Dueling Double Deep Q-Network for Training an Autonomous Vehicle in Simulation

Rita Tlemcani
Stanford University
Computer Science
ritabt@stanford.edu

***Abstract*—This paper presents an implementation and study of using Dueling Double Deep Q-Networks (DQN) to train an autonomous vehicle in a simulated Unreal Engine 4 environment. The reinforcement learning system uses RGB image inputs from a virtual camera placed at the front of the simulated car. We investigate different network architectures and reward functions then we compare the effect these changes have on the performance of the trained agent. The performance of the trained agent is evaluated quantitatively using the reward function and qualitatively by observing the behavior of the autonomous vehicle. We show that a Dueling Double DQN that uses ResNet50 as a preprocessing step for the images combined with a reward function that uses simulated waypoints to encourage staying in the right lane performs best. Using transfer learning with the Dueling Double DQN method showed better results when training an autonomous vehicle in simulation compared to the traditional convolutional method.**

***Index Terms*—DQN, Dueling Double Deep Q-Network, Reinforcement Learning, Simulated Autonomous Vehicle, ResNet50, Transfer Learning**

I. Introduction

Reinforcement Learning (RL) has many applications in the field of robot autonomy. In this study, we investigate using Deep RL to train a car to drive autonomously in a simulated environment. The Deep RL method we implement is a variation of the Deep Q-Network (DQN) [11]. Many modifications have been made to the original DQN algorithm to address some of its shortcomings such as maximization bias. The modifications we implement are Dueling DQN [17] and Double DQN [16] combined to create a Dueling Double DQN.

A high-quality realistic simulation environment is preferable to improve the ability of our model trained on simulation to perform well in the real world. To create a realistic simulation environment with interactions that work well with the goals of this project, we specifically combine four different tools: Unreal Engine 4 [5], Carla [4], Airsim [14], and Gym [2]. We use the assets from Carla [4], such as 3D models and lighting, to build the town and the Airsim interface to train the agent in the Unreal Engine 4 simulator. We use a virtual camera placed at the front of the car to extract RGB images that we use as a state for the car agent. With this simulation setup, we have access to input variables (steering and throttle), and simulation output images necessary to train our Dueling Double DQN model.

Some challenging aspects of training a DQN are state representation, neural network architecture, and reward function design. That is because small changes in these methods can greatly impact the performance of the trained model, and thus finding the right formulation is crucial. In this project, we experiment with two different state representations: raw pixels (RGB image) and using a pretrained ResNet50 [6] to create a 2048 vector representation of the state from the image. Depending on the state representation method, we modify the network architecture to accommodate the input type. When using the raw pixels as inputs, the network is a Convolutional Neural Network (CNN) to be able to process the image. When using ResNet50 [6] as a preprocessing step, we can use a simple fully connected neural network in the DQN.

The remainder of this paper discusses how we analyzed the impact of different state representations and network architectures on the agent's learning. We also analyze how different reward function formulations change the behavior of the trained agent. First, however, it discusses previous work that informed our approach.

II. Previous Work

RL algorithms are used to train an agent to navigate an environment through interactions with the environment. These interactions are formalized through a set of states S, a set of actions A and a reward function $R(S, A)$. At time step t, The agent is at state s_t will take an action a_t resulting in state s_{t+1} and receiving reward $R(s_t, a_t)$. The Q-Learning algorithm was introduced by Watkins [18] and uses these values to update a Q-Matrix to learns what the best action is at each state. The best action in this context is the one that maximizes future rewards. The Q-Matrix is updated using the Bellman equation [1].

DQN was introduced by Mnih et al. [11] and replaced the Q-matrix with a CNN. This approach allows states to be images and also allows the DQN to learn how to navigate more complex environments. Training CNNs requires a target value in order to compute the loss and update the weights. Mnih et al. [12] explained how using a Target network to create a target value makes the training more stable. The weights from the Q-Network are copied over to the Target Network every few steps. Mnih at al. [11] also used experience replay to store states transitions in a buffer and use them for batch training to make more efficient use of the training data.

(a) Screenshot of the simulator viewport

(b) Image captured by the front camera on the car

Fig. 1: Visualizations of the environment simulated in Unreal Engine 4 and the images captured by the virtual front camera on the simulated car which are used for training the Dueling Double DQN.

Deepmind's Rainbow algorithm [7] showed that combining multiple DQN modification can greatly improve the performance of the trained network. As such, we look at combining the Dueling DQN [17] and Double DQN [16] methods.

Dueling DQN, introduced by Wang et al. [17], is a neural network architecture that separates the stream into a value function stream and an advantage function stream. This DQN modification was introduced because we do not need to know the exact value of each action at every timestep.

Double DQN was introduced by Hasselt et al [16] and, in this method, two identical neural network models are defined, one for selecting actions and another for evaluating actions. This modification was able to address the maximization bias of the original DQN [11] method.

III. Simulation Environment

The simulation environment that we use for this study is a combination of different software packages. This allows us to create a realistic simulator that is easy to control, making the training scripts more straightforward. All of the following software is used with Unreal Engine 4 [5].

A. Carla

Carla [4] is an open-source simulator for autonomous driving research. The Carla simulator has many realistic town settings and car models that can be used as a training environment for RL projects. In this project, we create a new Unreal Engine 4 project [5] and import the asset files from the Carla build. The asset files are responsible for building the towns. See Fig. 1a for a visualization of the Carla town used in this project.

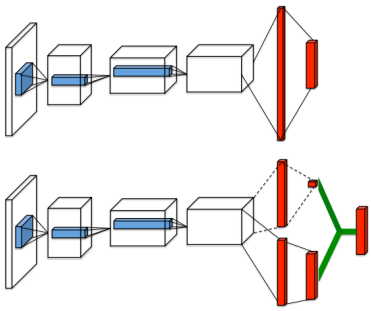


Fig. 2: Representation of the separating streams in a Dueling DQN by Wang et al. [17] (bottom) compared to a regular DQN [11] (top)

B. Airsim

Airsim [14] is an Unreal plugin for car and drone simulations. Once the Carla [4] town is imported in the new Unreal 4 project [5] we use Airsim as a way to interact with the simulated environment. We use the Airsim API in place of the Carla API due to its ease of installation and use. The Python client in the Airsim package offers methods to send an action to the agent and retrieve images from a virtual front facing camera.

C. Gym

OpenAI's Gym [2] is a toolkit for RL projects. It offers an API for controlling an agent in a simulated environment as well as toy environments which can be used for development purposes. In this study, a custom gym environment is created to handle all the agent's interactions. This also allows this project's source code to be generalizable to any environment that follows Gym's standardized simulated environment setup.

IV. Deep Reinforcement Learning Methods

A. Dueling Double DQN

In this project, we combine the Dueling and Double DQN methods. We create a network architecture that splits into two streams as in the Dueling DQN method [17] and duplicate the network similar to the Double DQN method [16].

The Dueling DQN method [17] uses the following equation for the Q-function:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|A|}\sum_{a=1}^{|A|} A(s,a) \qquad (1)$$

Where V is the value function and A is the action function. V and A are the two streams that are added to the DQN network and their output is combined in a final layer as in equation (1). See Fig. 2 for a representation of the separating streams. To make the Dueling Double DQN, we use the network as explained above and create two identical networks: a prediction network Q and a target network Q'. As introduced

by Hasselt et al. [16], we use the following equation to determine our target value Q^* using a discount factor γ:

$$Q^*(s_t, a_t) = r_t + \gamma Q(s_{t+1}, \text{argmax}_{a'} Q'(s_t, a_t)) \quad (2)$$

After calculating $Q^*(s_t, a_t)$ as in equation (2), we minimize the Huber loss between $Q^*(s_t, a_t)$ and $Q(s_t, a_t)$ as in equation (3)

$$L_\delta(Q^*, Q) = \begin{cases} \frac{1}{2}(Q^* - Q)^2 & , \text{ for } |Q^* - Q| \leq \delta \\ \delta \cdot (|Q^* - Q| - \frac{1}{2}\delta) & , \text{ otherwise} \end{cases} \quad (3)$$

B. Epsilon-Greedy Strategy

To balance exploration and exploitation we use an ϵ-greedy strategy [11]. Following this strategy, we select a random action with probability ϵ. By setting the initial ϵ to a high value such as $\epsilon = 0.9$, the agent will mostly take random actions encouraging exploration of the environment. We also used ϵ-decay to decrease ϵ after each episode. We set 0.01 to be a lower bound for ϵ and this method encourages the agent to exploit its learning towards the end of the training.

C. Polyak Update

The weights θ from the prediction network Q are used to update the weights θ' of the target network Q' every few episodes. We used the Polyak update, also known as soft update [10], for updating the target network. With the Polyak update, we do not update Q' all at once but instead we update it little by little each time. Equation (4) represents how we update the target network weights θ'

$$\theta' = \tau \cdot \theta' + (1 - \tau) \cdot \theta \quad (4)$$

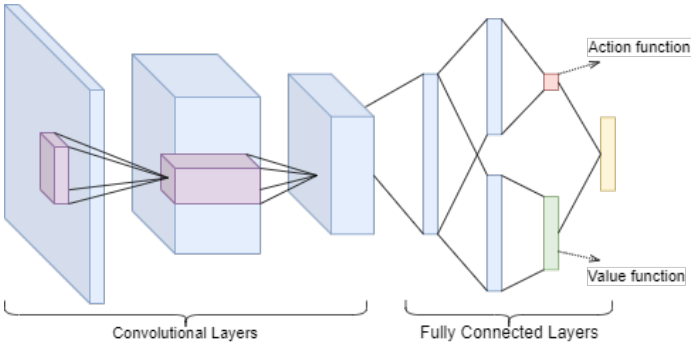
Where τ is the Polyak which is a hyperparameter that determines how much the new target weights will be different from the old ones. A higher polyak τ means that we are keeping more of the original θ' values.

D. Randomized Experience Replay

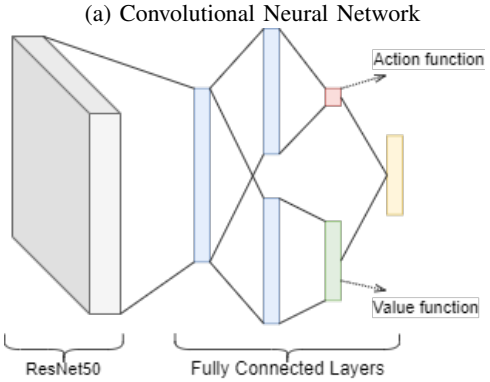
After observing a state s, taking an action a, receiving a reward r, and observing a new state s', we next use the experience $e = (s, a, r, s')$ to train the network. We create the experience replay buffer because we can learn more from each experience if we use it to train the network multiple times, which is a more efficient use of the data from the simulated environment. Therefore, we create a buffer and store experiences as we observe them. Then we randomly sample a batch from the buffer and use it to train the network. As we keep observing new experiences, the buffer is updated and the oldest experiences are discarded so as not to exceed the buffer's maximum size.

E. Skipping Frames

In the car simulation, two consecutive frames will look extremely similar to one another. Using each single one for training is repetitive and the agent would thus not learn any new information. To prevent that, we skip frames and capture frames, which represent states, with a low frame rate. This method also provides enough time to do one neural network training pass over a batch of experiences in between collecting frames.



(a) Convolutional Neural Network



(b) Neural Network with Transfer Learning

Fig. 3: Visualizations of the different DQN architectures we use for training the Dueling Double DQN.

V. DQN Architectures

A. Transfer Learning with ResNet50

In the car simulation setup, we observe images (See Fig. 1b for an example image from the virtual front camera on the simulated car) that are used to train the Q-network. Training a CNN on images can be time consuming and requires tedious hyperparameter tuning. Therefore, we use transfer learning [15] with a pretrained ResNet50 [6] on the ImageNet [3] dataset. We cut the pretrained ResNet50 before the last layer. The output is a vector of size 2048. With the transfer learning approach, we use the 2048 vector as the state representation in place of an image which is more memory efficient in the replay buffer. Fig. 3b provides a visualization of this neural network architecture. Fig. 3b shows ResNet50 used as a preprocessing step followed by the fully connected layers. The ResNet50 layer is color coded in gray to represent that it is not trained (the weights are frozen.)

We design two different networks to be used in the transfer learning approach:

1) One hidden layer: This network architecture has one fully connected layer of size 128 with a ReLU [13] activation

right before the two streams branch out for the value and action functions following the dueling method [17]. .

2) No hidden layer: In this method, we pass the 2048 vector directly as input to the two streams for the value and action functions.

B. Training a CNN from scratch

In addition to the transfer learning approach, we also follow the traditional image processing approach and train our own CNN. We use a few convolutional layers to process the image input and follow with a fully connected layer before branching out the value and action function streams. In this method, the replay buffer stores images as states. See Fig. 3a for a visualization of this convolutional neural network architecture. Fig. 3a shows the convolutional layers and a representation of their filters followed by the fully connected layers.

We design two different CNNs:

1) Simple CNN: This approach has two convolutional layers. The first layer has 64 filters each of size $(5, 5)$ with a stride of 2 and ReLU [13] as an activation function. The second layer is the same as the first but with 32 filters. The convolutional layers are followed by a fully connected layer of size 128 with a ReLU [13] activation before branching out into two streams.

2) Larger CNN: This approach uses three convolutional layers of sizes 64, 64, and 32 respectively. All the layers use filters of size $(5, 5)$ and stride 2 with a ReLU activation [13]. Each convolutional layer is followed by a batch normalization layer [9]. The batch normalization layers make the training robust to non-optimized weight initializations. This network is then followed by a fully connected layer with a ReLU [13] activation before branching out into two streams.

VI. Experiments

A. Reward Function

To determine if the car is in the right lane, we add waypoints in the form of (x, y, z) 3D world coordinates in the center of the right lane. Let W be the set of waypoints, car_p be the position of the car in the form of (x, y, z) 3D world coordinates. We define $\underline{d}$ to be the minimum distance from the car to all waypoints:

$$\underline{d} = min_i ||W_i - \text{car}_p||_2 \tag{5}$$

We conduct multiple reward function experiments to find an optimal function. The goal is to encourage minimizing the distance to the waypoints while maintaining a high speed. Initial iterations did not use waypoints and the agent could not learn how to navigate the complex environment. Other iterations did not include the speed in the reward and the agent learned that by staying idle it will accrue more reward.

The final reward function (6) makes use of the minimum distance (5), speed, and Unreal's collision detection system

$$R = \begin{cases} -10 & , \text{ if } \underline{d} > 5 \\ -10 & , \text{ if } t_{\text{idle}} > 2 \\ -100 & , \text{ if collision} \\ \frac{10}{\underline{d}} + v & , \text{ otherwise} \end{cases} \tag{6}$$

Where t_{idle} is the number of consecutive timesteps in which the car is idle and v is the car's velocity.

The collision reward of -100 is significantly lower than other state rewards to signify that avoiding collisions is a top priority. In the reward function (6) if any of the negative reward states are true, then the episode is terminated. The episode is also terminated after $30s$ to avoid infinite loops.

This reward function allowed the agent to learn how to drive in the right lane in a straight line for the duration of the episode.

B. Training in Unreal 4

For each one of the DQN architectures in Section V we run diffent training experiments. We train for 1000 episodes, with a starting epsilon of $\epsilon = 0.9$. Epsilon is decreased after every 100 timesteps by 5%. We do a search over the hyperparameter space of the learning rate and test different learning rate values ranging from 10^{-2} to 10^{-6}. We plot the reward per episode, epsilon, and the smoothed average of the reward per episode to quantitatively analyze the training (See Fig. 4-5). We visualize a few episodes using the trained model to qualitatively analyze the training.

VII. Results

The hyperparameter search over the learning rate space showed that the best learning for training the transfer learning network architecture with no hidden layer is $lr = 10^{-4}$ and the best learning rate for the transfer learning network with one hidden layer is $lr = 10^{-3}$.

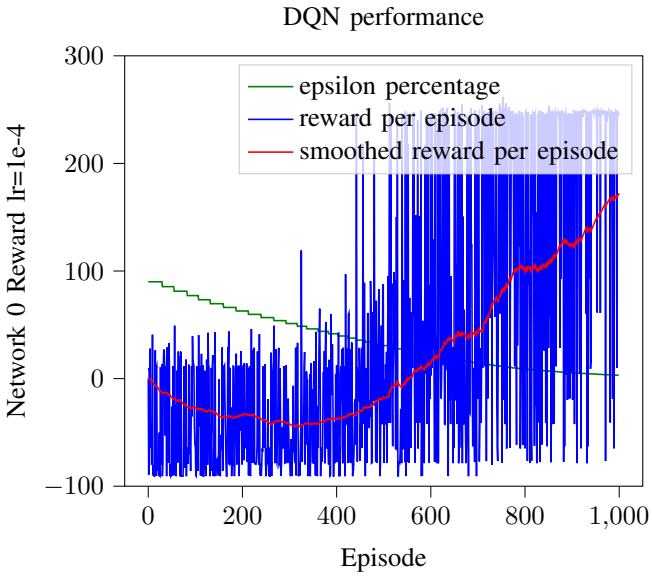
Training the CNNs from scratch showed that a learning rate of $lr = 10^{-5}$ is best for the simple CNN and a learning rate of $lr = 10^{-5}$ is best for the larger CNN with batchnorm layers.

We plot the reward per episode for each network architecture approach in Fig. 4-5 after training for 1000 episodes. In each plot in Fig. 4-5, we can observe the smoothed reward per episode increasing showing that the agent is learning how to interact with the environment. As the agent learns more and the epsilon drops, meaning that the agent is taking less random actions, the reward value increases significantly compared to the first few episodes in the training.

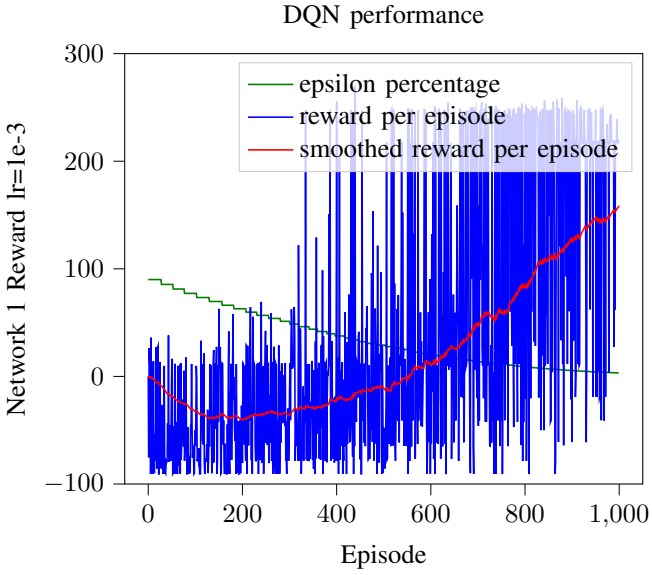
We observe no significant difference in the rewards between the two transfer learning approaches. See Fig. 4a-4b. The two plots show the smoothed reward increasing in the same manner and achieving the same final values. Moreover, adding an extra hidden layer does not improve the performance.

As Fig. 5b shows, when training the CNNs, the larger CNN reaches higher reward values compared to the simple CNN shown in Fig. 5a. The plots show that the smoothed reward reached higher final values for the larger CNN. This can be explained by the fact that the more complex network architecture is able to capture more information from the simulation environment images.

When comparing the transfer learning and CNN approach after training for just 1000 episodes, the transfer learning approach was able to reach reward values above 250 while

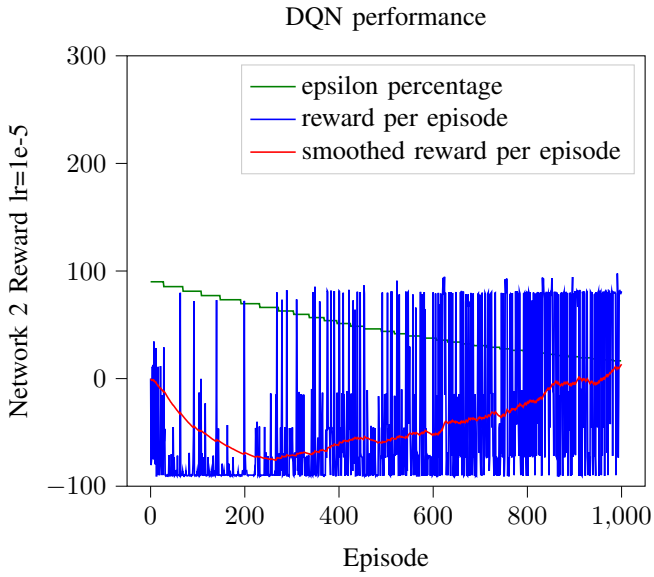


(a) Resnet Network with no hidden layer

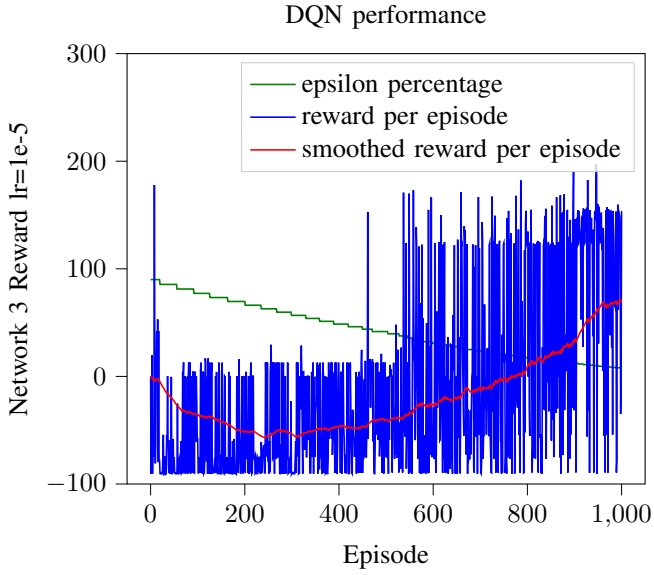


(b) ResNet network with one hidden layer

Fig. 4: Reward per episode plot for the transfer learning method



(a) Simple CNN



(b) Larger CNN

Fig. 5: Reward per episode plot for the CNN method

the CNN approach only reached 200 at best using the same reward equation 6.

Testing each trained network from each approach by using the trained network to drive along the street shows that the transfer learning approach with no hidden layer collides one out of ten times. The transfer learning approach with one hidden layer does not collide during the ten runs. Both CNN approaches collide one out of ten times. The simulated car keeps a more stable speed when running with the transfer learning approach as opposed to the CNN approach. In the CNN approach, the car speeds up and slows down randomly.

VIII. Conclusion

Overall, our results show that of the approaches taken, transfer learning is the most optimal. This can be explained by the fact that using a pretrained ResNet50 [6] as a preprocessing step means less training is needed for the network to capture the interactions with the complex environment. The CNN approaches may perform as well as the transfer learning approach with more training time. Using transfer learning

allows us to skip a large portion of the training that would be necessary if done from scratch.

A. Future Work:

In this project we were able to train an autonomous vehicle in simulation how to drive down the road using a dueling double DQN. Future work includes setting up waypoints in a right turn and left turn and train with those waypoints to determine if the agent can learn other maneuvers. To make the training more robust, we plan to retrain on diffent town maps with the same network. In addition, setting up waypoints over a larger area of the map and increasing the time limit per episode would improve the performance of the trained agent.

B. Code Implementation

All the implementation for this project can be found at `github.com/ritabt/SeniorProject_DDDQN` Follow the ReadMe for setting up a virtual environment with all the dependencies. The `Duel_DDDQN` folder is a package that can be used with any Gym [2] environment, even custom ones, to train with Dueling Double DQN. The entire repository can be placed in the PythonClient folder of the Airsim [14] starter package to get started training an autonomous vehicle. The code used in the repository is partially a modification of the implementation by Huan [8] and a modification of the Airsim [14] source code.

References

[1] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. 2016.

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Li Kai, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[4] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Conference on Robot Learning (CoRL)*, pages 1–16, 2017.

[5] Epic Games. Unreal engine, version 4.22.1, 2019-04-25.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[7] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[8] Chua Cheow Huan. Dueling ddqn. https://chuacheowhuan.github.io/Duel_DDQN/, Mar 2019.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, page 448–456, July 2015.

[10] Taisuke Kobayashi and Wendyam Eric Lionel Ilboudo. t-soft update of target network for deep reinforcement learning. In *Neural networks: the official journal of the International Neural Network Society*, volume 136, pages 63–71. Elsevier BV, apr 2021.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, and et al. Human-level control through deep reinforcement learning. Nature Publishing Group, Feb 2015.

[13] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning (ICML)*, June 2010.

[14] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[15] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of Research on Machine Learning Applications*, 2009.

[16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2016.

[17] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2016.

[18] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, 1989.