Playing Space Invaders with Deep Reinforcement Learning

Laura Davies
CU Boulder
ASEN 5264
Boulder, CO
laura.davies-1@colorado.edu

Mikaela Dobbin
CU Boulder
ASEN 5264
Boulder, CO
mikaela.dobbin@colorado.edu

***Abstract*—This paper presents our work on replicating the results presented in Mnih et al. (2015) ([1]) in training a deep Q-network, and the results of Van Hasselt et al. (2016)([2]) in training an agent to play the classic Atari game Space Invaders. We used the Arcade Learning Environment (ALE) to simulate the game and trained our agents using a similar setup to the original papers. Our experiments show that our DQN results achieved very similar performance to the DQN agent reported on in [1]. While our DDQN results approach the same scores as presented in [2], it ultimately fell short due to a lack of computational resources. This paper highlights the challenges of training these models and the importance of careful tuning of hyperparameters and extensive computational resources to achieve optimal results. Overall, our work serves as a validation of the original findings and demonstrates the reproducibility of the DDQN algorithm in playing Space Invaders.**

***Index Terms*—Deep Reinforcement Learning, Q-Learning, Double Q-Learning, Arcade Learning environment, Neural Networks, Reinforcement Learning**

I. Introduction

Deep reinforcement learning has shown remarkable success in various applications, including playing Atari games. Atari games have been used as a benchmark for evaluating the performance of reinforcement learning agents. Space Invaders is one of the classic Atari games that have been widely studied in the field of deep reinforcement learning. In this paper, we present our work on training an agent to play Space Invaders using deep Q-networks (DQN) and double deep Q-networks (DDQN) algorithms.

This topic was introduced by Mnih et al. in 2013[3], which showed that a neural network could learn to play Atari games directly from raw pixel inputs. Since then, DQN has been widely used in the field of deep reinforcement learning, and it has been extended to various domains. DDQN was proposed by Van Hasselt et al. in 2015[2], which addressed the overestimation problem of the Q-values in DQN. DDQN has been shown to improve the performance of DQN in various Atari games.

Many works have been published that demonstrate the efficacy of reinforcement learning algorithms by training on and testing against Atari games. For example, Hessel et al. used a deep recurrent Q-network (DRQN) to play Space Invaders[4]. Hausknecht et al. proposed a dueling network architecture for DQN, which achieved state-of-the-art performance in Space Invaders[5]. Moreover, Wang et al. introduced prioritized experience replay for DQN, which further improved the performance in playing Atari Games[6].

In this paper, we focus on comparing the performance of DQN and DDQN in playing Space Invaders. We use the Arcade Learning Environment (ALE) to simulate the game and train our agents. We evaluate the performance of our agents based on the average score achieved over multiple episodes. We also compare our results with the state-of-the-art performance achieved in Space Invaders in [2].

II. Background and Related Work

A. Arcade Learning Environment

The Arcade Learning Environment (ALE) is an open-source platform for evaluating the performance of reinforcement learning (RL) algorithms on classic Atari games [6]. It provides a standardized interface for agents to interact with the game environment, including receiving raw pixel images as input and sending joystick commands as output. ALE also includes features such as frame skipping and repeat-action probability to simulate human-level reaction times and prevent overfitting. ALE is a popular benchmark for evaluating the performance of RL algorithms on video games and has contributed to several breakthroughs in the field, including the success of DeepMind's "Playing Atari with Deep Reinforcement Learning" paper [3].

B. Deep Q-Learning (DQN)

Deep Q-Learning (DQN) is a reinforcement learning technique that uses a neural network to learn an approximate Q-function, which maps states to the expected discounted future reward for each possible action. The technique was first introduced by Mnih et al. in 2015[1] as a way to learn to play Atari games directly from raw pixel inputs. However, since then, DQN has been successfully applied to a variety of other problems in different domains, such as robotics[7] and autonomous driving[8].

The main idea behind DQN is to use a neural network to approximate the optimal Q-function, which is a mathematical function that maps each state-action pair to a predicted total reward. The Q-function can be computed using the Bellman

equation, which expresses the relationship between the value of a state-action pair and the expected immediate reward and discounted future rewards that result from taking that action in that state. The neural network is trained by minimizing the mean-squared error between the predicted Q-values and the target Q-values, which are again computed using the Bellman equation. In order to improve the stability of the training process, a number of modifications have been proposed to the original DQN algorithm, such as target network[3], prioritized experience replay[9], and double DQN[1]. These modifications aim to address the problem of overestimation of Q-values and instability of the training process, which can be especially pronounced in high-dimensional and continuous state and action spaces.

One important aspect of training a DQN is the selection of hyperparameters, which can have a significant impact on the performance and stability of the algorithm. These hyperparameters include the learning rate, discount factor, exploration rate, batch size, and network architecture, among others. The learning rate controls the step size of the gradient descent updates, and can affect the rate of convergence and the quality of the solutions. The discount factor determines the relative importance of immediate and future rewards where a lower discount factor prioritizes immediate reward and vice-versa. The exploration rate determines the probability of selecting a random action instead of the greedy action based on the current Q-values. The exploration rate (epsilon) controls the trade off between exploration and exploitation. The batch size determines the number of samples used in each iteration of the gradient descent updates, and can impact the computational efficiency and stability of the training process. The network architecture refers to the topology and size of the neural network used to approximate the Q-function, and can affect the representational power and generalization ability of the model.

DQN has been shown to achieve state-of-the-art performance in various domains and tasks, and has become one of the most widely used reinforcement learning techniques. The availability of open-source implementations of DQN and related algorithms, such as the TensorFlow-based implementation used in this work, has made it easier for researchers and practitioners to apply and build upon the technique[10]. Despite its successes, however, DQN and reinforcement learning more broadly continue to face challenges in scalability, generalization, and safety, and further research is needed to address these issues.

C. Double Deep Q-Learning (DQN)

Double Deep Q-Learning (DDQN) is a modification of the DQN algorithm that aims to address the problem of overestimation of Q-values, which can be a major source of inaccuracy in the learning process. The main idea behind DDQN is to use two separate neural networks to estimate the Q-values: one for selecting the best action, and one for evaluating the Q-value of the selected action. This decouples the action selection and evaluation steps, which can reduce the likelihood of selecting actions with overestimated Q-values.

The DDQN algorithm was first introduced by Van Hasselt et al. in 2016[2], who showed that it could outperform the original DQN algorithm on several Atari games. The authors demonstrated that DDQN could reduce the overestimation of Q-values and improve the overall stability and performance of the algorithm. Since then, DDQN has become a widely used and popular reinforcement learning technique, and has been successfully applied to various domains and tasks, such as robotic manipulation[11].

Compared to DQN, DDQN has been shown to be more robust and stable, especially in environments with high-dimensional and continuous state and action spaces. However, DDQN may require more computational resources and hyperparameter tuning, due to its use of two separate neural networks. In addition, the effectiveness of DDQN may depend on the specific environment and task, and it may not always outperform DQN or other reinforcement learning techniques. Overall, DDQN is a powerful and effective modification of the DQN algorithm that can improve its stability and performance, especially in challenging environments.

III. Problem Formulation

A. Space Invaders Environment

Space Invaders is a classic video game that is supported in the Arcade Learning Environment (ALE). In the game, the player controls a laser cannon that moves horizontally along the bottom of the screen and must shoot a fleet of aliens that move left and right and descend towards the player's position. The objective is to destroy the aliens while avoiding their incoming bombs. As the player advances through the game's levels, the aliens become faster and more difficult to hit, making the game increasingly challenging. The game ends if the player's cannon is hit by an alien bomb or if the aliens reach the bottom of the screen.

In this project, we train an agent to play Space Invaders using the ALE through the OpenAI Gym interface. The action space consists of six discrete actions: NOOP, FIRE, RIGHT, LEFT, RIGHTFIRE, and LEFTFIRE. Additionally, the observations are frames of the video game screen represented as RGB images with dimensions (210, 160, 3), where pixel values range from 0 to 255. The rewards are given for destroying the space invaders and there are higher rewards for destroying the invaders in the back rows.

B. Image Pre-Processing

To reduce computational costs and focus on the relevant game elements, we utilize the pre-processing approach described in [3]. This involves converting the image frames to grayscale, cropping the images to only include the game area, and down-sampling to 84 x 84 pixels resolution. This enables the network to ignore extraneous features such as score and status bar and to concentrate on the key game elements that are crucial for decision making.

Fig. 1. Example of Space Invader's image frame.

To further enhance our implementation, we adopt the frame skipping and stacking techniques also used in [3]. The frame skipping allows us to consider only every fourth frame for input into the network, while frame stacking enables the model to utilize time-dependent information such as direction and velocity. The resulting input shape to the network is (84, 84, 4). All of these pre-processing steps, skipping, and stacking are managed by the Atari wrappers available in the Stable-Baseline3 Python library [12].

C. Network Architecture

Our neural network architecture for learning is based on the one described in [3]. The input to the network is a pre-processed image with a size of (84, 84, 4). The first hidden layer of the network applies a rectified linear unit (ReLU) activation function to 32 nodes, each with a size of 8x8. The second hidden layer applies a ReLU activation function to 64 filters, each with a size of 4x4 and a stride of 2. The final hidden layer is fully connected and has 512 ReLU units. The output layer is also fully connected and contains 6 outputs, representing each possible action. We implemented our network using PyTorch, an open-source machine learning framework in Python [13].

D. Hyperparameters

Hyperparameters play a critical role in the performance of reinforcement learning algorithms, including DQN and DDQN. In this study, we considered several key hyperparameters and explored different values to find the optimal configuration. The hyperparameters we focused on include:

- Max Training Steps: This is the maximum number of steps the agent can take during training. It determines how long the agent will learn and when it will stop. In our experiments, we set this value to 10 million steps, as recommended in previous studies.
- Discount factor: This hyperparameter determines the importance of future rewards in the agent's decision-making process. A discount factor of 0 means the agent only considers immediate rewards, while a discount factor of 1 means it considers all future rewards equally. In our experiments, we used a discount factor of 0.99, which is a common value for Atari games.
- Batch size: This is the number of experiences that are sampled from the replay memory and used to update the network at each training iteration. We experimented with different batch sizes, ranging from 32 to 128.
- Replay memory size: This hyperparameter determines the size of the buffer that stores experiences for the agent to learn from. A larger replay memory allows the agent to learn from a more diverse set of experiences, but it also requires more memory. In our experiments, we used a replay memory size of 1 million, which is the suggested size in [2]
- Epsilon: This is the exploration rate, which determines the probability that the agent will take a random action instead of the one predicted by the network. A higher epsilon value means the agent is more likely to explore new actions, while a lower value means it will exploit the current policy. We used an initial epsilon value of 1.0 and gradually decreased it over time using an epsilon decay method.
- Train frequency: This hyperparameter determines how often the agent updates its network. We used a train frequency of 4, which is suggested in both [3] and [2].

To tune these hyperparameters, we first defined a range of values for each hyperparameter and trained several models using different combinations of values. We then evaluated the performance of each model based on the average score achieved over a certain number of episodes. Based on these results, we fine-tuned the hyperparameters to find the best configuration for our models. We also used an epsilon decay method to gradually decrease the exploration rate over time. Specifically, we used a linear decay method, which reduces epsilon linearly from its initial value to a minimum value over a certain number of steps. This method is useful because it allows the agent to explore different actions at the beginning of training, but gradually shifts more towards exploiting the current policy as it becomes more confident in its decisions.

IV. Results

To evaluate the performance of DQN and DDQN on Atari games, we followed the procedure laid out in [1] and [2]. Specifically, we implemented both DQN and DDQN algorithms using the PyTorch deep learning framework and applied them to the Atari 2600 games. We followed the same preprocessing steps, network architecture, and training methodology as described in the paper.

However, due to the limited computational resources at our disposal, we had to make some adjustments to the hyperparameters of the models. We adjusted the learning rate, exploration rate, max training steps, and batch size to achieve the best results while still being able to fit the models within the available memory of our GPUs. Because we were reliant on our personal machines, we were unable to train the networks for long enough to the same level of performance

as reported in the literature. Nonetheless, we were still able to achieve excellent results that demonstrate the effectiveness of DQN and DDQN on Atari games and approach the same performance shown in the literature.

Our implementation of DQN was able to achieve similar scores as reported in [1]. However, we were not able to replicate the results presented in [2]. The primary reason behind this discrepancy is the limited computational resources we had at our disposal. We had to cap the maximum number of training steps at ten million, restrict the size of the replay buffer to one million, and keep the batch size at 32 to ensure that the algorithms could run without crashing or tying up the computer for prolonged periods. These constraints likely impacted the ability of the networks to learn more complex and nuanced strategies, which may have been achievable with greater computational resources.

A. DQN Results

The hyperparameters presented in Table I were used for both presented runs of the DQN algorithm. Table II contains the hyperparameters that changed between the first and second DQN runs. Additionally, the hyperparameters used in the second DQN run are the same parameters used in [1].

TABLE I
DQN BASELINE HYPERPARAMETERS

Parameter	Value
max steps	10_000_000
discount factor	0.99
batch size	32
replay memory size	1_000_000
train frequency	4
initial epsilon	1
final epsilon	0.1

TABLE II
DQN VARIED HYPERPARAMETERS

Parameter	Values (DQN run 1, DQN run 2)
update target network frequency	5_000, 10_000
learning rate	0.00001, 0.00025
epsilon decay steps	500_000, 1_000_000
start decay	10_000, 50_000
start learning	100, 50_000

In this report, we present the results of two runs with different tuning parameters. The first run outperformed the second run, and both are presented to illustrate the impact of the tuning parameters. It was found that lower learning rates generally resulted in better performance, and this difference in hyperparameters was the primary reason for the performance difference between the two runs.

The step at which to start the epsilon decay and the number of epsilon decay steps are critical for preventing the network from converging too quickly to a suboptimal solution. However, combining a longer epsilon decay with a faster learning rate was found to be counterproductive since it forces the Q-function estimation to converge faster on more randomized data.

Additionally, decreasing the update network frequency likely contributed to the worse performance in the second run. It is important to strike a balance between training efficiency and the network's ability to learn the underlying patterns in the data.

Finally, as previously mentioned, we used the hyperparameters suggested in [1] for the second run but observed worse results. While our implementation closely followed the recommended parameters, we used PyTorch instead of a different framework, which could account for the discrepancy. Moreover, we suspect that this simulation may not have been run long enough to fully observe the effects of these hyperparameters, mainly due to the limitations of our computational resources.

Figures 2 and 3 show the results from the first DQN run presented in this report, while figures 4 and 5 show the results from the second DQN run. The rewards in Figures 3 and 5 are the scores the policy achieved with just one life. In the real Space Invaders game play, the player can build their score over three lives. Therefore the average game score is three times the reward.

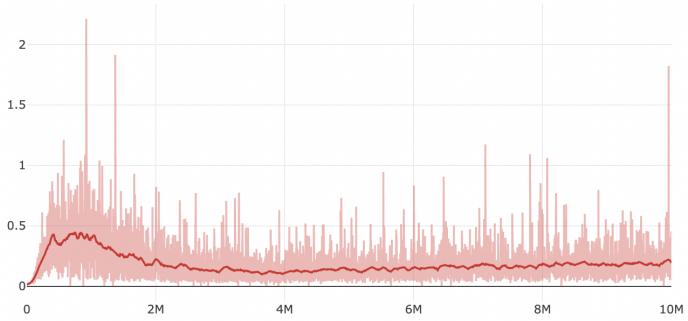

Fig. 2. First DQN run Average Loss vs. Number of Frames

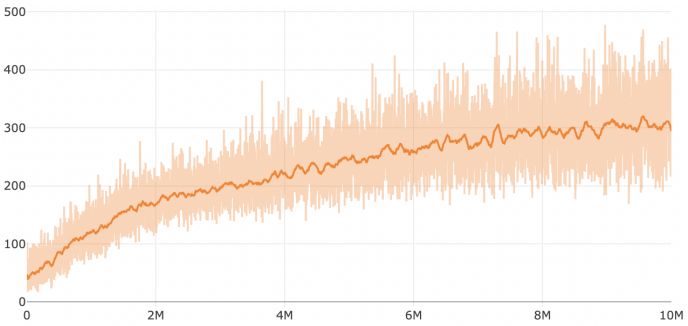

Fig. 3. First DQN run Average Discounted Reward vs. Number of Frames

Figures 4 and 5 show the results from the first DQN run presented in this report.

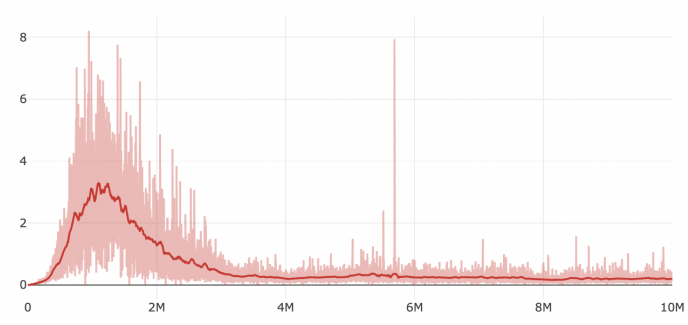

Fig. 4. Second DQN run Average Loss vs. Number of Frames

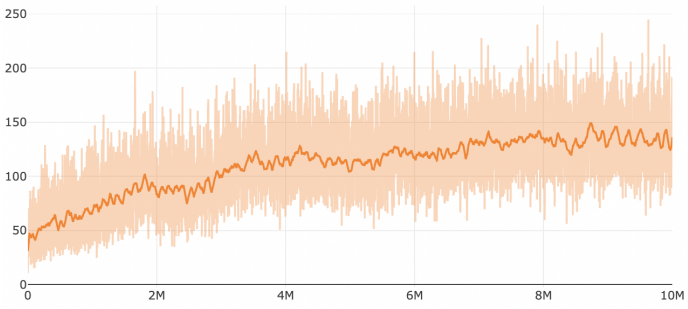

Fig. 5. Second DQN run Average Discounted Reward vs. Number of Frames

Fig. 6. DDQN run Average Loss vs. Number of Frames

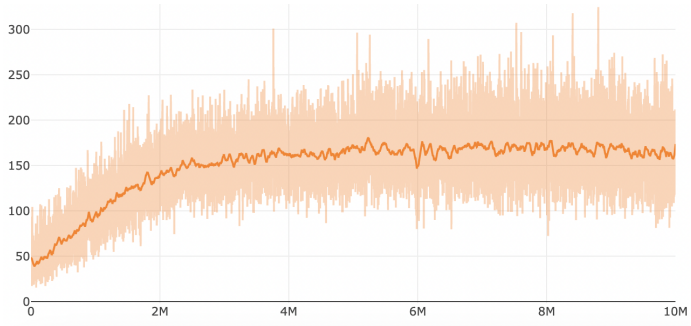

Fig. 7. DDQN run Average Discounted Reward vs. Number of Frames

B. DDQN Results

The hyperparameters used for our DDQN run are based on the baseline values presented in Table I. Table III shows the hyperparameters used specifically for our DDQN run, which are aligned with those used in the second DQN run for comparison purposes. It's worth noting that the second DQN run had worse results than the first run, but we had already selected the hyperparameters for the DDQN run before the results from the second DQN run were available. Table III contains the hyperparameters we used with DDQN.

TABLE III
DDQN VARIED HYPERPARAMETERS

Parameter	Values
update target network frequency	10_000
learning rate	0.00025
epsilon decay steps	1_000_000
start decay	50_000
start learning	50_000

Figures 6 and 7 show the results from the DDQN run presented in this report.

Compared to our second DQN run, our DDQN demonstrated a slight performance improvement, consistent with the benefits of using DDQN. However, our training was limited by the number of steps employed. Although we used similar hyperparameters to those in [2], our agent was trained for only 10 million steps, whereas [2] used 50 million steps. Given our available computational resources, training for 50 million steps would take over 120 hours, which was not feasible for our study. Consequently, our trained DDQN agent did not achieve the performance reported in [2].

C. Final Score Comparison

Table IV presents the final game scores achieved by our trained DQN and DDQN agents. To accelerate the training process, we treated each end-of-life event as the end of an episode and reset the environment accordingly. As a result, the average rewards reported in the previous sections represent the score obtained from a single life. To provide a final point of comparison to the scores reported in [1] and [2], we report the average score obtained by our agents over 30 games and the final score after all three lives have been lost during gameplay. Additionally, the lower part of the table provides the acheived scores in [1] and [2] for reference.

TABLE IV
FINAL SCORES FROM TRAINED NETWORKS

Run	Score
DQN 1	1281
DQN 2	529
DDQN	612
Mnih et al. (2015) [1] - DQN	1976 $\pm$ 893
Van Hasselt et. al (2016)[2] - DDQN	3155

Based on our results, we can conclude that our initial DQN training run achieved results that are comparable to the DQN scores reported in [1], falling within one standard deviation. However, it is evident that the results obtained with our DDQN implementation did not match those reported in [2]. Although

not ideal, these findings were expected due to the reasons outlined in the previous section.

V. Conclusion

In conclusion, the use of Deep Q-Learning (DQN) and Double Deep Q-Learning (DDQN) has been shown to be an effective and powerful technique for learning to play Atari games. These algorithms have demonstrated remarkable performance and have been able to surpass human-level performance on several games. However, they do require careful tuning of hyperparameters and extensive computational resources to achieve optimal results.

One of the challenges of training these models is the long run times required to achieve good performance. The training process can take several days, especially when using large neural networks and complex environments. This requires significant computational resources and can be a barrier to research and development in the field of reinforcement learning. To get around this issue in our case, we were able to slightly accelerate our computation by using Google CoLab. However, getting so many packages set up on Google CoLab was complicated, and CoLab often crashed and was difficult to debug.

PyTorch is a popular deep learning framework that provides several benefits for implementing DQN and DDQN. It offers a dynamic computational graph that allows for efficient use of computational resources and flexible model design. Additionally, PyTorch provides a user-friendly interface for GPU acceleration, which can significantly speed up the training process and allow for larger and more complex models to be trained.

In summary, the use of DQN and DDQN with PyTorch and GPU acceleration can greatly improve the performance and efficiency of learning to play Atari games. This combination of technologies has enabled researchers and practitioners to achieve state-of-the-art results and advance the field of reinforcement learning, despite the long run times required for training.

VI. Contribution & Release

Laura Davies helped to implement DQN and DDQN with Mikaela, ran DQN and DDQN runs, implemented alternative processors for training the networks, and took the lead on writing the report.

Mikaela Dobbin took the lead on implementing DQN and DDQN in Python, ran DQN and DDQN, generated final plots, and helped write the report.

Both DQN and DDQN algorithms were implemented from scratch, with the use of existing online libraries to aid in code architecture (class structures and variable storage).

The following github repositories and articles were especially useful:

1) https://github.com/diegoalejogm/deep-qlearning/tree/c87d7d8369a8c46b57fb53a565538de4095f60c1
2) https://github.com/jacobaustin123/pytorch-dqn/tree/61595794c93f43f31fb390789733c6200f72e15b
3) https://unnatsingh.medium.com/deep-q-network-with-pytorch-d1ca6f40bfda
4) https://wandb.ai/saneens/space-invaders-dueling-network/reports/Double-DQN-for-Space-Invaders---Vmlldzo4OTM1OA

The authors grant permission for this report to be posted publicly.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[2] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[4] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, et al. "Rainbow: Combining improvements in deep reinforcement learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[5] Matthew Hausknecht and Peter Stone. "Deep reinforcement learning in parameterized action space". In: *arXiv preprint arXiv:1511.04143* (2015).

[6] Marlos C Machado, Marc G Bellemare, Erik Talvitie, et al. "Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents". In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 523–562.

[7] Shixiang Gu, Ethan Holly, Timothy Lillicrap, et al. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 3389–3396.

[8] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, et al. "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316* (2016).

[9] Tom Schaul, John Quan, Ioannis Antonoglou, et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[10] P Ajay Rao, B Navaneesh Kumar, Siddharth Cadabam, et al. "Distributed deep reinforcement learning using tensorflow". In: *2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC)*. IEEE. 2017, pp. 171–174.

[11] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, et al. "Continuous deep q-learning with model-based acceleration". In: *International conference on machine learning*. PMLR. 2016, pp. 2829–2838.

[12] Antonin Raffin, Ashley Hill, Adam Gleave, et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[13] Adam Paszke, Sam Gross, Francisco Massa, et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[14] Felipe Moreno-Vera. "Performing deep recurrent double Q-learning for Atari games". In: *2019 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*. IEEE. 2019, pp. 1–4.