Playing Atari with Deep Reinforcement Learning

Daniel Mathews

University of Colorado Boulder
M.S Aerospace Engineering
dama7862@colorado.edu

May 8, 2023

***Abstract*—In this paper, I present an implementation and investigation of a deep reinforcement learning algorithm for training an artificial agent to play the classic Atari game, Pong. The primary goal of this paper is to investigate various strategies for improving the performance and efficiency of the Deep Q-Network algorithm in the context of Pong gameplay. This algorithm combines the common Q-learning with a deep convolutional neural network. I will also focus on the exploration of techniques such as double DQN, prioritized experience replay, and tree-based planning for more challenging and visual domains. With reliance on convolutional neural networks and tuning of hyperparameters, DQN is able to consistently achieve a winning score, showing effectiveness in DQN-based algorithms for reinforcement learning tasks beyond traditional Atari games such as Pong. The network architecture developed allows for generalization to any type of high-dimensional sensory input since it's input comes directly from raw pixel data where optimal policies may be extracted.**

***Index Terms*—Reinforcement Learning, Deep Q-Network**

I. Introduction

The field of artificial intelligence (AI) has experienced a dramatic shift in recent years, with advancements in machine learning and particularly in deep learning opening new horizons. One area of AI, known as reinforcement learning (RL), has shown remarkable progress, fostering the development of decision-making models that learn optimal strategies in complex, interactive environments. This research paper aims to delve into the application of one such algorithm, the Deep Q-Learning Network (DQN), particularly in its ability to master Atari games, which are well-known benchmarks in the field of reinforcement learning.

The DQN algorithm, first introduced by the team at DeepMind in 2013 [13], was a breakthrough in the field of RL. It demonstrated for the first time that a neural network-based model could learn to play Atari 2600 games directly from raw pixels, and other types of environments with high-dimensional states alleviating the curse of dimensionality. The algorithm combines the strengths of deep learning's ability to handle high-dimensional inputs with Q-Learning's capacity to handle sequential decision-making processes, thereby revolutionizing the way intelligent agents are developed.

The application of DQN in Atari games is of particular interest due to the overhead simplicity, deterministic action spaces, and diversity of these games. Each game presents a unique challenge and requires different strategies, making them an excellent benchmark for evaluating the versatility and robustness of RL algorithms. Furthermore, since the input to the DQN is the raw pixel data from the game, it requires no prior knowledge about the game's rules or structure, reinforcing its ability to learn from scratch, an essential attribute for general artificial intelligence.

In this paper, I will present my research on developing a DQN algorithm to play Atari Pong. I investigate various strategies to improve the performance of the DQN, analyze the challenges encountered, and discuss potential solutions.

II. Background and Related Work

The DeepMind team made a groundbreaking contribution to the field of DRL with the introduction of the DQN algorithm in 2013. The DQN represented a fundamental shift from traditional methods by applying a deep learning model to approximate the Q-value function in Q-Learning, enabling the handling of environments with high-dimensional inputs like video games. Recently, the team has demonstrated the ability of DQNs to achieve super-human performance in every single Atari game, the game of GO, chess, and many more. Their implementation is a significant step forward in model free reinforcement learning by focusing only on the most important aspects of the environment for planning and generalizing their implementation. The *MuZero* algorithm [12] is acknowledged as the breakthrough of artificial intelligence and currently paves the way for powerful learning and planning methods to a number of real-world domains.

H. van Hasselt., et al. (2015) [2] proposed the DDQN to address the overestimation bias in the original DQN. This overestimation occurred due to the max operation used in the Q-Learning update, which could lead to overly optimistic Q-value estimates. The DDQN improved upon this by decoupling the selection and evaluation of the action in the Q-value update. The Dueling DQN, proposed by Z. Wang et al. (2016)

[10], introduced a novel network architecture that separately estimates the state value function and the state-dependent action advantage function. This architecture allowed for better policy estimation even when only a few actions were available. T. Schaul et al. (2016) [5] introduced Prioritized Experience Replay, which improved the DQN's efficiency by learning more from significant transitions. This method changed the sampling process in the DQN's experience replay, making it proportional to the absolute TD-error, instead of being uniformly random.

In the following sections, the implemenation of a DQN agent for Atari Pong is developed, taking into account the advancements of the aforementioned studies.

III. Problem Formulation

A. Levels of Success

This project aims to develop algorithms in increasing complexity for training agents to play the classic Atari game of Pong, using techniques from deep reinforcement learning. The development and evaluation of these algorithms are structured around the following three major milestones, which should allow for insight into the advantages and disadvantages of certain algorithms with respect to a consistent environment.

1) **Develop a Deep Q-Network Algorithm.** The first milestone is to develop a basic Deep Q-Network (DQN) algorithm, following many of the processes laid out in the paper from V. Mnih, et al [1], to train an agent to play Pong. The DQN algorithm will need to combine convolutional neural networks with Q-learning, allowing the agent to learn directly from raw pixel inputs and make decisions based on the state of the game. This milestone will involve implementing the DQN architecture, designing a suitable reward function, and training the agent using the game environment interface provided by Gymnasium and Arcade Learning Environment (ALE). Details on these interfaces are provided in the Environment Setup section below.
2) **Modify the Algorithm for Double DQN and Prioritized Experience Replay.** The second milestone focuses on improving the performance and stability of the learning process. To achieve this, I will modify the existing DQN algorithm to incorporate two enhancements: Double DQN and Prioritized Experience Replay. Double DQN aims to reduce the overestimation of Q-values which may lead to overoptimistic value estimates. By separating the selection of actions from their evaluation and using two separate networks, one for action selection and another for action evaluation, this will result in less overoptimistic value estimates.
   Prioritized Experience Replay, on the other hand, aims to improve the learning efficiency by emphasizing more informative experiences during the training process. Instead of sampling experiences uniformly, the algorithm makes the most out of replaying memory from each experience based on TD-error, thus sampling more important experiences with a higher probability.
   With these two modifications made to the original DQN algorithm, I will compare the results obtained with the algorithm developed in Milestone 1 to assess the impact of the modifications and their and characterize their performance in the learning process.
3) **Implementing the *MuZero* algorithm for improved performance and generalization across various game environments.** The final milestone involves exploring a different type of algorithm to facilitate the training of the agent across multiple Atari games by leveraging the knowledge acquired in Pong. This third milestone expands upon the DQN, Double DQN, and prioritized experience replay concepts by incorporating the *MuZero* algorithm, which combines the strengths of model-based planning and model-free deep reinforcement learning. *MuZero* effectively integrates tree-based search with a learned model to plan and make decisions in various game environments without relying on explicit game rules.
   By implementing the ideas of the *MuZero* algorithm in Pong and potentially across multiple Atari games, I aim to demonstrate the potential of model-based planning models to generalize across different tasks and enhance their overall applicability in real-world scenarios.

B. Environment Setup

1) Gymnasium: This project relied on a fork from OpenAi's Gym library for reinforcement learning called Gymnasium. It provides an API in order to limit the overhead for many simple and traditional reinforcement learning environments. Gymnasium provides an API for single agent common environments, the one of which will be studied in this paper is the Atari Pong environment. The API also provides four key functions which are **make**, **reset**, **stop**, and **render**, which allow the user to generate and interact with environments. Each environment is represented as a markov decision process allowing for construction of the agent/environment acting loop. [6]

2) Arcade Learning Environment: Along with Gymnasium, the arcade learning environment (ALE) provides a framework specifically for Atari 2600 games to test and deploy AI agents. [4] ALE is very useful for this project since it also provides a game-handling layer that can be used for reinforcement learning problems. The following section will detail the spaces and reward definitions that are generated from using ALE, specific to Pong. ALE lets the user also initialize episodes and emulate frames up to 6000 frames per second across many games through a common interface.

3) Atari Pong Formulation: The Pong environment consists of an observation space that is a 210x160x3 RGB image representing the game screen where each layer has three color channels for red, green, and blue. This observation space captures the player and the computer's paddles, score, and ball with each pixel ranging from 0 to 255. Since this observation space is quite large and we are not concerned with the actual colors in the game, the observation state space was modified and transformed to a grayscale image with less resolution to make the training process quicker by reducing computational complexity. Details on this can be found in the section below, Solution Approach.

The next parameter of the environment is the action space, where ALE naturally provides the full 18 actions that are possible with a digital joystick. Again, to reduce the computational complexity of this environment, this can be reduced down to the essential discrete actions required to play pong. This includes the following actions.

- No-Operation (noop): Paddle doesn't move
- Move Right (up): Move the paddle up
- Move Left (down): Move the paddle down

The action space is assumed to be deterministic in that the transition probabilities are 100 percent and mapped directly to the action.

Finally, the reward function provides the agent with the ability to actually learn better policies during the game. The agent received positive rewards for getting the ball across the opponent's paddle and losses points if the ball passes the agent's paddle. Every game is played first to 21 points so the maximum number of rewards the agent can get is 21 and the lowest is -21. The reward function is shown by the conditions below.

- +1 point: The agent scores a point by hitting the ball past the opponent's paddle.
- -1 point: The opponent scores a point by hitting the ball past the agent's paddle.
- 0 points: No change in the score during a timestep.

By maximizing the cumulative reward, this reward policy encourages the agent to learn a policy that can effectively strike the ball and try to get it past the computer's paddle.

IV. Solution Approach

The sections below provide specifics on the implementation to validate each major milestone in this project, which are done so using a few key tools to assist in the approach of this problem. These algorithms are implemented by constructing models using the Tensorflow library in python allowing for an efficient setup of training agents and a flexible platform for designing solutions, along with additional processing of the observation data received from using the Gymnasium environment. The below sections detail the processes and parameters used to developing each respective part.

A. Pre-processing Data

As episodes are started and frames are generated, the observation spaces end up being much higher resolution and more data than we need to train the model. This is why each frame that is trained on by the agent is pre-processed prior to being input to the agent. Each frame is cropped, converted to grayscale, and resized using nearest-neighbor interpolation where it can converted to a uint8 datatype. These images are now 88x80 pixels in resolution which is a dramatic decrease from the original 210x160x3. Although, doing this only on one frame at a time is inadequate for training our model. This is because as the agent will only be seeing a single observation of the environment, it has no idea which way the ball or paddles are moving. This is why frames are combined together to form consecutive frames which now allow the agent to extract the velocity from the ball. For the purpose of the game of pong, each frame is combined with it's previous four frames to form an input to the network of 88x80x4.

B. Deep Q-Network Learning Algorithm

The DQN algorithm is a multi-layered neural network that takes inputs from the observation of the game and outputs a vector of action-value pairs, Q-values. Doing this allows the network to learn optimal policies for a wide range of problems.

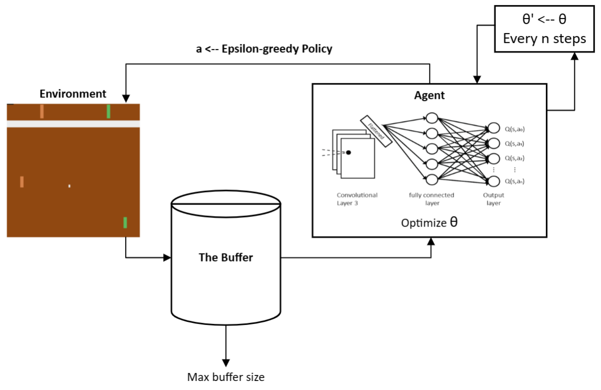

Fig. 1. Overview of the Deep Q-Network Training Loop

To estimate this network, we can define the following target network equation.

$$y_j = \begin{cases} r_j, & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-), & \text{otherwise} \end{cases} \tag{1}$$

with the loss function being the following.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[(y_j - Q(s_j, a_j; \theta))^2] \tag{2}$$

By taking the gradient descent update of the loss function, we can update the network accordingly.

The general approach for this algorithm is displayed below in which the main training loop follows these steps.

Algorithm 1 Training Loop for Deep Q-Network Algorithm

```
1: while Less than Max episodes do
1:     a ← argmax$Q(s,a)$ w.p $1-\epsilon$ {Choose action with epsilon-greedy policy}
1:     $r \leftarrow takestep(env, a)$
1:     $s' \leftarrow observe(env)$
1:     Store experience in buffer (s,a,r,s')
1:     Sample Random minibatch of data from buffer
1:     run optimization {Optimize loss function by taking gradient descent step}
1:     $s \leftarrow s'$
2: end while =0
```

There are two important factors when design a DQN algorithm however, one is the use of an experience buffer, and another is the use of a target network. This target network contains the same parameters as the online network, expect that it is copied from the original network every n-steps. The moving target allows the network to train more effectively and improve stability. The experience replay is where state, action, reward, and next state values are stored in an experience buffer, where they may be sampled from to update and train the network. This typically has a maximum size to reduce the amount of old or unwanted data to sample from. Without an experience buffer, samples would be highly correlated when training the network on them or there would only be size-1 batches. This would cause issues for the DQN algorithm as it won't be able to generalize easily and learn new policies. Both of these modifications to the DQN algorithm dramatically improve its performance. Additionally, parameters such as epsilon decay, target network update rate, maximum buffer size, learning rate, and epsilon minimum are crucial hyperparameters to tuning this agent. Exact value for these are found in the table below.

DQN Parameters	
Name	Value
Epsilon Decay	0.1 per 150k steps
Minimum Epsilon	0.05
Max Buffer Size	200,000
Learning Rate alpha	.0005
Batch Size	500
Target Update Rate	10000 steps

1) Network Architecture: Designing the convolutional neural network architecture (CNN) is a crucial step in setting up the DQN algorithm. The architecture should be tailored to the problem at hand while considering the trade-offs between model complexity and computational requirements. This implementation of a CNN architecture for DQN consists of the following layers:

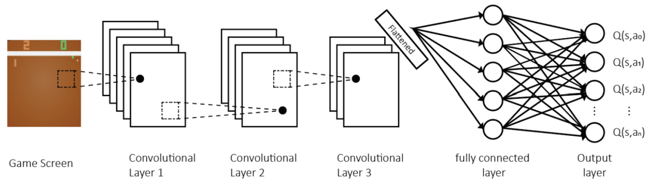

Fig. 2. Full convolutional network structure.

Input layer: The input layer receives the preprocessed image stack, with the dimensions corresponding to the height, width, and the number of stacked frames. This results in an input layer of 88x80x4 layer stack to account for four different frames at a time. By taking an input of four consecutive frames, we are able to see more consistent results since the agent will have a better idea about how the ball and paddles are moving over time.

Convolutional layers: Several convolutional layers are used to extract hierarchical features from the input images. Each layer applies convolution operations with a set of learnable filters or kernels, followed by an activation function, Relu. There are three different convolutional layers in this case, with 32, 64, and 64 filters respectively and 8x8,4x4, and 3x3 kernels, all with a stride of 2.

Fully connected layers: After the convolutional layers, two more fully connected layers are employed to map the extracted features into the Q-value predictions for each action. The number of output nodes in the final fully connected layer matchs the size of the action space. These dense layers contain 512 and the length of the action space number of nodes with the activation function again being Relu. With all of the layers connected we end up with 708691 total trainable parameters.

Optimizer: The ADAM optimizer from the python Keras library was used to update the CNN weights and minimize the loss function. In this optimizer, a learning rate of .0005 is set.

C. Double Deep Q-Network Learning Algorithm

The improved Double DQN (DDQN) learning algorithm takes a similar approach as mentioned above in the DQN section, with minor modifications made to the max operator on the expectation equation. The critical difference in this algorithm is found in the Q-learning target step, where the following target equation is updated to include two index to the Q-function. DDQN uses the same update as DQN, however this new modification reduces overestimation by decomposing the max operation for action selection and evaluation. [2]

$$y_j^{DDQN} = r_j + \gamma Q(s', arg \max_{a'} Q(s', a'; \theta_i)\theta^-) \tag{3}$$

D. Prioritized Experience Replay

Built further on top of DQN and DDQN, prioritized experience replay should improve the performance again by assigning priority values to the experience in the replay buffer. By choosing better experience that is considered more important, our agent would be able to learn more effectively. This is implemented by modifying the replay buffer to store experience along with a priority rating which is derived from its TD-error. The TD-error is simply the equation below with the priority rating updated such that no experience values had a priority of 0.

$$TD_{error} = r_j + \gamma Q(s', arg \max_{a'} Q(s', a'; \theta_i)\theta^-) \tag{4}$$

$$Priority = |TD_{error}| + \epsilon \tag{5}$$

After a priority has been assigned to the experience tuple, we must now determine how to actually sample the replay buffer. It is shown that greedy TD-error prioritization has issues such that values may never be sampled or not for a very long time and it is sensitive to noise spikes. [9] Due to some of these issues, a stochastic sampling method was chosen such that it was in between greedy sampling and pure normal sampling. The probability of sampling a given observation in the replay buffer is given by the equation below, where α determines how much prioritization is used.

$$P(i) = \frac{p_i^\alpha}{\Sigma_k p_k^\alpha} \tag{6}$$

E. Tree-Based Search with a Learned Model

The final technique should enable a reinforcement learning agent to incorporate a tree search algorithm, such as Monte Carlo Tree Search (MCTS). In the context of this project, this algorithm is being used to investigate its performance and ability to generalize the training process. The learned model and tree search will be combined to create a single agent that can plan and learn simultaneously. The agent will be able to update its model based on the outcomes of its simulated actions and learn to make better decisions over time.

For more explicit details, this algorithm takes the approach in reference listed below. A breif description of the process however, the model is represented by combining a representation function, a dynamics function, and a prediction function. The dynamics are represented deterministically, in which the policy and value functions can be computed from the internal state by the prediction function. Given this model, we can then search over hypothetical future trajectories given observations. This is done by implementing an MCTS with upper confidence bounds algorithm to output an optimal policy and estimated value in which an action can be selected. Every node of the search tree corresponds to an internal state where the edges are state-actions pairs. The simulations start from the root state of the tree, where it goes through a selection process for an action at each timestep to maximize over the following upper confidence bound. [12]

$$a^k = arg\max_a [Q(s,a) + P(s,a)\frac{\sqrt{\Sigma_b N(s,b)}}{1+N(s,a)} \\ (c_1 + \log(\frac{\Sigma_b N(s,b) + c_2 + 1}{c_2}))]$$

The tree is then expanded based on the dynamics function and a new node corresponding to the state is added to the search tree. Finally a backup is performed where the statistics along the trajectory are updated and the cumulative discounted reward can be estimated.

V. Results

In this section, I will present the results that were achieved for this project. The DQN algorithm for playing the game of Pong was analyzed on a few different criteria, as well as the learning process. By extracting useful data from the training process, it becomes apparent whether or not the agent is actually learning anything useful. This also translates directly to the performance of the agent when running simulations, since the evaluation tells us exactly what the average score per game is.

A. Empirical Results

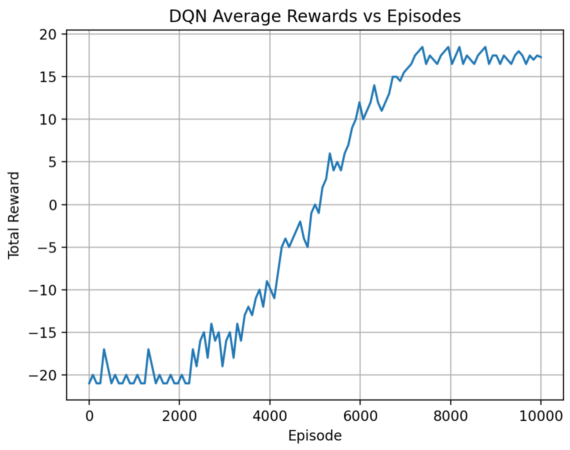

Fig. 3. Average reward over the past 100 episodes plotted over the time history of the agent training.

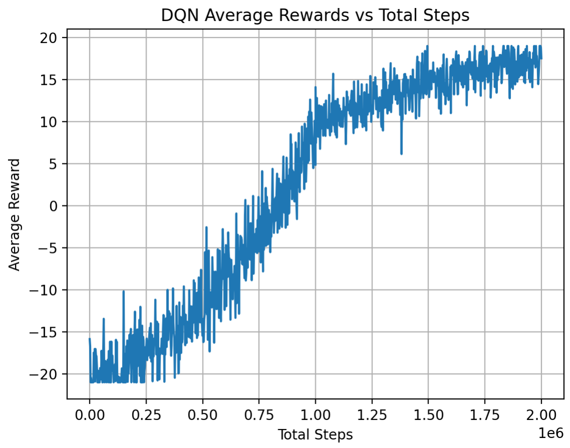

Fig. 4. Reward per iteration for training the agent.

The average score per game for the trained DQN agent was 17.5 points. Given that the maximum possible score in a game of Pong is 21 points, this suggests that the DQN agent was able to consistently win games with quite a large margin against the opponent. Although not achieving the maximum scores, this proves the effectiveness of deep neural networks ability to learn optimal policies.

By using an epsilon-greedy action selection strategy, the DQN agent was able to effectively balance the exploration-exploitation trade-off during the learning phase. The agent started with a high exploration rate ($\epsilon = 1.0$) and gradually decreased it to a minimum value ($\epsilon = 0.05$) over a couple million steps in the environment. This approach allowed the agent to explore various game states and actions in the initial phase, leading to a more comprehensive understanding of the game environment. As the agent gained experience and developed a better strategy, the exploitation of the learned Q-values became more dominant, which resulted in improved performance over time. Since the game space is known to begin with, the need for constant exploration late in the training process is minimal compared to exploiting the existing policies.

Also, the DQN agent demonstrated a relatively stable convergence. Within the first million training steps, the agent achieved a noticeable increase in performance, and by episode 8000, it reached fairly consistent average return values over the past 100 episodes. With more training, it may be possible to achieve higher, near-optimal scores, however due to time constraints on the training process, this data was unable to be collected.

B. Discussion

The results above show completion of the level one objects for this project. The additional levels of success were unable to be met due to complexity with implementing and analyzing the agent during the learning process. Also due to the relatively large observation spaces and agent parameters that I was dealing with in this project, the training time was much longer than expected where hour time intervals would have to go by before a noticeable update in the reward could be seen. This could also be from inefficient data handling in the implementation or possibly additional tuning of the hyperparameters to improve speed efficiency.

VI. Conclusion

A. General

This paper sought out to explore reinforcement learning strategies, and in particular DQN and modified versions of DQN. DQN is proven to be a powerful approach to developing optimal policies for many different types of applications in many domains. Namely, when dealing with high dimensional state spaces or sensory data such as images or videos, deep neural networks are very successful tools for training these large data sets. There are clear results shown for a DQN agent being able to learn policies for playing the Atari game of Pong. There are many more possibilities to expand this to other types of environments related to robotics or artificial intelligence agents.

B. Lessons Learned

- **Developing and employing effective parametric extraction is crucial for verifying the learning process of neural network agents.** It became apparent during this project that it is critical to gain insight into the inner workings of these neural networks by leveraging visual and numerical analysis tools. Interpreting these updates and results in a clear manner can enable the optimization of model architectures, hyperparameters, and training methods. By developing those pipelines earlier in the design of my solutions, there would have been less time wasted on debugging tasks, and more information to guide the development towards a more optimal solution.
- **Ethical Implications for Real-World Applications** Considering the ethical implications of deploying DQN-based agents in real-world applications was a final takeaway from this course. Future work should involve the development of safety and fairness measures, ensuring that the algorithm's decisions are transparent, interpretable, and aligned with desired goals.

Contributions and Release

Daniel Mathews is the sole contributer to this project. The author grants permission for this report to be posted publicly.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," arXiv.org, 19-Dec-2013. [Online]. Available: https://arxiv.org/abs/1312.5602.
[2] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," arXiv.org, 08-Dec-2015. [Online]. Available: https://arxiv.org/abs/1509.06461.
[3] M. C. Mchado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling, "Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents," [Online]. Available: https://jair.org/index.php/jair/article/view/11182/26388.
[4] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," arXiv.org, 21-Jun-2013. [Online]. Available: https://arxiv.org/abs/1207.4708.
[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," arXiv.org, 25-Feb-2016. [Online]. Available: https://arxiv.org/abs/1511.05952.
[6] "Gymnasium documentation," Pong - Gymnasium Documentation. [Online]. Available: https://gymnasium.farama.org/environments/atari/pong/.
[7] Mgbellemare, "Mgbellemare/arcade-learning-environment: The arcade learning environment (ALE) – A platform for AI research.," GitHub. [Online]. Available: https://github.com/mgbellemare/Arcade-Learning-Environment.
[8] D. Parmelee, "Getting an AI to play Atari Pong, with Deep Reinforcement Learning," Medium, 26-Jan-2021. [Online]. Available: https://towardsdatascience.com/getting-an-ai-to-play-atari-pong-with-deep-reinforcement-learning-47b0c56e78ae.
[9] "ArXiv:1911.08265v2 [cs.LG] 21 Feb 2020." [Online]. Available: https://arxiv.org/pdf/1911.08265.pdf.
[10] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," arXiv.org, 05-Apr-2016. [Online]. Available: https://arxiv.org/abs/1511.06581.
[11] "Improving DQN training routines with transfer learning - georgehe.me." [Online]. Available: https://georgehe.me/dqn_transfer.pdf.
[12] https://arxiv.org/pdf/1911.08265.pdf
[13] https://towardsdatascience.com/deep-q-network-with-pytorch-146bfa939dfe