Learning to play Pong with DQN

1RT747

Project Report by Group 9

May 11, 2024

Group members:

Chen Gu	chen.gu.3340@student.uu.se
Seyedehmoniba Ravan	moniba.ravan.3181@student.uu.se
André Ramos Ekengren	andre.ramosekengren.3051@student.uu.se
Nora Derner	nora.derner.8232@student.uu.se

1. Deep Q-Network

Deep Q-Learning, introduced by Mnih et al. in the papers [1] and [2], is an advanced and robust method in the field of reinforcement learning, based on the concept of Markov Decision Processes (MDP). It utilizes the model-free Q-learning method, in which the agent learns the optimal strategy to maximize the discounted sum of future rewards, based on the Bellman Equation. In contrast to traditional Q-learning, which uses a Q-table, DQN employs a neural network to approximate the Q-value function, enabling it to map more complex state-action spaces to optimal Q-values while ensuring effective usage of memory and computational resources.

To create a balance between exploration and exploitation, DQN uses the Epsilon-Greedy strategy. Initially, the agent performs random actions, but as experience accumulates, exploration is reduced, and the agent begins to utilize more of its learned strategy. Another significant component of DQN is the use of a replay memory buffer, where the agent's experiences during training are stored. These experiences are used in a randomly selected batch to train the neural network in an off-policy manner. To further stabilize the learning process, a second target network is used, which is updated at fixed intervals with the weights of the main network. The main network estimates the Q-values based on the current state and the action performed, while the target network provides the expected Q-values for the next state. The discrepancy between the Q-value predicted by the main network and the Q-value estimated by the target network for the next state, adjusted by the reward received, is known as Temporal Difference. This difference is used to calculate the loss and effectively train the main neural network. [1–3]

2. Step 1 - Implement and test DQN - CartPole-v1

In this chapter, we present the training results of our DQN agent in the Cartpole-v1 environment. The environment provides observations as an array of four floating point numbers describing the position and velocity of the cart as well as the angular position and angular velocity of the pole. The reward is a scalar value of $+1$ for each time step in which the pole remains upright. The maximum possible total reward for an episode is 500. The cart can perform two actions: Movement to the right or left.

2.1. Results of Default Configuration

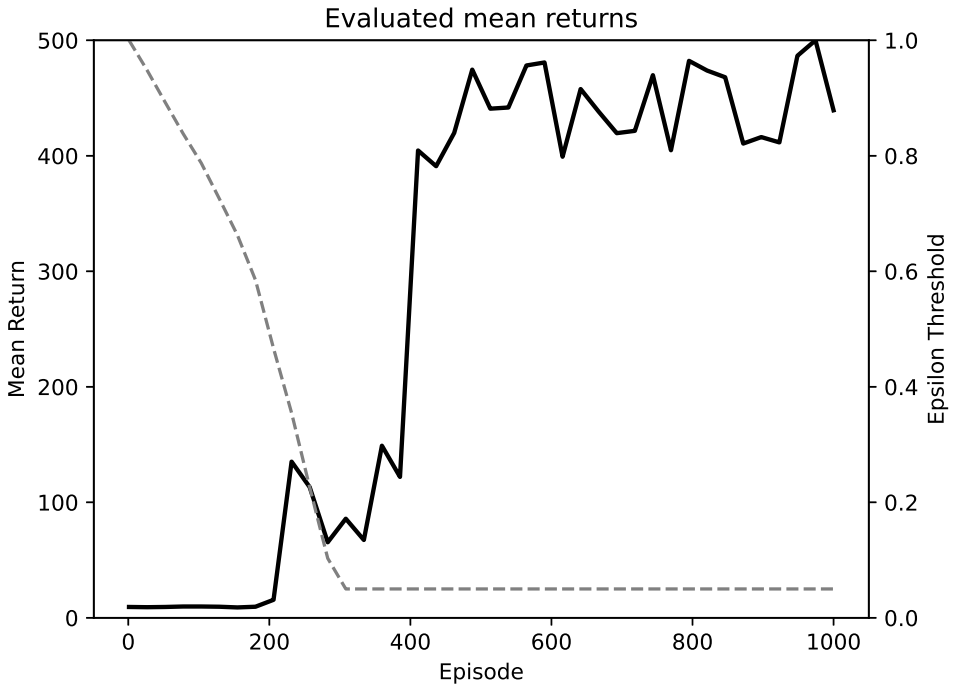


Figure 1: Default Configuration: Mean Return and Epsilon Threshold Development Across 1000 Episodes

Our final DQN agent was configured using the default hyperparameters shown in the first column of table 1. The mean return of our agent stabilises between the values 400 and 500 over 1000 episodes, as shown in Figure 1. This indicates that our agent has successfully learnt to balance the pole over a longer period of time. The mean return shows a significant increase between episodes 200 and 410, indicating that the agent is beginning to employ more effective strategies at this point. This rapid improvement likely results from both the gradual reduction of the epsilon threshold value and the continuous optimization of the network's weights through accumulated training data. From episode 300 onwards, the epsilon value stabilises at a minimum of 0.05 (eps_end). This effective use of the epsilon-greedy strategy ensures that while the agent continues to exploit its learned strategies for optimal action 95% of the time, the remaining 5% dedicated to exploration still provides opportunities for further optimization and refinement, securing robust performance improvements throughout the training.

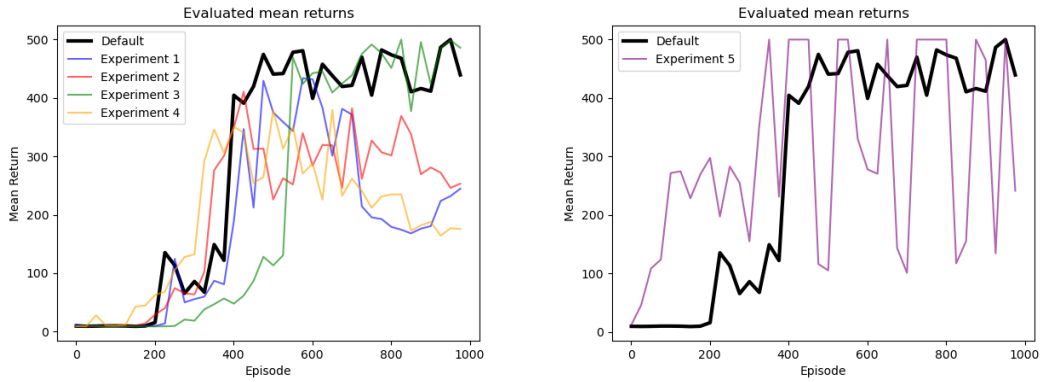
Parameter	**Default**	**Exp1**	**Exp2**	**Exp3**	**Exp4**	**Exp5**
memory_size	50000	50000	50000	50000	50000	50000
n_episodes	1000	1000	1000	1000	1000	1000
batch_size	32	32	64	32	32	32
target_update_frequency	100	100	100	10	1000	100
train_frequency	1	1	1	1	1	1
gamma	0.95	0.95	0.95	0.95	0.95	0.95
lr	0.0001	0.0001	0.0001	0.0001	0.0001	0.001
eps_start	1.0	1.0	1.0	1.0	1.0	1.0
eps_end	0.05	0.01	0.05	0.05	0.05	0.05
anneal_length	10000	10000	10000	10000	10000	10000
n_actions	2	2	2	2	2	2
final_return	439.4	244.6	253.2	485.8	175.6	241.2
best_return	500.0	433.8	410.8	499.0	379.6	500.0

Table 1: Default Configuration vs. Five Experimental Setups: Hyperparameter Configuration and Final Mean Return

2.2. Hyperparameter Experiments

In the following, we have carried out 5 experiments in which we have altered individual hyperparameter values separately while keeping the remaining parameters at their default settings. An overview of the hyperparameters used and the corresponding final mean return value after 1000 episodes is provided in the table 1. The results are shown in Figure 2.

Each experiment was conducted multiple times to ensure the reliability of our findings. However, for clarity and simplicity in our presentation, we have chosen to present only one representative result per experiment. We acknowledge that multiple iterations could yield varied results due to the inherent variability in the training processes of DQN, even when using the same hyperparameters. This variability is primarily introduced by the selection of actions using an epsilon-greedy strategy and the sequence in which experiences are sampled from the replay buffer. Nonetheless, our experiments exploring various hyperparameter configurations provide a foundational understanding of how these adjustments impact performance of our DQN agent.



(a) Default vs. EXP1 to EXP4

(b) Default vs. EXP5

Figure 2: Default Configuration vs. Five Experimental Setups: Mean Return Across 1000 Episodes

Experiment 1: eps_end = 0.01
In the first experiment, we set the epsilon end value from 0.05 to 0.01 with the same decay rate. This adjustment means that from episode 300 onward, the agent only explores the environment randomly with a 1% probability. Like the default configuration, the average mean return value rises rapidly from episode 375 to 400, then fluctuates within the 300 to 400 range. The value falls steeply around episode 700 and remains at a mean return value of around 200. It is not clear whether the change in the hyperparameter had a huge effect on the training. Since the epsilon is smaller in this case, less exploration will be done. It is possible that with less exploration, the model will take longer to learn if it gets stuck making sub-optimal decisions.

Experiment 2: batch_size = 64
Increasing the batch size from 32 to 64 means that twice as many experiences are drawn from the replay buffer for each training update, which contributes to reducing the variance and stabilizing the training process. At the same time this also results in less drastic weight updates, potentially slowing down the model's convergence compared to the default setting. As observed in Figure 2a we can recognise this tendency to some extent. There are fewer fluctuations and the learning curve appears smoother and more stable. To draw more definitive conclusions about the impact of doubling the batch size, extending the training duration may be necessary.

Experiment 3: target_update_frequency = 10
The mean return increased smoothly during the first 550 episodes, and with less frequency compared to the benchmark. Between episodes 600 and 900, the increase in mean return occurred less frequently than in the benchmark. However, in the

final 100 episodes, it exhibited a higher mean return compared to the default model. The $target\_update\_frequency$ serves as a method of stabilization. By keeping the target network fixed for a predetermined number of steps, it regulates the frequency at which the stable target, used to train the primary network, is updated with new values. When the $target\_update\_frequency$ is reduced from 100 to 10, it means that the target network's weights are updated more frequently using the most recent weights from the evaluation network. This increased frequency of replication between the target and evaluation networks can improve the learning process, resulting in higher mean return values. As a result, the model's overall performance is enhanced.

Experiment 4: target_update_frequency = 1000
The average mean return exhibits a steady upward trend over the initial 1000 episodes. We can see from the graph2a: notably, the rise in mean return between episodes 300 and 500 is marked, although it progresses at a slower rate compared to the benchmark model. After episode 400, the mean return fluctuates within a range of 350 to 800. In the last 200 episodes, there is a downward trend, in contrast to the benchmark, where the convergence mean return significantly dropped from above 400 to the 200+ range. We change the $target\_update\_frequency$ from 100 to 1000 when the overall length of episodes equals 1000. This means the target network would only update its weights once at the end of the training episodes. This can result in sub-optimal learning, in which the policy network fails to converge to the optimal policy because it is learning from out-of-date Q values. This also limits the agent's ability to fine-tune its strategy based on the most recent understanding of the environment, perhaps resulting in a incorrect strategy reinforcement.

Experiment 5: lr = 0.001
In Experiment 5 the learning rate was increased from 0.0001 to 0.001. This resulted in a final evaluated mean return of 241.2. From the graph 2b of mean returns we see that there is a lot of fluctuation in the results, hinting that the learning of the agent may not converge. The learning rate is a hyperparameter that signifies the step size taken in the chosen minimizing direction of the loss function. If the learning rate is too big, local minimum might be missed. If it is too small, the convergence towards a local minimum will be too slow. In this case, as mentioned, the learning rate was increased, leading to fluctuation in evaluated mean returns over time. It is possible that this is caused by the learning rate being too large, such that it repeatedly misses local minima.

2.3. Conclusion - CartPole-v1

The systematic examination of various hyperparameters and their influence on the performance of our DQN agent in the Cartpole-v1 environment has provided

important insights for fine-tuning the DQN training procedures. The default configuration already provided satisfactory results, but experimenting with variables such as the final epsilon value, batch size, target update frequency, and learning rate revealed significant differences in training dynamics and outcomes. Specifically, increasing the batch size led to more stable training at the expense of slower convergence, while more frequent updates of the target network tended to improve performance. A high learning rate led to instability in the learning process. Armed with these insights, we are now prepared to apply our refined DQN agent to the next testing environment, ALE/Pong-v5.

3. Step 2 - Learn to play Pong

In this chapter, we explore the training outcomes of our agent in the ALE/Pong-v5 environment. The agent's observations are represented by a stack of four consecutive frames, which are essential for perceiving motion and accurately predicting the ball's trajectory. The agent receives a score of +1 for every point scored against the opponent, and -1 for every point scored by the opponent. The game continues until one player reaches a score of 21. In this environment, the agent has three possible actions: move the paddle up, move it down or no movement.

3.1. Results of Default Configuration

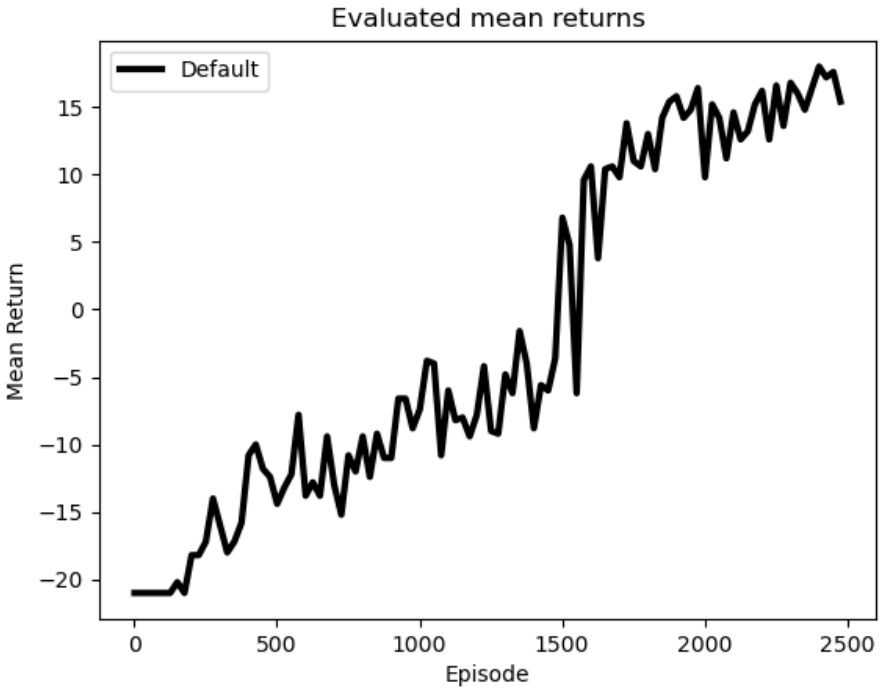


Figure 3: Default Configuration: Mean Return Across 2500 Episodes

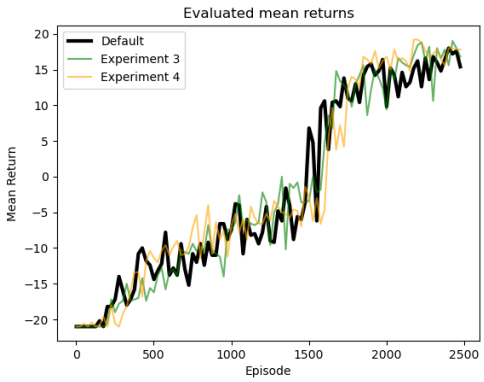
Our final DQN agent was configured using the default hyperparameters shown in the second column of table 2. Inspecting figure 3, we see that the default agent gradually improved at the game, settling in some range around, or above, +15 mean return, meaning that on average the agent was winning over its opponent with quite a large gap in points. During the training, however, a long time was spent losing before a sudden rapid increase in performance around episode 1500. This sudden increase could be because of the epsilon decaying, with a lot of exploration in the beginning of training, while continuously converting to a more exploitative policy. We will see later that tuning the hyperparameters can, for example, lead to a much shorter amount of training episode wise, before the agent starts to win.

Parameter	**Def**	**Exp1**	**Exp2**	**Exp3**	**Exp4**	**Exp5**	**Exp6**	**Exp7**
memory_size	10000	10000	10000	100000	100000	10000	10000	10000
n_episodes	1000	2500	2500	2500	2500	3000	2500	2500
batch_size	32	32	32	32	32	32	32	32
tar_upd_fre	1000	4000	1000	4000	8000	1000	1000	100
train_fre	4	4	4	4	4	4	4	4
gamma	0.99	0.99	0.95	0.99	0.99	0.99	0.95	0.95
lr	0.0001	0.00009	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
eps_start	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
eps_end	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
ann_length	10**6	10**6	10**6	10**6	10**6	10**4	10**6	10**6
n_actions	2	2	2	2	2	2	3	3
final_return	15.4	12.4	18.2	16.8	17.8	17.6	17.2	17.6
best_return	18	17	18.6	19	19.2	19.6	18.6	19

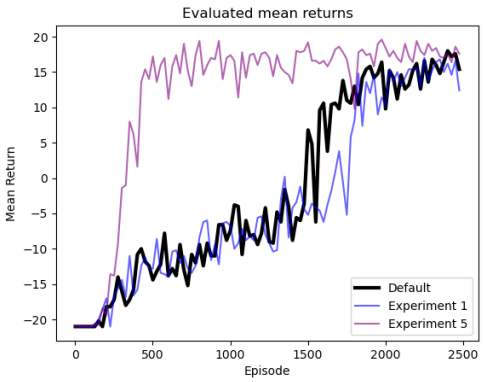
Table 2: Default Pong Configuration vs. Seven Experimental Setups: Hyperparameter Configuration and Final Mean Return

3.2. Hyperparameter Experiments

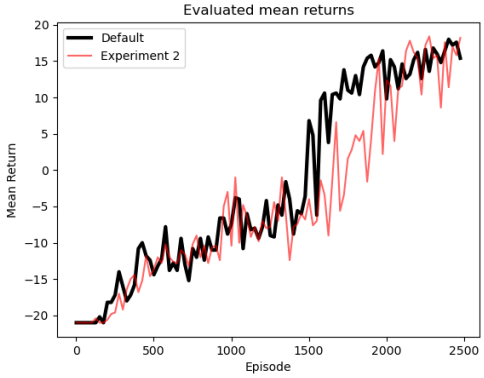
In the following, we performed 7 experiments in which we changed individual hyperparameter values separately and in combination, while leaving the other parameters at their default settings. At the end, based on the model that performed best, we continued the training on that model with fine-tuned parameters. An overview of the hyperparameters used and the corresponding final mean and best mean return value after 2500 episodes can be found in the table 2. The results are shown in Figure 4.



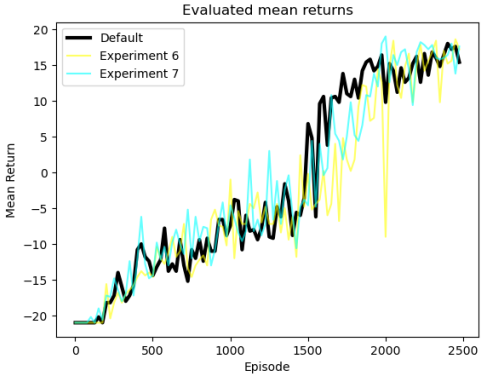
(a) Default vs. EXP3 to EXP4



(b) Default vs. EXP1 and EXP5



(c) Default vs. EXP2



(d) Default vs. EXP6 and EXP7

Figure 4: Default Configuration vs. Seven Experimental Setups: Mean Return Across 2500 Episodes

Experiment 1: lr = 0.0009 , target_update_frequency = 4000
Reducing the learning rate results in less adjustments being made to the weight parameters during updates. This often leads to slower convergence, which we can also see from the plot 4b compared to the default, especially between episodes 1500 and 2000. Here, we observe that the default model achieves improved return values earlier than the model with the lower learning rate. Additionally, the target update frequency was increased to 4000 in this model, further contributing to slower convergence as the target weights are updated less frequently.

Experiment 2: gamma = 0.95
Compared to the benchmark, the performance of Experiment 2, as shown in Figure 4c, closely mirrors the default in terms of the rate of increase in mean returns from episodes 0 to 1500; both curves show a similar trajectory. However, after episode 1500, noticeable differences emerge. The Experiment 2 curve demonstrates a less stable and consistent increase in mean returns. The parameter γ determines how much future rewards are discounted when calculating the current value or

an action. [3] We changed the $\gamma = 0.99$ to $\gamma = 0.95$, and with a lower γ, means that future rewards are discounted more heavily, making the immediate rewards relatively more important, and as a result, the agent with lower γ focuses on achieving higher rewards sooner, rather than taking longer to possibly find the best long-term strategies. And the results appear to be improving, with the best return increasing from 17 to 18.6.

Experiment 3: Memory_size = 100.000, tar_up_freq= 4000
In this experiment, both the replay memory size and the target update frequency was updated. The replay memory was increased tenfold compared to the default, to 100 000, and the target update frequency was increased to 4000. Since random sampling batches from the replay memory is a way to reduce correlations between data points in the training, having a larger memory size might reduce the correlation in the batches even more. In a similar way, updating the target network less frequently may reduce the correlations between the approximated action-values and the target values, stabilizing the training. At the end of training the mean return was 16.8, with the best mean return during training being 19.0.

Experiment 4: Memory_size = 100.000, tar_up_freq= 8000
Experiment 4 differs only from experiment 3 in that the target update frequency was increased even further, to 8000. This was mainly tested because in the Nature paper [2] the target is updated every C updates of the weights, while in our code it is updated every action taken once training has started, and thus updated more frequently. Therefore, we increased it even further to see if there would be a difference. In the end, the mean return was 17.8, while the best mean return during the training was 19.2. Looking at figure 4a, the changes to these hyperparameters do not appear to have had a huge impact on either experiment 3 or 4.

Experiment 5: anneal_length=10000
Compared to the benchmark, the mean return began to increase during the initial 200 episodes, and between episodes 500 and 2500, the mean return exhibited greater stability than observed in the benchmark 4b. Additionally, the mean return increased from 18 to 19.6. The *anneal_length* represents the number of steps over which epsilon should linearly decrease from, and the decay rate is calculated with the formula:

$$\text{decay\_rate} = \frac{\text{eps\_start} - \text{eps\_end}}{\text{anneal\_length}} \tag{1}$$

Originally, with $anneal\_length = 10^6$, the decay rate is decay_rate $= \frac{1.0-0.1}{10^6} = 0.0000009$ per steps, but with the new $anneal\_length = 10^4$, the decay rate is changed to decay_rate $= \frac{1.0-0.1}{10^4} = 0.00009$ per step, which is 100 times faster. By raising the decay rate, *eps_threshold* achieves the *eps_end* value considerably earlier in the training process. Reaching exploitation faster can speed up training,

so the agent starts exploiting its learnt information sooner rather than exploring. However, there is a risk that the agent will not explore enough various methods; hence, when tweaking this hyper-parameter, we should consider a good balance of exploration and exploitation.

Experiment 6: gamma=0.95, n_actions = 3
Compared to the benchmark, Experiment 6 as shown in Figure 4d follows a similar trend but exhibits more fluctuations. We used the default configuration here, altering only the number of operations from two to three. Increasing the amount of actions may improve the agent's capacity to make better decisions, better mimic human behavior, provide finer control over the game environment, and allow greater strategic depth in reaction strategies. But since we discovered that the mean return did not change until 800 episodes, at which point we modified the gamma value and obtained the better result. The $mean\_return$ value increased from 18 to 18.6

Experiment 7: tar_up_freq= 100, gamma=0.95, n_actions = 3
The only difference between Experiment 7 and Experiment 6 is that we introduced another hyper-parameter, which is the $target\_update\_frequency$. We reduced it from 1000 to 100, which means we update the target network more frequently. In a sense, while this may decrease the stability of the model, it can increase the learning speed of the model. Therefore, our final total return also increased from 18 to 19.

Fine-Tuning of a Pre-Trained DQN Model We employed our best model (Experiment 7) and then tried to enhance its performance by fine-tuning the parameters. We reduced the learning rate to 1e-6 to refine the adjustments made during each optimization step, increased the batch size to 64 to gather more information per learning instance, and decreased the epsilon start value from 1 to 0.05 to primarily utilize the strategies we had already learned and optimize them further. This resulted in a very good average return value of 19.8, underscoring the effectiveness of our adjustments.

3.3. Discussion/Conclusion Implementation - ALE/Pong-v5

Our initial configuration proved to be robust, and increasing the memory size and changing the target update frequency did not result in any significant changes. The most significant difference came from reducing the anneal length to 10000, which allowed the agent to achieve better results much faster, as can be seen in Figure 4b. A reduction of Gamma, especially in combination with the increase of possible actions from 2 to 3 improved the performance. Our experiments were limited to 2500 episodes, thus we did not account for convergence behaviors beyond this point. In our evaluation of the trained model from Experiment 7, the finished

agent was able to defeat the computer opponent 21:0. The agent exhibited less jittery behaviour, as we also used the "no movement" action. A video and the entire developed code are attached.

3.4. Discussion - Using DQN for training a Pong-playing agent

The application of Deep Q-Network (DQN) to training an agent for playing Pong has proven highly effective in enhancing the agent's decision-making capabilities within the game's dynamic environment. The architecture of the DQN allows it to analyze the actions and states of the game, closely resembling human-like perception, and supports the learning of suitable responses. Our training and validation results have been very satisfactory, as elaborated in the report. Through systematic hyperparameter tuning, specifically using a shorter *anneal_length* we have successfully improved our agent's performance over time. This adjustment ensures that epsilon reaches its end value more quickly, enhancing the agent's transition from exploration to exploitation. One of our disadvantages of our DQN implementation is that it requires a lot of time to train and compute resources.

In future projects, it would be interesting to compare the performance of our DQN agent with other algorithms. We are particularly interested in exploring variants such as Double DQN and Dueling DQN. Additionally, testing our agent in more complex environments would provide deeper insights. Transitioning to other Atari games, such as Breakout, could reveal how well the strategies learned from one game can be applied to another and assess our agent's robustness against various challenges.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:205242740

[3] R. Sutton and A. Barto, *Reinforcement Learning, second edition: An Introduction*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2018.