Learning to Drive with Reinforcement Learning and Variational Autoencoders

Bryon Kucharski
University of Massachusetts Amherst
bkucharski@umass.edu

Abstract

The use of deep reinforcement learning (RL) for following the center of a lane has been studied for this project. Lane following with RL is a push towards general artificial intelligence (AI) which eliminates the use for hand crafted rules, features, and sensors. A project called Duckietown has created the Artificial Intelligence Driving Olympics, which aims to promote AI education and embodied AI tasks. The AIDO team has released an open-sourced simulator which was used as an environment for this study. This approach uses the Deep Deterministic Policy Gradient (DDPG) with raw images as input to learn a policy for driving in the middle of a lane for two experiments. A comparison was also done with using an encoded version of the state as input using a Variational Autoencoder (VAE) on one experiment. A variety of reward functions were tested to achieve the desired behavior of the agent. The agent was able to learn how to drive in a straight line, but was unable to learn how to drive on curves. It was shown that the VAE did not perform better than the raw image variant for driving in the straight line for these experiments. Further exploration of reward functions should be considered for optimal results and other improvements are suggested in the concluding statements.

1. Introduction

Autonomous vehicles are one of the most researched topics today and have shown a large amount of progress due to the recent advances in computer vision and deep learning. Typically, these systems use a variety of sensors to perceive their surroundings, such as radar, computer vision, Lidar, sonar, GPS, odometry and inertial measurement units.

One argument for reinforcement learning (RL) is that it eliminates a lot of the complexity of current systems, and relies on a more natural learning experience. Consider a child learning how to ride a bike. There is not a list of hand-crafted rules for him or her to follow and no sensors attached to the bicycle. All he or she is given is a safe environment to test different ways to ride the bike. After enough trial and error, he or she is able to successfully stay balanced. This is the approach that RL applies to self-driving cars. Regardless whether one agrees with this approach or not, it is worth studying to explore new approaches to progress autonomous vehicle research.

A major factor in any learning algorithm is the curse of dimensionality made famous by Richard Bellman. While exploring different topics in dynamic programming, Richard explains that as the number of dimensions grows, there is a significant increase in the amount of data and computation needed to explore the complete state-action space. This ”curse” is still relevant in many aspects of learning today. The field of reinforcement learning has experienced this curse as well, as one of the most common algorithms, Q-Learning [10] works particularly well in smaller dimensional state spaces. Methods have been developed called function approximation to address these issues, but not fully eliminate the issue. Even more relevant today, many deep reinforcement algorithms are learning off of pixels directly instead of hand crafted features. Thus, research has led to algorithms to handle continuous state (DQN) [5] and continuous actions (DDPG) [4] spaces. While these handle these continuous spaces well in some situations, there are still many situations which still suffer from the curse of dimensionality. Maybe it is possible to use another dimensionality reduction method, a Variational Autoencoder (VAE), to compress a state space as input to a continuous algorithm. This project aims to study the use of VAEs and it’s importance in training a deep reinforcement learning algorithm.

2. Background/Related Work

2.1. Reinforcement Learning

A reinforcement learning agent learns from exploring the environment and has a goal to find a policy π that maximize the reward signal given from the environment. The environment and agent are modeled mathematically using a Markov Decision Process (MDP). An MDP consists of

- State set S: - all possible states in environment
- Action set A: - all possible actions the agent may choose
- Reward function R: - reward for entering state $s^{'}$ by taking action a in state s
- Initial state distribution: d_0 - probability of starting in each state
- Discount factor γ - how far into the future to consider rewards
- (Optional) Transition Matrix P: describes the probability of transitioning between states item

The state-value function $V^{\pi}(s)$ is scalar measure of how good it is for the agent to be in state s when using policy π. The action-value function $Q^{\pi}(s, a)$ is a scalar measure of how good it is for the agent to take action a in state s and follows the policy π after. Many times in RL, the state-value function and/or action-value functions are not directly known for every state or state-action pair. In this usual situation, a function approximator is used to estimate these values. Linear function approximators may be used such as the Fourier Basis and Polynomial Basis [3], but neural networks may also be used as non-linear function approximators. This is known as deep reinforcement learning.

Another important aspect of RL used in this project is the Temporal Different (TD) error, which is most commonly defined at time t as

$$\delta = target - prediction$$

$$\delta = (R_t + \gamma V(S_{t+1})) - V(S_t)$$

The TD error is a measure of what actually happened $R_t + \gamma V(S_{t+1})$ between what the agent predicted happened $V(S_t)$. A positive TD error means there was a better outcome than expected, and a negative TD error means there was a worse outcome than expected. The TD error is essential in training the networks for the Deep Deterministic Policy Gradient (DDPG) algorithm.

2.1.1 Deep Deterministic Policy Gradient

DDPG is a deep RL algorithm introduced by DeepMind [4] used to solve continuous state and continuous action based MDPs. It is an actor-critic based algorithm, where an actor network uses the current policy π is to predict the continuous actions given an input state. The critic network uses the output of the actor network to estimate the $Q^{\pi}(s, a)$ function.

The training of the DDPG algorithm is done online. At every iteration, the agent receives state s, takes action a, receives reward r, and ends up in a new state s', These experiences are stored as a tuple $(s, a, r, s^{'})$ (also known at experience replay). To update the weights of each network, these experiences are randomly sampled to create a batch of experiences. One of the key aspects for this algorithm to work is the idea of using target networks to estimate the target values in the TD error. Thus, there is actually four networks trained during training: critic $Q(S_t, A_t)$, critic target $Q^{'}(S_t, A_t)$, actor $u(S_t)$, and actor target $u^{'}(S_t)$. During the training process, a batch of $(s, a, r, s^{'})$ are given and the targets are calculated with

$$target = r + \gamma Q^{'}(s, u^{'}(s))$$

The critic network is then updated by minimizing the mean squared error between the critic network and the target network

$$L = MSE(target, Q(s, a))$$

The policy network is then updated by taking the policy gradient [8] (PyTorch pseudocode)

$$\nabla J = -Q(s, u(s)).mean().backwards()$$

Both the weights of the targets $\theta^{Q^{'}}$ and $\theta^{u^{'}}$ are updated to match the weights of the non-target networks θ^{Q} and θ^{u}

$$\theta^{Q^{'}} = \tau\theta^{Q} + (1 - \tau)\theta^{Q^{'}}$$

$$\theta^{u^{'}} = \tau\theta^{u} + (1 - \tau)\theta^{u^{'}}$$

where τ is a hyperparameter

Any reinforcement learning agent is required to explore the environment to learn. If part of the state-action tuples are not explored, the agent will not have complete knowledge of the environment. DDPG uses an exploration method called Ornstein Uhlenbeck process [9]. The idea behind this process for RL is to add stochasticity to every action being selected.

2.2. Variational Autoencoders

A vanilla autoencoder is a relatively simple idea where one can compress input data into a smaller dimensional z vector using a neural network without losing much information about the original image. To train the network, one could decode the z vector back into the original data using a neural network, and compare the difference between the original data and the decoded data as a loss function. This is called the reconstruction loss. A variational autoencoder is an extension to the vanilla autoencoders. Instead

of encoding into a single z vector, a mapping is created to a distribution where one vector represents the mean of the distribution and one vector represents the standard deviations. To get the z vector, a sample is taken using these two vectors as

$$z \sim q(\mu, \sigma)$$

The loss function is modified to have the same reconstruction loss as the vanilla architecture. A second term is added to the loss function called the KL divergence which is used to ensure the distribution is relatively close to a normal distribution. There is one more detail called reparameterization which is needed because it is not possible to differentiate a sampled distribution. Instead of sampling from the distribution directly, the z vector is constructed using the formula

$$z = \mu + \sigma * \epsilon$$

where μ and σ are learned parameters and ϵ is a normal distribution. Using this trick, the function is differentiable but still include the desired stochastic node.

2.3. Duckietown

Duckietown is a project that started at MIT [6]. It began as a course to teach different aspects of self driving cars. It consists of a town made of rubber mats and Raspberry Pi based robots called DuckieBots. The DuckieBots traverse the town without hitting other DuckieBots or the rubbery-duckie inhabits of the town. It has been a rapidly growing project, which now focuses on teaching embodied AI. The Duckie Foundation has organized the first AI Driving Olympics (AIDO), which took place at NeuralIPS in December 2018. I missed the deadline to submit for the competition, but the second round of the AIDO are taking place in 2019 at ICRA which I may be interested in submitting too. This competition has four main tracks

- Lane Following
- Lane following with other vehicles
- Navigation with Dynamic Obstacles
- Autonomous Mobility-on-Demand

I chose the Lane Following task for my project. There is one simple goal for this challenge: stay in the middle of the lane at all times.

The AIDO team released an open source simulator for the competition. The simulator is created with Python and PyGlet. It is Open-AI Gym compatible which makes it easy to use. There are YAML files which can describe different map configurations as well was wrapper classes for more customization such as resizing observations, creating a discrete action set, etc. Figure 1 shows example screenshots of the simulator.

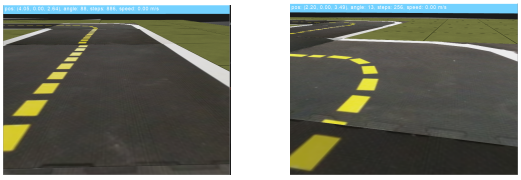


Figure 1: Examples of Gym-Duckietown simulator

2.4. Related Work

The idea for combining DDPG with VAEs with was exposed to me from a paper by a United Kingdom based self-driving car company named Wayve [2]. In Wayve's experiment, the authors detailed an experiment to drive 250km using DDPG with convoluted images using CNNs versus DDPG with encoded images using a VAE. In their experiment, they showed VAE vastly decreased the amount of training episodes to reach the 250km goal.

The Gym-Duckietown [1] team provides a baseline implementation of reinforcement learning algorithms including ACKTR, Proximal Policy Optimization (PPO), and Asynchronous Actor Critic (a2c). The AIDO team also released a baseline implementation of DDPG which does not work of the box.

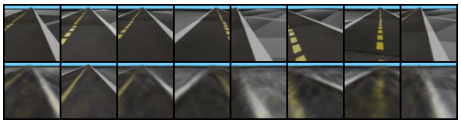
3. Approach

The main idea of this project is rather simple - determine if I could improve an agent's policy or an agents training time by encoding an the state image into a smaller dimensional vector, and use that as the new state representation. The MDP was constructed as

- S (continuous) - all possible images
- A (continuous) - One variable between [0,1] which controls how much velocity to apply to the DuckieBot. A second variable in the range [-1,1] which controls the steering angle. -1 is all the way left and +1 if all the way right.
- R: Varies in each experiment
- d_0: Experiment 1 - Agent always starts in the middle of the lane at the beginning of a straight track.
  Experiment 2 - agent is randomly placed on the map in the middle of the right lane.
- γ: 0.99
- P: Not used (Model Free)

I have used fully connected layers for all networks in this project. The hyperparameters and network size for DDPG follow the original paper [4]. For the VAE, a fully connected network with one layer and 400 units was used for both the encoder and decoder. The input images where size 3x64x64

(a) Epoch 1



(b) Epoch 25

Figure 2: Example Reconstructiton of VAE from start of training to end of training

and encoded into a 1x100 vector which drastically reduces the dimensionality of the input data. A dataset was collected to train the VAE by manually controlling the DuckieBot around the environment and collecting images at every update. The VAE was trained offline from the DDPG training on a dataset of 10,000 training images and 2,000 test images which are included in the supplementary material. When DDPG training began, the pre-trained VAE model was used to encode the input image. Algorithms 1 and 2 show the main training loop of the DDPG algorithms. The .train() method is the process described in the background section of this report. As Algorithm 2 shows, the only different in the training loop was to add the extra encoding step of the input image. Figure 2 shows comparisons of the input image and reconstructed image.

Algorithm 1 Training Loop of Deep Deterministic Policy Gradient with Raw Images

procedure DDPGRAW(ENV,DDPG)
  $s \leftarrow$ env.reset().flatten()
  steps $\leftarrow 0$
  while *steps* < 20000 **do**
    $a \leftarrow$ ddpg.selectAction(s)
    $r, s^{'}$ *done* $\leftarrow$ env.step(a)
    $s^{'} \leftarrow$flatten($s^{'}$)
    ddpg.remember($s, a, r, s^{'}$)
    ddpg.train()
    steps $\leftarrow$ *steps* $+ 1$
    if *done* **then**
      $s \leftarrow$ env.reset().flatten()
    else
      $s \leftarrow s^{'}$

Algorithm 2 Training Loop of Deep Deterministic Policy Gradient with VAE

procedure DDPGVAE(ENV,DDPG, VAE)
  $s \leftarrow$ env.reset().flatten()
  $s \leftarrow$ vae.encode(s)
  steps $\leftarrow 0$
  while *steps* < 20000 **do**
    $a \leftarrow$ ddpg.selectAction(s)
    $r, s^{'}$, *done* $\leftarrow$ env.step(a)
    $s^{'} \leftarrow$flatten($s^{'}$)
    $s^{'} \leftarrow$ vae.encode($s^{'}$)
    ddpg.remember($s, a, r, s^{'}$)
    ddpg.train()
    steps $\leftarrow$ *steps* $+ 1$
    if *done* **then**
      $s \leftarrow$ env.reset().flatten()
      $s \leftarrow$ vae.encode(s)
    else
      $s \leftarrow s^{'}$

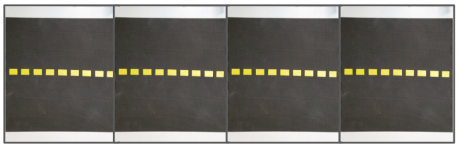


Figure 3: Gym-Duckietown map for experiment 1

4. Experiments

4.1. Experiment 1

The first experiment I set up was to have the agent learn to drive in a straight line. The map used is shown in Figure 3. To start, the agent was placed in the middle of the lane. The optimal policy would be to just drive straight without moving the steering wheel. At every time step, the agent receives a reward computed by the reward function. A reward of -10 is given for driving off the road.

4.1.1 Reward Function Exploration

The original simulator had a reward function as follows

$$R(s, dot, d) = (1.0 * s * dir) + (-10 * |d|)$$

where s is the speed of the robot, dir is the dot product of the robot (where 1.0 is facing forward), and d is the distance to the middle of the lane. Simply put, the agent should stay in the center of the lane and face forward. I trained the agent using this baseline reward function and observed an interesting result. When the agent was going to drive off center and stop receiving the positive reward, it stopped in its tracks and didn't move at all to still collect positive rewards. This revealed a fault in the original reward function.

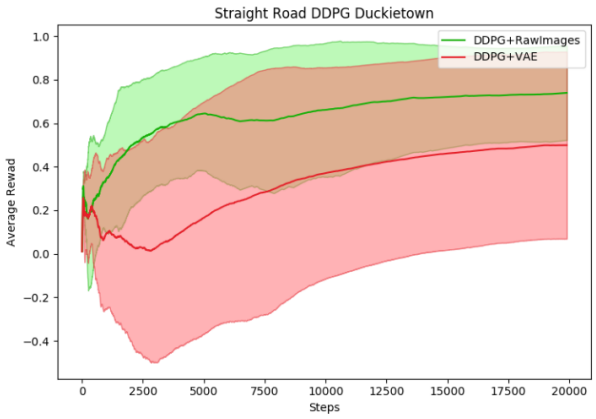


Figure 4: Experiment 1 Results. The plot shows the average reward at every iteration of the training sequence, averaged over 10 trials. The ideal average reward would approach 1.0

There is not enough emphasis on making forward progress. Thus, I added another term $movePenalty$

$$R(s, dot, d) = (1.0 * s * dot) + (-10 * |d|) + movePenalty$$

where

$$movePenalty = \begin{cases} 0 & \text{if } \Delta \text{ x} > 0.02 \text{ cm/s} \\ -1.5 & \text{else} \end{cases}$$

Δ x is the change in distance of the DuckieBot. The threshold 0.02 cm/s was a parameter chosen by trial and error.

The agent was able to achieve the desired results with this reward function for both the DDPG+VAE and DDPG+Raw. See the video in the supplementary for both working. Upon careful examination of the policy, an interesting fact is the agent actually learned a near discrete policy for both methods. As seen in the video, there is 'jerky' steering motion which is something not expected from a continious control algorithm. It learns to always drive forward with max velocity, which is expected based on the reward function, but it also learns to steer left and right with max value (1 or -1).

4.2. Experiment 1 Discussion

Figure 4 details the average reward of the agent over 20,000 training steps. That is, every iteration of the training loop, the reward for current transition from s to $s^{'}$ was added to the current total reward, divided by the number of steps, and recorded in the plot. The reward to start is expected to be low as the agent has not trained, and the ideal number to approach would be around 1.1 as this is about the max reward the agent may receive based on the reward function detailed in the previous section. The reward is averaged over 10 trials and shows standard deviation error bars.

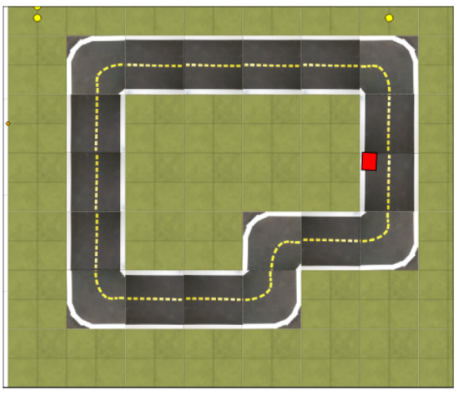


Figure 5: Gym-Duckietown map for experiment 2

As Figure 4 suggests, DDPG with raw images as input does not out perform DDPG with VAE. While my data proves otherwise, I still support the original hypothesis that DDPG+VAE would outperform DDPG+Raw. I feel as if my setup and experiment is not enough to disprove this hypothesis. There are a number of aspects which contribute to my final result. Mainly, I am thinking that the set up may be 'easy' enough for the DDPG with raw images to learn. In order for the agent to achieve max reward, all it needs to do it look forward and drive straight. Thus, there are limited frames that the agent needs to learn from. I would think that VAE might help significantly when there are different types of roads involved - straight, left curve, right curve, right turn, left turn, etc. This is a factor that caused me to explore a new experiment. In addition to being 'easy' enough to learn with just raw images, there are other factors where my implementation could be improved, which are outlined in the conclusion.

One aspect that was slightly better with the DDPG+VAE was the training time of the agent. Since the input dimensions where significantly smaller, the agent was able to train slightly faster than the raw image variant. If you factor in the time to collect the images and train the VAE, then there is not much significance for using the VAE based model. The raw image variant still used relatively small images of 3x64x64, so there should be a more significant speedup if the images are larger.

4.3. Experiment 2

For the second experiment, I wanted train my agent on the same map as the the official AIDO lane following competition with the hope of working towards a submission for the competition. The map is shown in Figure 5 and contains both left and right turns in addition to the straight away roads.

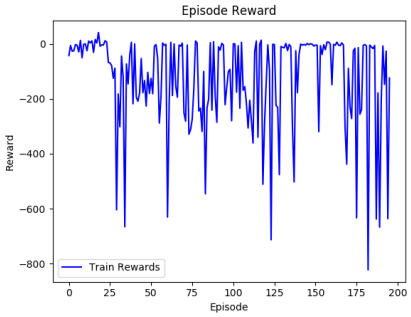


Figure 6: Training reward for a single trial of experiment 2. It is shown here that the agent is never achieve a positive reard. A positive reward corresponds with staying the in the middle of the lane. The agent stays in the middle of a lane for the straight away sections, but not for the curved sections of the map. See the supplementary video for a demonstration of this.

4.4. Experiment 2 Discussion

This experiment did not go as planned and the agent was not able to learn the curves on the map. I began by using the same reward function as experiment 1, where I encourage the agent to be moving forward, looking straight, and in the middle of the lane for a positive reward. The biggest issue with this experiment was the fact that there are sparse positive rewards when the agent is at a curve. What happens is the agent is encouraged to be moving fast from the $movePenalty$ in the reward function, so when it reaches the curve it must slow down to 'physically' be able to stay in the center of the curve. Without slowing down, the agent will be moving to fast, and wont turn quick enough. This makes my modification of the original reward function counter intuitive because now the agent needs to drive slow to make it past the curve. The agent actually learns that it will get less negative reward if it drives off the road instead of attempting to stay in the middle because the cumulative negative reward when attempting to stay in the middle of the lane is more than the -10 for driving right off. Thus, as seen in Figure 6, the best policy learned is one that receives an average reward of about 0. There is video of this in the supplementary submission.

I have spent the majority of this project trying different reward functions to guide the agent to learn how to drive on the curves. To begin, I got rid of the $movePenalty$ and observed the same issue where the agent is not encouraged to make forward progress. I have tried a lower threshold of Δx described in the first experiment. I have tried only incorporating the $movePentalty$ when the agent is not on a curved tile, giving it the option to slow down on curves. I have tried giving the agent a lot larger negative reward to prevent the agent from learning to drive off the road. All of these have not been successful in learning the curved part of the map. From this, it became apparent that I need to find a mixture of encouragement for forward progress, but not too much emphasis so the agent could still learn the curves.

As I mentioned in experiment 1, the agent actually learns a near discrete policy, meaning that the actions it takes are always either exactly or close to -1, or 1 for the steering angle and 1 for the velocity. It makes sense for the velocity to be max because it is rewarded for making forward progress. It does not make clear sense why the steering angles are near discrete. This raised more alarm for the second experiment, because the agent must use continuous actions on the curved part of the road. I feel an explanation of why the steering angles are near discrete will help solve some of the issues I've had for experiment 2. My initial reasoning for this is maybe the models are too shallow and are doing something equivalent to over fitting, where the agent is not able to learn an optimal policy due to the restriction of the network themselves, not the reward function. Further examination of my models is required to get to the root of this observation.

A VAE was not trained for this experiment because I was unable to successfully train the agent with use images and I would not have anything to compare it too. It may work better than raw images, but I infer the reward function is a bigger issue than the state representation.

5. Conclusion

After the completion of this project, I have gained insight on how difficult it is to get RL applications to work well. Most of my time was spent trying to tune the reward function. I have a list of improvements that are suggested as future work.

- Different network architectures - I used fully connected networks for all the architectures. I would think CNN architectures may be better at creating features for state representations.
- Tuning Networks - Since most of my time was spent on the reward exploration, I did not change any parameters at all. I followed the paper in the original DDPG paper [4]. A hyperparameter search may prove to be beneficial to find parameters that work best for my problem instead of all the problems in the paper.
- More training images for VAE
- Different Algorithm - Maybe an algorithm like PPO may be able to learn a better policy?
- Linear Function Approximation - Deep reinforcement learning has proven to be difficult to tune and work well. Maybe I could receive similar or better results using a different function approximator than a neural network.

- Wayve explains the use of prioritized experience replay [7], which is a method to improve on randomly sampled tuples of experiences during RL training and is based on sorting the tuples. This may improve performance of both of my algorithms.
- Exploring different Ornstein-Uhlenbeck process parameters to encourage, discourage more/less exploration
- Other dimensionality reducing methods instead of VAE. Maybe something like PCA?

As for the AIDO competition, I have made the decision not to submit this work. It became apparent to me as I progressed through the project how difficult it is to get a perfectly working model using reinforcement learning. If I was to continue with this work for the submission, I think I would rather go towards the track of imitation learning. While this would introduce a wide range of new problems, I think intuitively it moves more sense to "show" the robot how it should drive on the road rather having it learn from scratch. I even think classical control methods may work better or just as good as any machine learning based algorithm. Although I will not submit to this competition, I am glad I got to express two interests of mine in reinforcement learning and variational autoencoders.

The supplementary documents for this report include the training set for the VAE, a video of experiment 1 working properly for both DDPG+Raw and DDPG+VAE, and a video of experiment 2 not working properly. The code has been posted to GitHub (Click for link).

References

[1] M. Chevalier-Boisvert, F. Golemo, Y. Cao, B. Mehta, and L. Paull. Duckietown environments for openai gym. `https://github.com/duckietown/gym-duckietown`, 2018.

[2] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah. Learning to drive in a day. *arXiv preprint arXiv:1807.00412*, 2018.

[3] G. Konidaris, S. Osentoski, and P. S. Thomas. Value function approximation in reinforcement learning using the fourier basis. 2011.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[6] L. Paull, J. Tani, H. Ahn, J. Alonso-Mora, L. Carlone, M. Cap, Y. F. Chen, C. Choi, J. Dusek, Y. Fang, et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 1497–1504. IEEE, 2017.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[8] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[9] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[10] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.