Reinforcement Learning Project

Giorgos Tsouderos, Jonas Dixon, Yagna Karthik Vaka, Markus Rupp

May 2024

1 Introduction

In our project, we implemented a Deep Q-Network (DQN) with the goal of training an agent to play the Atari game Pong effectively. A Q-Network is a specific reinforcement learning algorithm that combines traditional reinforcement learning with deep learning and neural networks. It is often used to approximate the Q-values for large or continuous state-spaces as those in atari games. Among other things, a DQN utilizes experience replay and a target network in order to balance the training process. By utilizing raw pixel data from the game as input, we employed convolutional neural networks (CNNs) to process this visual information. We enhanced our DQN with a specialized technique to better handle the unique challenges posed by the nature of the input data.

A significant hurdle in training neural networks with sequential data, such as frames from a game, is the high correlation between consecutive inputs. To address this, we incorporated experience replay, a method designed to break these correlations and stabilize the learning process. We began by testing our approach on the simpler Cart Pole environment before extending it to the more complex Pong environment.

1.1 CartPole

The CartPole environment is a classic problem in the field of reinforcement learning and serves as an ideal starting point for testing and validating new algorithms. This environment is provided by OpenAI's Gym, a toolkit designed for developing and comparing reinforcement learning algorithms.

In the CartPole environment, an agent interacts with a system consisting of a pole attached by an un-actuated joint to a cart, which moves along a frictionless track. The agent's goal is to prevent the pole from falling over by applying a force of +1 or -1 to the cart. The state space is represented by four continuous variables:

1. Cart Position: The position of the cart on the track.
2. Cart Velocity: The velocity of the cart.
3. Pole Angle: The angle of the pole with the vertical.
4. Pole Velocity at Tip: The velocity of the tip of the pole.

The agent receives a reward of +1 for every time step the pole remains upright, and the episode ends if the pole falls past a certain angle or the cart moves too far from the center. The objective is to maximize the total reward, which corresponds to keeping the pole balanced for as long as possible.

1.2 Pong

The Pong environment, based on the classic Atari game, is a more complex and visually rich task compared to the CartPole environment. In this game, the agent controls a paddle on one side of the screen, and the objective is to hit the ball back and forth with an opponent paddle controlled by the environment. The goal of the agent is to win the game by scoring more points than the opponent.

The state space in Pong consists of raw pixel data from the game screen, typically represented as a 210x160x3 RGB image, which captures the entire visual scene. This high-dimensional input space poses significant challenges for reinforcement learning algorithms.

The agent receives a reward of +1 each time it successfully gets the ball past the opponent's paddle,

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 1: Algorithm DQN, source: [2]

scoring a point. Conversely, the agent receives a reward of -1 each time the ball gets past its own paddle, resulting in the opponent scoring a point. Neutral Rewards: During most frames, when neither player scores, the reward is 0.

2 Implementation

2.1 Algorithm

Q-learning is a model-free reinforcement learning algorithm that aims to learn the value of the optimal action-selection policy. The core idea is to iteratively update the Q-value function, $Q(s, a)$, which represents the expected cumulative reward of taking action a in state s and following the optimal policy thereafter.

The idea is minimzing the loss function with stochastic gradient descent. We utilized a technique known as experience replay to store the agent's experiences at each time step proposed in the paper "Playing Atari with Deep Reinforcement" by Deep Mind Technologies [1]. Learning Each experience, denoted as $e_t = (s_t, a_t, r_t, s_{t+1})$, consists of the current state s_t , the action taken a_t , the reward received r_t , and the next state s_{t+1} . These experiences are collected into a dataset $D = \{e_1, e_2, ..., e_N\}$ within a replay memory that spans multiple episodes.

During the algorithm's inner loop, we perform Q-learning updates, specifically minibatch updates, on randomly sampled experiences from the replay memory. This random sampling from the pool of stored experiences helps mitigate the issue of correlated inputs. After updating the Q-values through experience replay, the agent selects and executes an action based on a given policy, balancing exploration and exploitation.

For evaluation, we utilized a function that executes 1000 episodes, calculating the average total returns across these runs to assess the performance of the trained agent. This rigorous testing ensures that our DQN implementation not only learns to play Pong but does so effectively and consistently.

2.2 CartPole

For the CartPole environment, an epsilon greedy policy is implemented. This policy is a simple exploration strategy that utilizes a parameter ϵ. The agent will act randomly with probability ϵ and greedily with probability $1 - \epsilon$. Additionally, ϵ will decay as training progresses. This encourages the agent to explore the state-space and all actions early on in the training process and then act more greedily towards the end of training when the environment has been learned. Key benefits of this policy is its simplicity, that it allows for direct control of the exploration ratio and that it usually is effective for many reinforcement learning tasks. In our case, epsilon is decayed by a factor of 0.995 after each action taken by the agent.

2.3 Pong

To allow for the Pong environment to be learned by our model, several changes were required. First, we implemented preprocessing using the `AtariPreprocessing` function from `gymnasium.wrappers`. Additionally, since no information regarding the velocity of the ball can be gathered from one frame, multiple frames must be passed to the DQN. This was done by stacking the last four frames in each tensor passed to the DQN. Additionally, to deal with the visual data, a CNN architecture was implemented for the DQN. This architecture is based on the one described in the Nature DQN paper provided in the project instructions.

The first implemented model utilized an epsilon greedy policy similarly to the one described above for the CartPole environment. However, instead of decaying ϵ by a factor of 0.995 after each action, epsilon was decayed linearly starting at 1 and ending at 0.01 with an anneal length of $8 * 10^5$. However, due to wanting to improve the performance of the model further, we decided to try another method.

For our second model, we implemented the technique of noisy networks as suggested by Meire Fortunato et al. [3]. We introduced noisy layers into the DQN whose weights and biases are perturbed by a parametric function of noise. During training, these parameters are adapted using gradient descent. This method allows the network to explore without the need of epsilon since stochasticity is induced to the agent's policy.

2.4 Challenges

One challenge that we faced during the implementation of the DQN on both the CartPole and Pong environments was that of tensor dimension issues. This was not a large issue, and was relatively easy to overcome, however it was present during most of the steps. These types of issues were commonly due to tensors having one too many or few dimensions and were solved by either squeezing out the extra dimension or keeping the dimensions at various steps.

Another challenge that we faced was that of convergence for the epsilon-greedy DQN in the Pong environment. Initially, the model would not converge and the returns would ossilate around -20 and -15. However, this issue was overcome by setting and changing the amount that epsilon decayed over the training process. The process of tuning the epsilon hyperparameter is discussed more in the following section.

2.5 Hyperparameter Tuning

For the hypermarameter tuning of our model we started by choosing the proposed values given to us from the instructions. However, we didn't manage to reach the desired rewards and we continued further with tuning. Specifically, we experimented thoroughly with the epsilon values (epsilon and epsilon decay) since they play an important role in balancing the exploration and exploitation of the learnt policy by the agent. Originally, we steadily decreased the epsilon value by a certain amount, regardless of the number of epochs we had chosen. From observing our agent, that lead either to exploiting the current learnt policy before enough exploration was done or the opposite, resulting in poor rewards. That prompted us to decay epsilon by a constant amount which was dependent by the amount of epochs our model would run, until it reached the value 0. We then proceeded with running the model over various amounts epochs since we needed to find a number that allowed the agent to explore sufficiently before epsilon decayed enough to force the exploitation of the learnt policy. Since the training could take several hours, finding the correct hyperparameters proved to be a demanding task. As a result, we found an alternative model which included noisy layers in the neural network, working as a bypass for the epsilon tuning. Both versions of the model though, reached the desired rewards in the end.

3 Results

3.1 CartPole

As described above, the DQN network has been trained and evaluated on the CartPole-V1 environment. Below, in figure 2, the results from this training process in visualized. The training consisted of 200 evaluations. Plotted below is the return for each of these 200 episodes as well as the trailing average computed with a window size of 20.

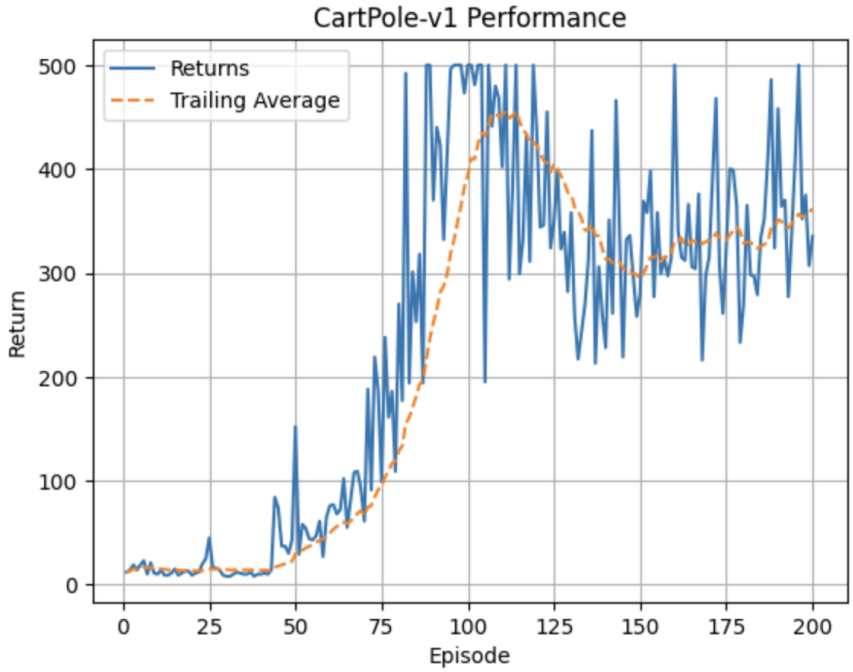

Figure 2: CartPole results

As can be seen, the performance peaks after around 100 iterations at a value of about 450. Then, it drops off and stabilizes between episodes 150 and 200 to a return of about 350. This behavior indicates that the model is being trained for too long and starts to overfit after about 100 episodes. To account for this, early stopping after 100 episodes could be implemented to ensure that the final model receives the best performance.

3.2 Pong

Below, in figures 3 and 4, the results for the epsilon-greedy and noisy-net DQN models are presented. As can be seen, the final returns of the epsilon-greedy model land around 13, whilst the returns of the noisy-net model land at around 10. It is worth noting that the epsilon-greedy model was trained for 1000 episodes while the noisy-net was trained for only 250. Through, testing we observed that the noisy-net plateaued at around this episode, whilst the epsilon-greedy models performance improved for many more episodes. Due to GPU limitations, we were unable to train the epsilon-greedy network for more than 1000 episodes.

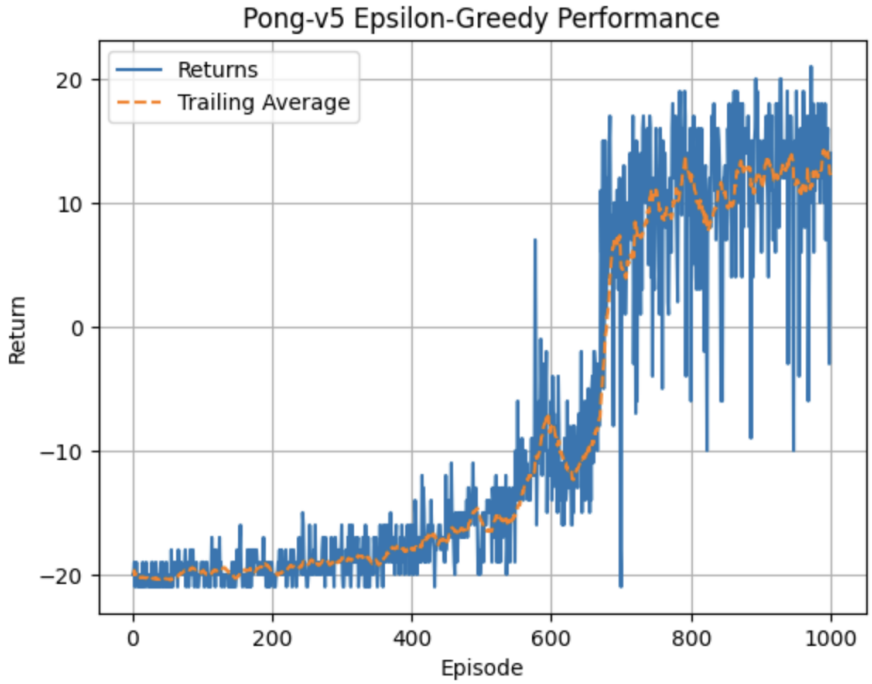

Figure 3: Epsilon-Greedy Pong results

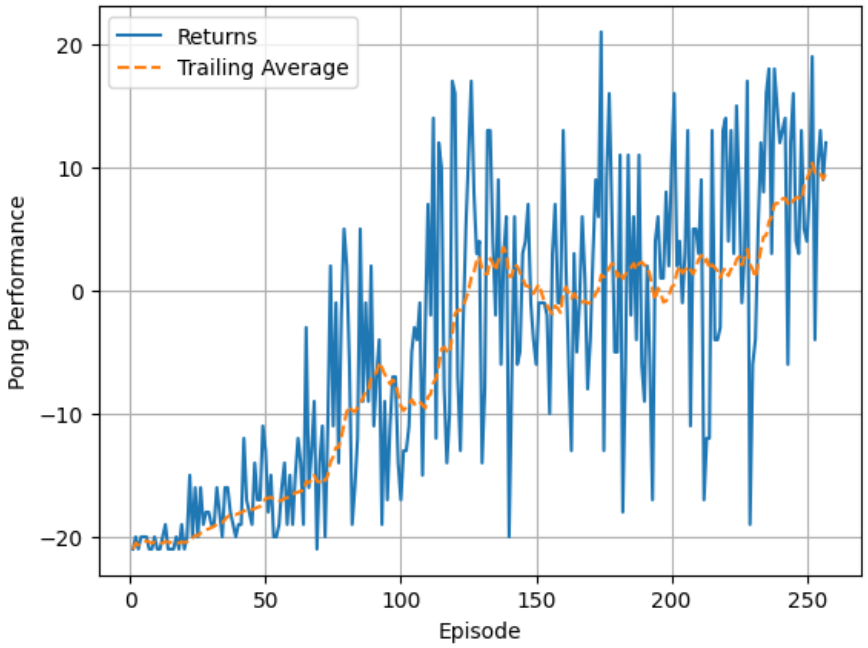

Figure 4: Noisy-Net Pong results

4 Discussion

Deep Q-networks (DQN) have been used in this project to solve the CartPole-v1 and ALE/Pong-v5 environments with the gymnasium AtariPreprocessing for effective frame preprocessing. AtariPreprocessing module offers a streamlined solution for frame processing and resizing frames of 84x84 pixels, converting them grayscale to reduce the complexity, and rescaling pixel values to the range [0,1]. The maximum number of no-operations actions at the beginning of each episode is retained at 30 during gameplay, and frame skipping is allowed as well. DQN and their impact on reinforcement learning in gaming environments is quite innovative and we explored them in terms of Experience replay, target network and mean

squared error (MSE). Storing transitions and replay are experienced by DQN in a memory buffer. The agent is allowed with this method to learn from previous transitions or replays and could perform better with temporal correlation i.e which are delayed rewards. we also updated the target network to stabilize the training which can lead to better convergence.

4.1 CartPole

A fundamental reinforcement learning task is the CartPole-v1 environment for training DQN. In this environment, we have 2 actions which are left and right to maintain the balance of the pole on the cart. Through a series of training sessions, our agent has learned a policy that maximizes a cumulative reward and that means the model has learned to maintain the pole upright on the cart. Even, we also modified some hyperparameters and experimented with them to achieve optimal performance. With these tuning parameters, we improved the agent's efficiency in learning and convergence.

4.2 Pong

ALE/Pong-v5 has a lot more complexity and challenges than CartPole-v1 while Training DQN agents. In this environment, temporal dynamics necessitate the integration of convolutional neural networks and the stacking of frames to capture the patterns in the game environment. We have reduced the dimensions and extracted some relevant features from the raw data with grayscale conversion and frame stacking. Finally, we have larger state spaces and require the agent to learn more about the actions of the game environment. We have also used NoisyLinear layers in the pong environment because it has many advantages like enhanced exploration, leading to better and faster learning and the ability to capture complex actions.

References

[1] David Silver Alex Graves Ioannis Antonoglou Daan Wiestra Martin Riedmiller Volodymyr Mnih, Koray Kavukuoglu. Playing atari with deep reinforcement learning. 1994.

[2] David Silver 1 * Andrei A. Rusu1 Joel Veness 1 Marc G. Bellemare1 Alex Graves 1 Martin Riedmiller 1 Andreas K. Fidjeland1 Georg Ostrovski 1 Stig Petersen1 Charles Beattie1 Amir Sadik1 Ioannis Antonoglou1 Helen King1 Dharshan Kumaran1 Daan Wierstra1 Shane Legg1 Demis Hassabis Volodymyr Mnih1 *, Koray Kavukcuoglu1 *. Human-level control through deep reinforcement learning. 2015.

[3] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks for Exploration, July 2019. arXiv:1706.10295 [cs, stat].