possible.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right]$$

• Q-learning — is almost the same, but learns from the best action possible.

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

4.7 Approximate methods

Methods where every action-state value cannot be stored on a table because it will be infinite or impossible to manage.

4.7.1 DQN

Deep Q-Network (DQN) combines Q-learning with deep neural networks [3]. The table is replaced for a neural network, that can estimate the values.

How do DQN Algorithms (Algorithm 4) work? You send the actual state to the network input (for example, the game image or the needed variables) and the network outputs the estimate values for every action possible. Then, the greatest value is chosen and executed, and get the new state and the reward. It utilizes temporal difference to calculate the error between the predicted Q-value and a new estimate based on the reward and the next state. Finally, the DQN Network is updated.

4.7.2 REINFORCE

Reinforce [10] is a policy gradient algorithm that learns how to act directly, instead of learning how good each action is (like DQN methods). Instead of using a Q-table or estimating values, Reinforce tries to learn the best policy, the best way of acting, by adjusting the probabilities of choosing an action in each situation.

How do Reinforce works? It plays an episode, then it looks what happened. If an action led to a good reward, it will make that action more likely to be chosen next time. Otherwise, if an action led to bad rewards, it will make it less likely to be chosen. Gradually improves how it plays.

5 SINGLE-AGENT LEARNING

5.1 The gymnasium library

Gymnasium [2] is a python library, forked from OpenAI Gym Library, that provides a standarized interface for creating enviroments where the agents can interact and learn from. The library provides a clear API aligned with the Markov Decision Processes where the RL agents can observe a state, choose an action and receive a reward.

FrozenLake-v1[3] and CartPole-v1[4] are two Gymnasium enviroments that are used in the tabular and approximate implementations of RL algorithms.

FrozenLake, as shown in Figure 2, is a a discrete grid. The agent must learn to navigate a frozen lake without falling into holes.
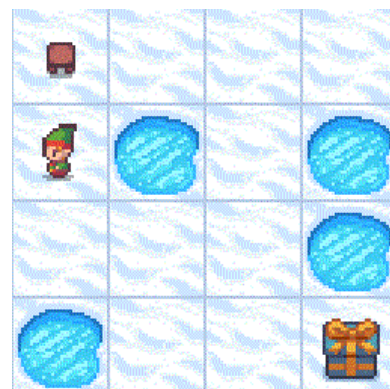

Fig. 2: FrozenLake-v1

Meanwhile CartPole, as shown in Figure 3, is a classic

control problem with continuous states. The goal is to balance a pole on a moving cart.
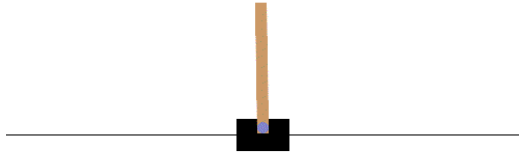

Fig. 3: CartPole-v1:

5.2 Implementing tabular methods

5.2.1 Implementation of the algorithms

Three tabular methods were implemented and tested on the FrozenLake-v1 enviroment: Monte Carlo (Algorithm 1), SARSA (Algorithm 2) and Q-Learning (Algorithm 3). FrozenLake has an important enviroment option: slippery. If slippery is true the game is not deterministic and the agent has a 30% chance to slide and choose a random The agent can choose from 4 discrete actions: left, right, up, down. Reward shaping was implemented because the original FrozenLake enviroment provides plain and binary rewards: 1.0 reward if the agent reaches the goal, 0.0 otherwise. To accelerate learning and guide the agent more effectively, the following shaped rewards were applied: 1.0 when reaching the goal, -1.0 if the agent falls into a hole, -0.01 if the agent does a step.

Algorithm 1 Monte Carlo Control (First-Visit)

1: Initialize $Q(s, a)$ arbitrarily
2: **for** each episode **do**
3:     Initialize empty list $\mathcal{E} \leftarrow []$ ▷ stores (s, a, r)
4:     Initialize state s
5:     **while** episode not ended **do**
6:         Choose action a using ε-greedy policy
7:         Take action a, observe reward r and next state s'
8:         Append (s, a, r) to $\mathcal{E}$
9:         $s \leftarrow s'$
10:     **end while**
11:     **for** each (s, a) in $\mathcal{E}$ (first visit only) **do**
12:         Compute $G \leftarrow$ total discounted return after first occurrence
13:         $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [G - Q(s, a)]$
14:     **end for**
15: **end for**

Algorithm 2 SARSA (On-policy TD Control)

1: Initialize $Q(s, a)$ arbitrarily
2: **for** each episode **do**
3:     Initialize state s
4:     Choose action a using ε-greedy policy
5:     **while** episode not ended **do**
6:         Take action a, observe reward r and next state s'
7:         Choose next action a' using ε-greedy policy
8:         $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [r + \gamma Q(s', a') - Q(s, a)]$
9:         $s \leftarrow s'$
10:         $a \leftarrow a'$
11:     **end while**
12: **end for**

Algorithm 3 Q-Learning (Off-policy TD Control)

1: Initialize $Q(s, a)$ arbitrarily
2: **for** each episode **do**
3:     Initialize state s
4:     **while** episode not ended **do**
5:         Choose action a using ε-greedy policy
6:         Take action a, observe reward r and next state s'
7:        $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
8:         $s \leftarrow s'$
9:     **end while**
10: **end for**

5.2.2 Results

All the same hyperparameters were used for every algorithm: 100000 episodes, 150 max steps of an episode, alpha (α) 0.1, gamma (γ) 0.99, 1.0 initial epsilon, 0.9999 epsilon decay every episode, and the minimum epsilon 0.01. The enviroment was set to 8x8 grid with 0.8 probability of holes. 30000 episodes and 150 max steps were used for testing every algorithm.
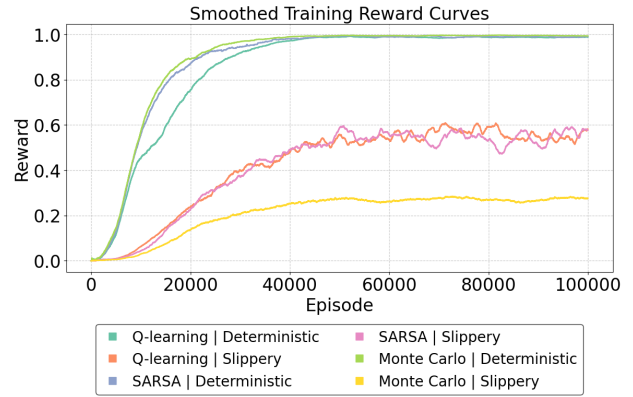

Fig. 4: Smoothed training reward curves

All the algorithms performed with good results (Figure 12) in the deterministic enviroment archieving a 100% success on all the tests. Monte Carlo was a little bit slower in the training of deterministic enviroment and performed worse when using non-deterministic enviroment. Q-learning performed the best on non-deterministic archieving more than a 70% success. Monte Carlo performed poorly on the non-deterministic reaching just less than 35% success rate.
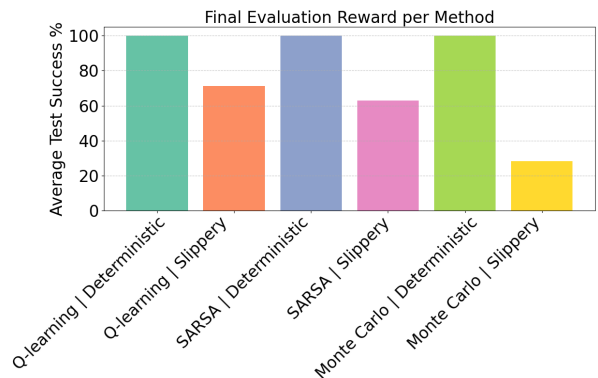

Fig. 5: Tested success (%)

5.3   Implementing approximate methods

5.3.1   Implementation of the algorithms

One approximate method was implemented on the CartPole-v1 enviroment: Deep Q-Network. The Q-Network consists of two hidden layers with 64 neurons each and ReLU activations. The input of the network is the observable space of the enviroment: Cart position, cart velocity, pole angle and pole angular velocity. The output is 2 discrete actions: moving the cart left and moving the cart right. The algorithm, with the pseudocode in Algorithm 4, begins by initializing the Q-network with random weights and an experience replay buffer to store past transitions. During each episode, the agent interacts with the environment by selecting actions using an e-greedy policy having a random chance of selecting a random action or choosing the best action according to the current Q-network, which estimates the expected future rewards, balancing exploration and exploitation. After each action, it observes the resulting reward and next state, and stores the transition in the replay buffer. When enough experiences have been collected, the algorithm samples a minibatch of transitions from the buffer to update the Q-network. For each sampled transition, it calculates a target value: if the transition ends the episode, the target is the reward; otherwise, it is the reward plus the discounted maximum Q-value of the next state. The network is trained to minimize the difference between the predicted Q-value and this target. Over time, the exploration rate e decays to allow the agent to shift from exploration to exploitation.

Algorithm 4 Deep Q-Network (DQN)

1: Initialize Q-network with random weights
2: Initialize replay buffer $\mathcal{B}$
3: **for** each episode **do**
4:      Initialize state s
5:      **for** each step of the episode **do**
6:          **if** random number $< \varepsilon$ **then**
7:              Choose random action a
8:          **else**
9:              $a \leftarrow \arg\max_a Q(s, a)$
10:         **end if**
11:         Execute action a, observe reward r and next state s'
12:         Store $(s, a, r, s', done)$ in buffer $\mathcal{B}$
13:         $s \leftarrow s'$
14:         **if** buffer $\mathcal{B}$ has enough samples **then**
15:             Sample minibatch of transitions $(s, a, r, s', done)$ from $\mathcal{B}$
16:             **for** each transition **do**
17:                 **if** *done* **then**
18:                     target $\leftarrow r$
19:                 **else**
20:                     target $\leftarrow r + \gamma \cdot \max_{a'} Q(s', a')$
21:                 **end if**
22:                 Compute loss: $\mathcal{L} = (Q(s, a) - \text{target})^2$
23:                 Update Q-network to minimize $\mathcal{L}$
24:             **end for**
25:         **end if**
26:     **end for**
27:     $\varepsilon \leftarrow \max(\varepsilon_{\min}, \varepsilon \cdot \text{decay\_rate})$
28: **end for**

5.3.2   Results

The hyperparameters were: 1000 episodes, 200 test episodes, 500 max steps of an episode in training and 1000 in testing, 32 batch size, gamma (γ) 0.99, alpha (α) 1e-4, 1.0 initial epsilon, 0.99 epsilon decay every episode and minimum epsilon 0.1.
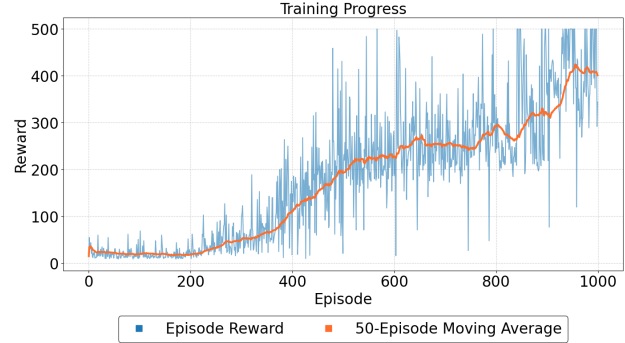

Fig. 6: Smoothed training reward curves

200 episodes and 1000 max steps were used for testing the algorithm. The best model using a 10 episode average was saved and used for the tests. The model performed very good even with very little episodes in the training. The agent is performing more than 500 steps even reaching 1000 sometimes (Figure 7).
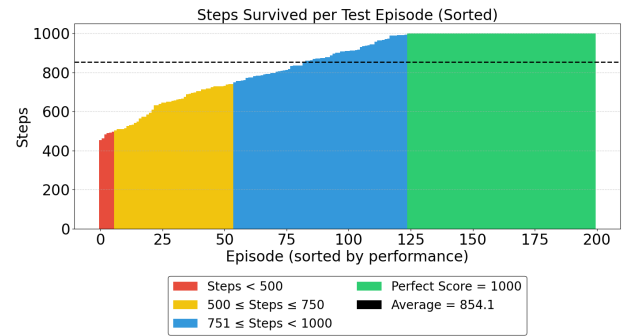

Fig. 7: Steps archieved for every test episode

5.4   Deep Q-Network applied to image based game

5.4.1   Implementation of the algorithms

Pong from Atari is a classic Atari 2600 game and it's the enviroment it will be used to traing and evaluate a reinforcement learning. The difference between this enviroment and the CartPole-v1, which we also trained with a DQN, is that we now have a bigger input in the deep neural network. The input is the preprocessed as shown in Figura 8 - Figura 10. Firstly, is transformed to 84x84 in greyscale images. Also FrameSkip of 4 is used (4 frames) to remove useless in-between frames. Then FrameStack of 4 is used (4 frames), to capture motion. A deep neural network with 3 convolutional layers (extracting spatial and motion features), followed by a flattening layer and 2 fully connected layers, ending in 6 outputs representing the Q-values for each possible action. In the training, the agent utilizes a epsilon-greedy policy to explore the space of actions.

The model was trained on the cloud using Kaggle[5] using GPU acceleration (T4 x2, 15GB VRAM each) and 29 GB of RAM for faster iteration cycles and better performance. To stabilize training, the DQN architecture uses two separate networks: the policy network and the target network. The policy network is the one being actively trained and used to select actions, while the target network is a copy of the policy network that is updated less frequently (e.g., every few thousand steps) and is used to compute the target Q-values. This separation helps reduce oscillations and divergence during training.
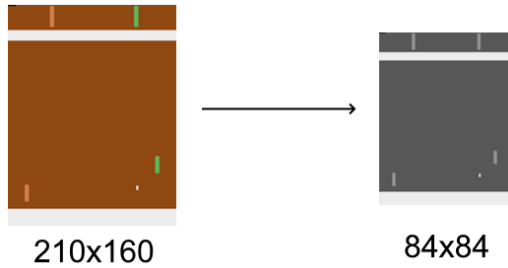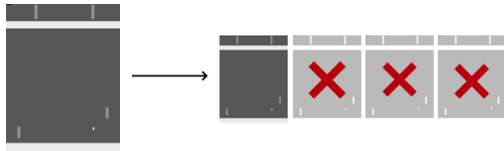

Fig. 8: Preprocessing
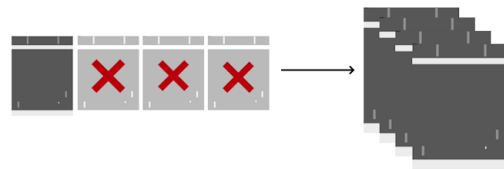

Fig. 9: Frame skipping, getting a frame every 4

Fig. 10: Frame stacking, getting 4 frames for the input
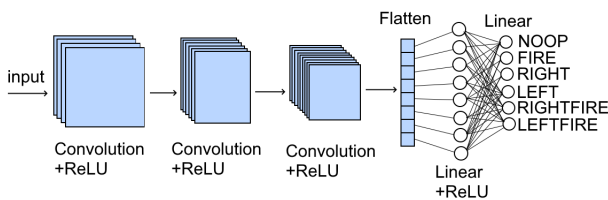

Fig. 11: Deep Q Network, input is 4 frames stacked and ouput the best actio

5.4.2 Results

The training hyperparamters: 8M frames (11200 episodes), gamma (γ) 0.99, alpha (α) 1e-4, Batch-size 32, buffer size 100k, buffer started with 50000 random actions. In the training Figure 12, epsilon started with 1 to 0.1 from A to B.

Great progress can be seen here – and this is normal, because the agent is doing a lot of exploration (random actions), which makes the model take non correct actions. When the epsilon decays, the model improves proportionally. From B to C the epsilon was static in 0.1. A little progress was made here, maybe if more frames were trained with the epsilon on 0.1, the model would have get better. And finally, from C to the end epsilon was changed to 0.0 to test if that could give the model a bit more progress. No progress or very little progress is seen.

Some limitations the training process had – 32 GB of RAM which limited the buffer size to 100k. This contrasts with the original 2015 DQN implementation by Mnih et al. [3], which used a replay memory buffer of 1 million transitions, allowing the agent to sample a much wider variety of past experiences during training and improving the stability and performance of learning.
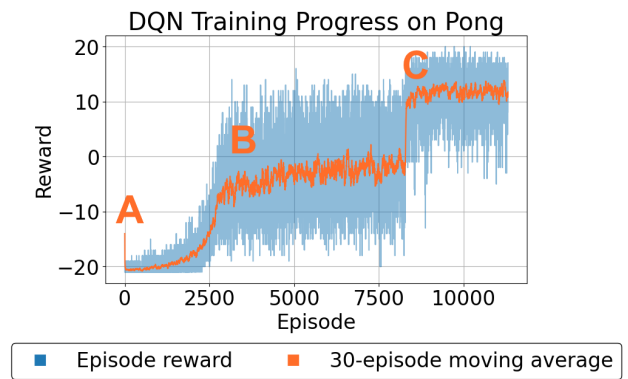

Fig. 12: Smoothed training progress

To choose the best model possible to evaluate, 3 options were tested on 200 episodes, getting the trained network at different points. The best average reward model in training within 10 episodes window, the last model when training finished and the best reward episode (a model that got the last training episode reaching 20 reward). The model that had the best performance on the tests was the best average reward model in training with 12.61 average reward, is the model in the Figure 13 and Figure 14. The other two models had 12.35 (last model after training) and 11.14 (the best episode model, with reward of 20). The model won 199 out of 200 test episodes. Mostly, the model wins by more than 7 points and more than half of the time it wins by more than 13 points.
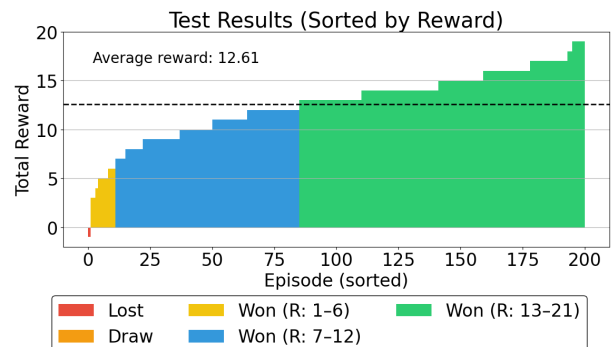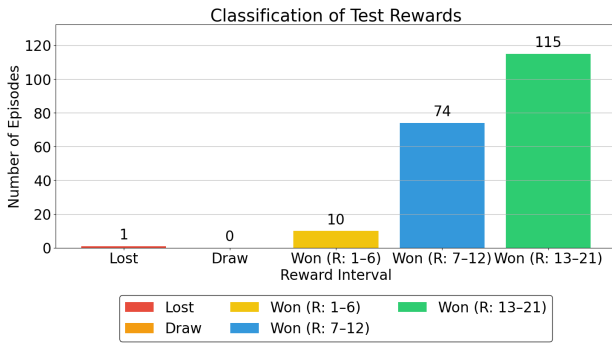

Fig. 13: Testing results, sorted

Fig. 14: Testing results, by reward intervals

6 MULTI-AGENT LEARNING

In MARL, multiple agents learn within a shared enviroment. Unlike traditional single-agent RL where the enviroment is stationary, MARL introduces complexity due to the enviroment being non-stationary. An enviroment is stationary when the transition probabilities and the reward function do not change over time. MARL is non-stationary because the other agents will change their strategies while training, so the current model strategy that works now, may not work in the future when the other/s agents changes his strategy. The agents can be competitive, cooperative or both. When the agents compete between them, they have opposed goals, one agent gain will mean the other agent loss. For example in chess, one player captures a piece and has better chance on winning, making the other player loose chance on winning. When agents cooperate, agents share the same goal and must coordinate their actions to complete the goal. For example, robots working together to move and object. Mixed scenarios combine both: some agents cooperate in teams to compete with other teams of agents. For example, a common multiplayer game Capture the Flag. where the agents cooperate with the team to win the other team.

6.1 The Petting Zoo library

PettingZoo[6] is a Python library designed to standarize enviroments specific for MARL learning. Like Gymnasium, it provides a unified API and a diverse set of enviroments for testing and benchmarking MARL algorithms. Can only be run in Linux or macOS.

6.2 Implementing the MARL algorithm. DQN vs REINFORCE

The enviroment used is PettingZoo Pong[7]. The agents will be competitive. When one agent scores, it penalizes the other agent that takes -1 reward meanwhile the agent that scored gets +1. Unlike the environment used in Gymnasium's Pong implementation, in PettingZoo's Pong environment, if the agent does not serve the ball, it gets penalized: "Serves are timed: If the player does not serve within 2 seconds of receiving the ball, they receive -1 points, and the timer resets. This prevents one player from indefinitely stalling

---

the game, but also means it is no longer a purely zero-sum game"[**?**]. Now instead of implementing my own preprocessing, frame stacking and frame skipping methods, im using pre-built ones from Python library SuperSuit [8]. Using frameskipping and framestacking of 4 like the single-agent Pong. To stabilize the training from both agents, starting on 1M frames and beyond, every 50k frames the training for one of the agents is freezed and the other one enabled. The DQN model is the same implementation as the single-agent but using Petting Zoo enviroment and SuperSuit wrappers. The REINFORCE agent utilizes a neural network with a softmax final layer to generate an action probability distribution. It chooses actions randomly for the first 75,000 steps to promote early exploration. Afterwards, it samples actions from the policy and accumulates log-probabilities and their respective rewards. At the end of an episode, it rewards actions leading to better results and adjusts the network so it is more likely to make such actions again. It also adds a small amount of randomness (entropy) so it can continue exploring and prevent repeating the same actions. This time the model is trained in a local PC using GPU 5070 with 12 GB of VRAM and 32 GB of RAM.

The reinforce algorithm implementation follows the pseudocode in Algorithm 5. The process begins by initializing the policy network with random weights and setting up storage for log-probabilities of actions and the rewards collected during each episode. For every episode, the environment is reset and the agent repeatedly selects actions either randomly (during a pretraining phase) or by passing the current state through the policy network to sample from the resulting probability distribution over actions. After each action, the resulting reward and new state are recorded. Once the episode ends, the algorithm computes the cumulative returns (discounted sum of future rewards) for each time step, normalizes and clips these returns to stabilize training, and calculates a loss by multiplying the negative log-probabilities of the taken actions by their respective returns. This encourages the policy to increase the likelihood of actions that lead to high rewards. An entropy bonus is added to the loss to promote exploration by preventing the policy from becoming overly deterministic too early. The policy network is then updated using gradient descent, and the stored log-probabilities and rewards are cleared in preparation for the next episode.

---

[6] https://pettingzoo.farama.org/
[7] https://pettingzoo.farama.org/environments/atari/pong/

[8] https://github.com/Farama-Foundation/SuperSuit

Algorithm 5 Reinforce algorithm

1: Initialize policy network with random weights
2: Initialize empty lists for log-probabilities and rewards
3: **for** each episode **do**
4:     Reset the environment and get initial state s
5:     **while** episode not finished **do**
6:         **if** still in pretraining (e.g., less than 75000 frames) **then**
7:             Choose a random action a
8:         **else**
9:             Convert state to tensor and pass through policy network
10:             Get action probabilities and sample an action a
11:             Save $\log$(probability of a) to log-prob list
12:         **end if**
13:         Take action a, get reward r and next state s'
14:         Add r to reward list
15:         $s \leftarrow s'$
16:     **end while**
17:     Calculate discounted returns G_t from rewards
18:     Normalize G_t (mean 0, std 1) and clip between $[-2, 2]$
19:     Compute loss: $-\log(\text{prob}) \times G_t$ for each step
20:     Add entropy bonus to encourage exploration
21:     Update policy network using optimizer
22:     Clear log-probabilities and rewards for next episode
23: **end for**

6.2.1 Results and Conclusions from Training and Evaluation

Trained with 8.5M frames. The first 1000000 frames the DQN was mainly exploring because of the epsilon greedy policy that went from 0.99 to 0.1. After that, the models freezed and let the other model train to a freezed model till 5M frames. From 5M frames to 8.5M the models could train together. The rewards, shown in Figure 15 tend to be lower, because at some point the agents think it's good to just not serve the ball, and the enviroment penalizes that.
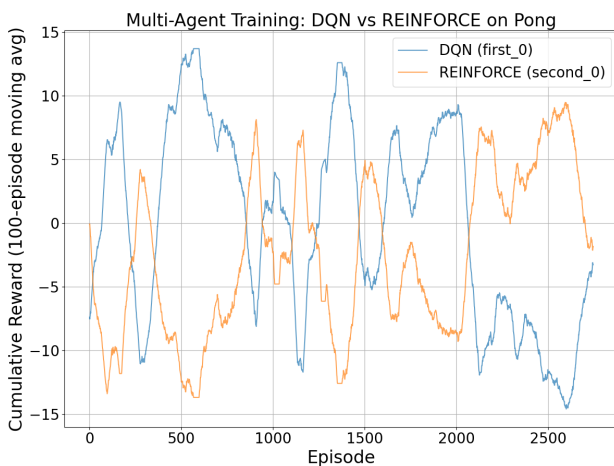

Fig. 15: Training results, by reward intervals

To evaluate, two distinct methods will be used. The first, to see which agent is better, the episode reward is compared (Figure 16). The second (Figure 17), to see if the models are well trained and competitive, the episode lenght in frames will be compared to episode lenght of random action models. It's tested on 200 episodes.
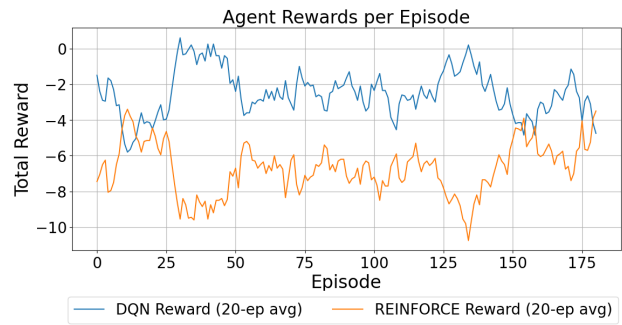

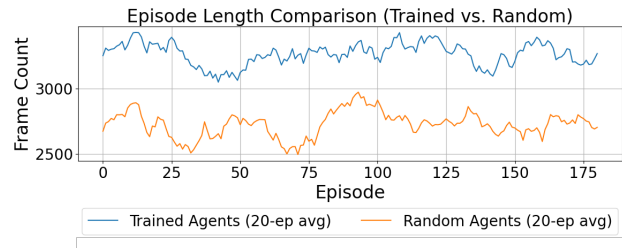Fig. 16: Testing results, DQN vs REINFORCE

Fig. 17: Testing results, trained models vs random agents

7 CONCLUSIONS

7.1 Tabular methods conclusions

The slippery option is a bit experimental and can be confusing. The agent can do the right action but get a bad reward just because it slips. Even a human can fail just because of randomness. The deterministic results are very great and that's what tells us the algorithms are well implemented.

7.2 Aproximate methods conclusions

Maybe if the training was done with more than 500 max steps could get a better result, because the model just learnt to hold the first 500 steps. Even if holding the first 500 steps the cart can tilt just a bit to the side and touch the wall, resulting worse results with more than 500 steps because of an unexplored area. Besides that, the trained model is impressive and can be better in not trained situations (more than 500 steps), as shown in Figure 7.

7.3 Deep Q-Network applied to image based game conclusions

The model is very robust, because it almost always win the machine, and by a lot of points. The result is very good and could be better by having greater resources. The replay buffer couldn't be bigger than 100k because of the limited RAM of Kaggle. Professional implementations like Deep-Mind's one [3] use +1M buffer sizes. And with more hours avaliable, could be trained with more frames.

7.4 Multi-Agent learning conclusions

The results are not what were expected, but it can be justified. Training two models from zero, that both are very bad at first, can led to many problems. The first behaviour

was found is sometimes, the model learns to not serve, and if the model does not explore a lot in the first frames it can kill the training process. Playing vs a bad model leds sometimes to a lazy agent: it gains rewards just by serving the ball because the other agent can't return it. The agent think it's performing well just because the opponent is not performing. Even if the model learns to play without lazyness, when the other model explores or isn't performing, the model can return to be lazy because getting rewards just for serving the ball. A better approach could have been training a model vs the machine, then training the other model with the already trained model so those problems do not happen.

7.5   Final conclusions

Hyperparameter complexity is one of the biggest difficulties when it comes to working with reinforcement learning (RL). Factors such as learning rate, discount factor, and exploration parameters can have a massive influence on whether an agent succeeds or completely fails. It gets worse still due to the behavior of each environment being different. Something that works in a given game or simulation may not in another, so each experiment tends to need manual tweaking and trial and error. Cost of training is another key consideration. Having to train a model for hours and not getting the expected result can be frustrating sometimes, and costing a lot of hours trying to get your desired performance on the model.

REFERÈNCIES

[1] Sutton & Barto (2018). Reinforcement learning: an introduction.

[2] Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Graves, Alex and Antonoglou, Ioannis and Wierstra, Daan and Riedmiller, Martin. Playing Atari with Deep Reinforcement Learning. (2013).

[3] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).

[4] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016).

[5] Bellman, R. (1957). A Markovian decision process. Journal of Mathematics and Mechanics, 6(5), 679–684.

[6] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3), 210–229.

[7] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. Machine Learning, 3, 9–44.

[8] Rummery, G. A., & Niranjan, M. (1994). On-line Q-learning using connectionist systems (Tech. Rep. No. CUED/F-INFENG/TR 166). Cambridge University Engineering Department.

[9] Watkins, C. J. C. H. (1989). Learning from Delayed Rewards (PhD thesis). University of Cambridge.

[10] Williams, Ronald J. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. (1992).

[11] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *arXiv preprint* arXiv:1606.01540.
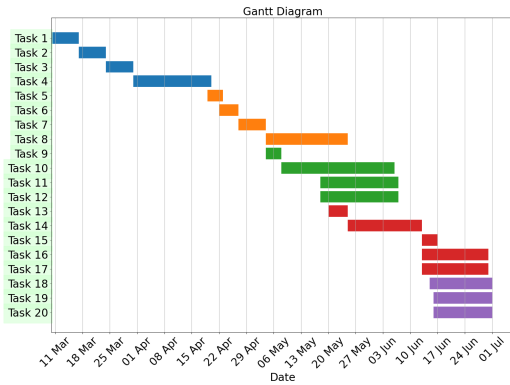
APPENDIX

Gantt diagram

Fig. 18: Grantt diagram

Some DQN algortihm implementations for the SARL image based pong

```python
def preprocess_frame(frame):
    frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
    frame = cv2.resize(frame, (84, 84), interpolation=cv2.INTER_AREA)
    return frame / 255.0  # Normalize to [0, 1]
```

```python
class FrameStack(gym.Wrapper):
    def __init__(self, env, k=4):
        super().__init__(env)
        self.k = k
        self.frames = deque([], maxlen=k)
        self.observation_space = gym.spaces.Box(low=0.0, high=1.0, shape=(k, 84, 84), dtype=np.float32)

    def reset(self, **kwargs):
        obs, info = self.env.reset(**kwargs)
        processed = preprocess_frame(obs)
        for _ in range(self.k):
            self.frames.append(processed)
        return np.array(self.frames, dtype=np.float32), info

    def step(self, action):
        obs, reward, terminated, truncated, info = self.env.step(action)
        processed = preprocess_frame(obs)
        self.frames.append(processed)
        done = terminated or truncated
        return np.array(self.frames, dtype=np.float32), reward, done, info
```

```python
class FrameSkip(gym.Wrapper):
    def __init__(self, env, skip=4):
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        terminated = False
        truncated = False
        info = {}

        for _ in range(self._skip):
            obs, reward, term, trunc, info = self.env.step(action)
            total_reward += reward
            if term or trunc:
                terminated = term
                truncated = trunc
                break

        return obs, total_reward, terminated, truncated, info
```

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def add(self, state, action, reward, next_state, done):
        # Store raw pixels as uint8 to save RAM
        self.buffer.append((
            (state * 255).astype(np.uint8),
            action,
            reward,
            (next_state * 255).astype(np.uint8),
            done
        ))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        # Convert back to float32 and normalize to [0, 1]
        return (
            np.array(states, dtype=np.uint8).astype(np.float32) / 255.0,
            np.array(actions, dtype=np.int64),
            np.array(rewards, dtype=np.float32),
            np.array(next_states, dtype=np.uint8).astype(np.float32) / 255.0,
            np.array(dones, dtype=np.float32),
        )

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

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def add(self, state, action, reward, next_state, done):
        # Store raw pixels as uint8 to save RAM
        self.buffer.append((
            (state * 255).astype(np.uint8),
            action,
            reward,
            (next_state * 255).astype(np.uint8),
            done
        ))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        # Convert back to float32 and normalize to [0, 1]
        return (
            np.array(states, dtype=np.uint8).astype(np.float32) / 255.0,
            np.array(actions, dtype=np.int64),
            np.array(rewards, dtype=np.float32),
            np.array(next_states, dtype=np.uint8).astype(np.float32) / 255.0,
            np.array(dones, dtype=np.float32),
        )

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

```python
def select_action(model, state, epsilon, num_actions):
    if random.random() < epsilon:
        return random.randrange(num_actions)
    state_tensor = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)
    with torch.no_grad():
        q_values = model(state_tensor)
    return q_values.argmax(dim=1).item()
```

```python
def compute_td_loss(batch, policy_net, target_net):
    states, actions, rewards, next_states, dones = batch

    states = torch.tensor(states, dtype=torch.float32, device=device)
    actions = torch.tensor(actions, dtype=torch.int64, device=device)
    rewards = torch.tensor(rewards, dtype=torch.float32, device=device)
    next_states = torch.tensor(next_states, dtype=torch.float32, device=device)
    dones = torch.tensor(dones, dtype=torch.float32, device=device)

    q_values = policy_net(states)
    next_q_values = target_net(next_states)

    q_value = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)
    max_next_q_value = next_q_values.max(1)[0]
    expected_q_value = rewards + GAMMA * max_next_q_value * (1 - dones)

    loss = nn.MSELoss()(q_value, expected_q_value.detach())
    return loss
```

Some REINFORCE algorithm implementations for the MARL image based pong

```python
class REINFORCEPolicy(nn.Module):
    def __init__(self, input_shape, num_actions):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4), nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2), nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(3136, 512), nn.ReLU(),
            nn.Linear(512, num_actions),
            nn.Softmax(dim=-1)
        )

    def forward(self, x):
        return self.net(x)
```

```python
class REINFORCEAgent:
    def __init__(self, input_shape, num_actions, gamma=0.99, lr=1e-3, device='cpu'):
        self.device = device
        self.gamma = gamma
        self.num_actions = num_actions
        self.model = REINFORCEPolicy(input_shape, num_actions).to(device)
        self.optimizer = optim.Adam(self.model.parameters(), lr=lr)
        self.log_probs = []
        self.rewards = []

    def select_action(self, state):
        if len(self.rewards) < 75000:
            return np.random.randint(self.num_actions)
        state = torch.tensor(state, dtype=torch.float32, device=self.device)
        state = state.permute(2, 0, 1).unsqueeze(0)

        probs = self.model(state)

        dist = torch.distributions.Categorical(probs)
        action = dist.sample()
        self.log_probs.append(dist.log_prob(action))
        return action.item()

    def store_reward(self, reward):
        self.rewards.append(reward)

    def update_policy(self):
        if not self.rewards or not self.log_probs:
            return

        G = 0
        returns = []
        for r in reversed(self.rewards):
            G = r + self.gamma * G
            returns.insert(0, G)

        returns = torch.tensor(returns, dtype=torch.float32, device=self.device)
        print(f"REINFORCE mean return (unnormalized): {returns.mean().item():.2f}")
        returns = (returns - returns.mean()) / (returns.std() + 1e-9)
        returns = returns.clamp(-2.0, 2.0)

        loss = 0
        for log_prob, R in zip(self.log_probs, returns):
            loss -= log_prob * R

        probs = torch.stack([lp.exp() for lp in self.log_probs])
        entropy = -probs * torch.stack(self.log_probs)
        entropy = entropy.sum()
        loss -= 0.1 * entropy  # Encourage exploration

        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)
        self.optimizer.step()

        self.log_probs = []
        self.rewards = []
```