National University of Singapore

ESP3201 Machine Learning in Robotics and Engineering

Project Report

Sim-to-Real Transfer using Webots and Anki Vector Robot

Group 4

He Cheng Hui

Foo Fang Wei

Contents

1. Abstract

This project aims to use Reinforcement Learning techniques and robotic simulation to produce a neural network model that can also be used in real-life. Deep Q-Learning or DQN was selected as the Reinforcement Learning Algorithm while Webots was chosen as the software to simulate the training and testing process. The robot used in this project is a tracked wheel human companion robot called Vector, made by Anki. It was found that using a single frame as the state representation seems to yield the best result with the agent able to complete the training track while other methods failed to even complete one round. It was also found that using a white backdrop in the simulation environment results in faster convergence rate while also allow easier replication of environment in real-life, which results in a successful Sim-to-Real transfer.

2. Introduction

Autonomous vehicles are transforming the way we live, work, and play. It allows for safer roads and more efficient means of transportation.

Imagine getting into a car, providing a location into the vehicle's interface, then letting it drive itself to the destination while you read a book, surf the web, or nap. The field of self-driving vehicles have seen much advancement, especially from companies like Tesla and Google. Such companies are going to radically change what it is like to get from point A to point B.

Autonomous vehicles could also provide benefits for the disabled and elderly such as more mobility and independence. For businesses, it holds great potential for freight transportation and utility services sectors. For instance, deploying autonomous systems at night instead of during the daytime could ease traffic congestion during peak hours and reduce the need for drivers.

Most self-driving methods learn directly from mapping images to driving behaviours, which could potentially have low generalisation abilities. To improve generalisation ability, reinforcement learning (RL) methods are used in conjunction with a simulated environment to train the model, and then have the model transferred to the actual car to be driven.

3. Objective

This project was inspired by the combination of robotics and RL to create a self-driving robot vehicle, where we could explore the potential of self-driving technology. To realize this goal, a simulation software Webots was used, to serve as a virtual environment where a virtual robot from Anki called Vector, would learn how to drive by itself around a

marked course. This project also attempts to use RL techniques to teach the Vector agent how to drive around a training course, before testing it on three similarly themed courses that the model has never seen before, to test its generalisation ability. Finally, the passing model would then be used with the actual Vector robot to drive around the training course built in the real-world.

4. Deep Q-Learning (DQN)

Q-learning is an off-policy RL algorithm that aims to find the best action to take given the current state. It is considered off-policy as the Q-learning function learns from actions that are outside of its current policy. This involves taking random actions, and hence a policy isn't needed. Specifically, Q-learning aims to learn a policy that maximizes the total reward it gets.

Essentially, a main part of Q-learning is forming the q-table and using the bellman equation to update that table. However, this becomes computationally intensive if the environment becomes more complex as there can be many permutations of state action pairs. This is also an issue if the state is represented by an image instead of a simple numerical value.

To overcome this issue, a deep Neural Network (NN) can be used to process these image inputs to represent the q-table, while the weights could be iteratively based on the bellman equation. Therefore, by integrating Q-learning and deep NN it can be referred to as Deep Q-learning or DQN algorithm.

Even though there are many variants of DQN, this project focus on a basic DQN structure of three convolution layers and two fully connected layers to train the Vector agent.

5. Reinforced Learning Framework

The RL framework employs the use of Webots as the simulation software and Pytorch to handle the NN. Webots is an open-source robot simulation software that was chosen due to its multi-platform capabilities and ease of use compared to other software like Gazebo [1]. For training, Pytorch was chosen as it is a popular pythonic open-source machine learning library [2]. This section explains how training a DQN will be implemented, and structure of the DQN used in this project.

5.1. Overall Framework (high-level)

The training code used in this project bridges the virtual environment in Webots and the NN using the Pytorch library, as shown below.

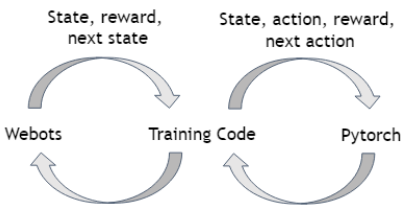


Figure 1: High-level view of training code

As seen in Figure 1, the training code will extract state inputs as image tensors from the virtual robot's camera, while also getting the rewards and next state image tensors after executing an action. This information acts as the input into the DQN built in Pytorch, which then outputs an action. Finally, the training code would communicate with the agent to take what action, and the agent will then execute that action in Webots, starting the cycle again until a termination state of 75 thousand episodes.

However, due to the time frame of this project, training is typically terminated before 75 thousand episodes upon our discretion. The model is judged to be good enough when it completes one round around the track and stays on track, while the opposite is judged based on the reward obtained per episode and the prospects of it improving.

5.2. Episode Framework (low-level)

With the high-level overview in mind, the following will describe the events that happens within an episode.

As seen in Figure 2, starting from the top left, the Vector robot model will first capture an image of the state it is in. State inputs are

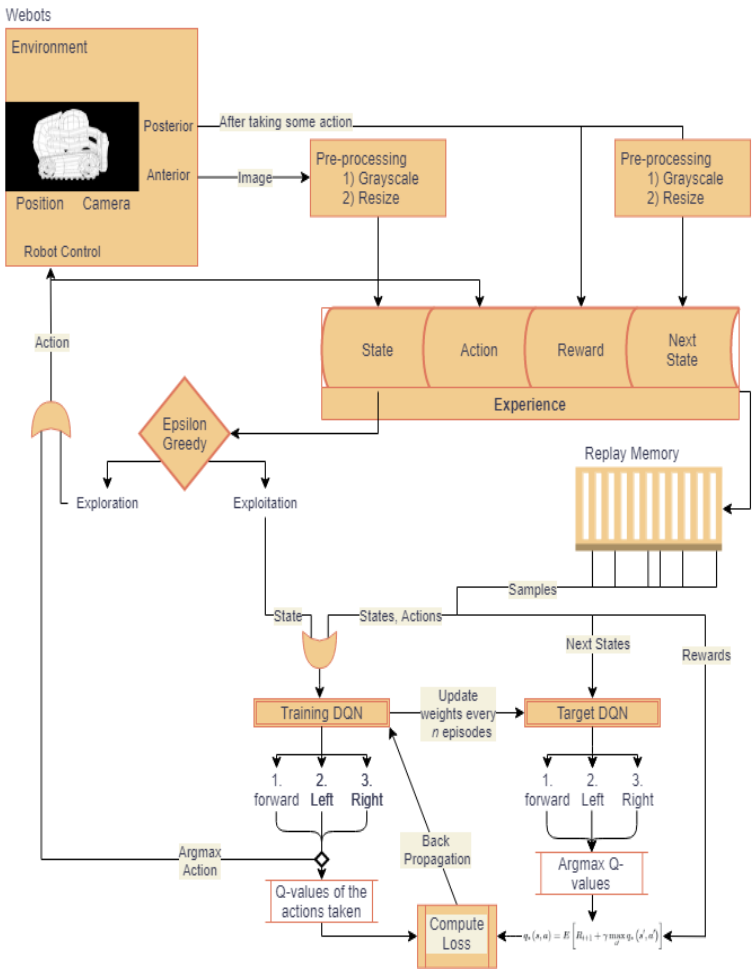


Figure 2: Flow chart of events happening within an episode

obtained through simulating the robot's camera and getting images from that virtual camera. To reduce computation cost, the image undergoes gray-scaling and resized from 640x360 to 64x36 to maintain its aspect ratio.

Subsequently, using Epsilon-greedy strategy, it will choose an action based on either exploration or exploitation. The epsilon will start from one and then decrease exponentially according to Formula 1,

$$e_{end} + (e_{start} - e_{end})^{-e_{decay}*t} \quad (1)$$

Where e_{end} is the minimum epsilon value and is set to 0.01, e_{start} is the starting epsilon value and is set to 1, e_{decay} is the epsilon decay factor that would be further explored and t is the timestep since the start of the training process.

Thereafter, the agent will receive an action signal, and the virtual robot will carry out that action and get some reward according to the pre-defined reward function by looking at its position. Subsequently, the agent will then capture the next state it is in after performing that action.

After pre-processing the image for the next state, this concludes one experience cycle and an experience tuple of (state, action, reward, new state) will be stored in the replay memory. A replay memory or experience replay is a container of certain size that stores the most recent experiences. Once it is full, training of the NN begins and the oldest experience is discarded to make space for new ones. At every training timestep, a random batch of experiences are sampled, and the purpose of uniform sampling is to mitigate temporal correlations and improve training efficiency [3]. For this project, a replay memory size of 100,000 was chosen while the batch size will be explored.

After sufficient episodes have passed to store enough experiences in the replay memory, the training process is initiated. In this process, states and actions are sent to the training network which will be continuously updated, while next states and rewards are used for the target network will synchronise with the training network after some episodes.

The purpose of an additional target network is to overcome the problem when using the same network for training and updating. Similar to why training data is uniformly sample, using the same network could lead to high correlation when training, resulting in frequent sub-optimal convergence and unstable training [4]. In this project, the number of episodes between each synchronisation was also explored.

Once the Q values of the actions taken and the target Q value from the bellman equation

are obtained, the loss for an episode is computed using the Huber Loss algorithm, which combines the benefit of Mean Square Error and Mean Absolute Error. Finally, back propagation based on the Adam algorithm is used to update the weights in the training network and mark the end of one timestep. This would continue until a termination state is achieved and that would be the end of one episode. The termination states would be expressed in the Rewards Function section.

6. DQN structure

In this project, a basic DQN structure implemented consists of three convolution layers, followed by two fully connected layers and lastly an output layer as shown below.

As seen in Figure 3, the convolution layers have a kernel size of eight and stride of four, followed by a kernel size of four and stride of two, and lastly a kernel size of three and stride of one. The fully connected layers would have size of 128, then 32 and finally three to represent the three action outputs for the robot; forward, turn left and turn right. Between each hidden layer would consist of a rectified linear (ReLU) activation function.

Since this basic DQN structure involves stacking three convolution layers, this project also attempts to explore the potential of using residual neural network (ResNet) to replace the convolution portion.

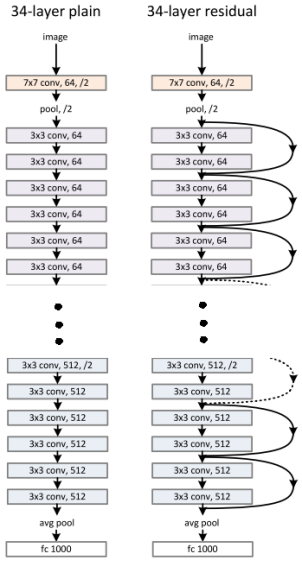


Figure 3: Comparison between stacking 33 convolution layers (plain) vs ResNet version

As seen in Figure 4, the left is a plain deep NN with 33 convolution layers and one fully connected layer as the output. On the right shows the ResNet implementation with an

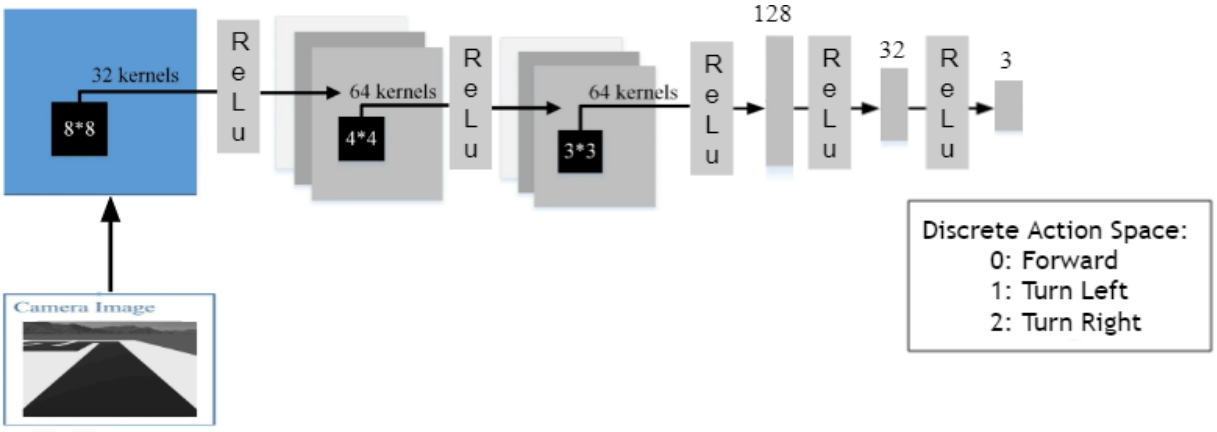


Figure 4: Basic DQN structure with 3 convolution layers, 2 fully connected layers and an output layer of size 3

arrow connecting the input from one convolution layer to the output of another.

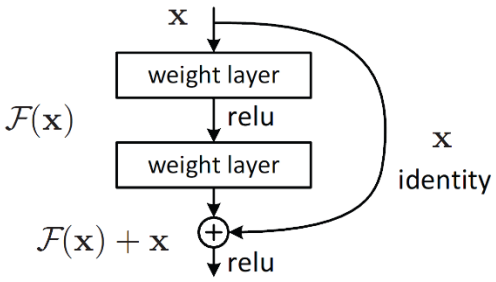


Figure 5: Diagram of an individual residual layer connected by an identity X

As seen in Figure 5, the arrow can be represented by an identity X, or a "shortcut" [5]. The convolution layers connected by an identity X creates one residual layer and a few of them would create a residual block, as depicted by the difference colour group in Figure 4. Thus, allowing ResNet to easily build even deeper networks.

ResNet is useful for training deep convolution networks as using conventional stacking method could produce more training error in practice as shown below.

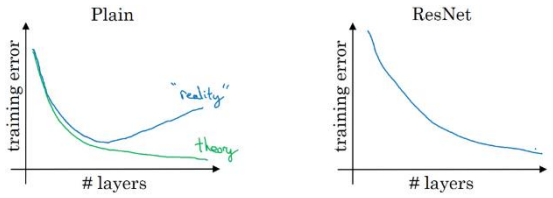


Figure 7: Graphical depiction of training error between using conventional stacking methods vs ResNet stacking method

As seen in Figure 6, while in theory deeper convolution network should produce smaller training errors, degradation of accuracy starts happening after a certain number of layers, causing higher training error [5]. However, ResNet stacking method was proven to decrease training error the deeper it is built [5]. Thus, a variant that uses ResNet50 in place of the three convolution layers was used in this project. ResNet 50 was chosen due to its much better accuracy over shallower ResNet like ResNet 18 and 34 [5], while still maintaining reasonable training speeds.

7. Training Environment

Figure 6: Example of the environment in Webots as seen from a top-down view

In this project, a 50 by 50cm track with 8 cm wide paths is used to train the robot model with a 7 by 10cm wheel base.

7.1. Reward Function

The reward function implemented in this project is as follows:

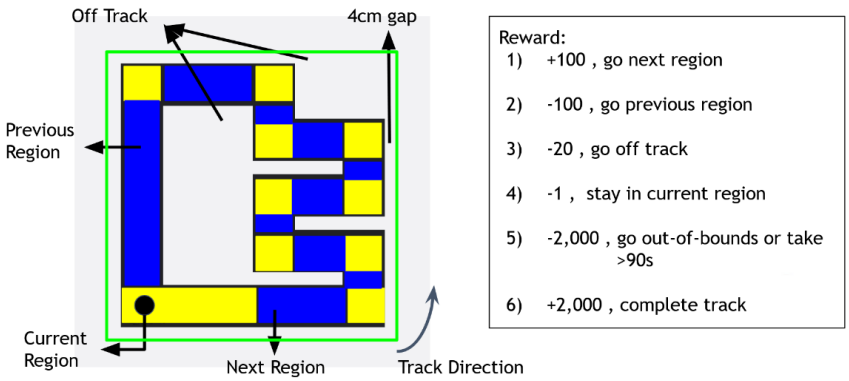


Figure 8: Graphical representation of the reward function according to the agent's position in the environment

A seen in Figure 8, the initial reward design was defined by splitting the track into 20 regions to allow for easier tracking of the path the agent took. If the agent goes to the next region, it will be rewarded by 100. If it goes back to its previous region, it will be penalised by 100. If the agent goes off track, it will be penalised by 20. If it stays in the current region, it will be penalised by 1 to encourage going to the next region faster. If the agent reaches a termination state by going out of bounds which is outside the green box, or taking more than 90 seconds in timestep, it will be penalised by 2,000 and that episode will end. Finally, if it manages to complete the track, it will be rewarded by 2,000.

7.2. State Representation

In the initial stages of the project, a single camera frame was used as a state representation. However, a single image might not be comprehensive enough to represent the agent's state as it is not able to show the direction and speed of the robot.

As seen in Figure 9, two other methods for state representation were explored. One was to stack frames while the other was to use the pixel value difference between the current and previous frame, which makes the environment similar to a Markovian environment [6].

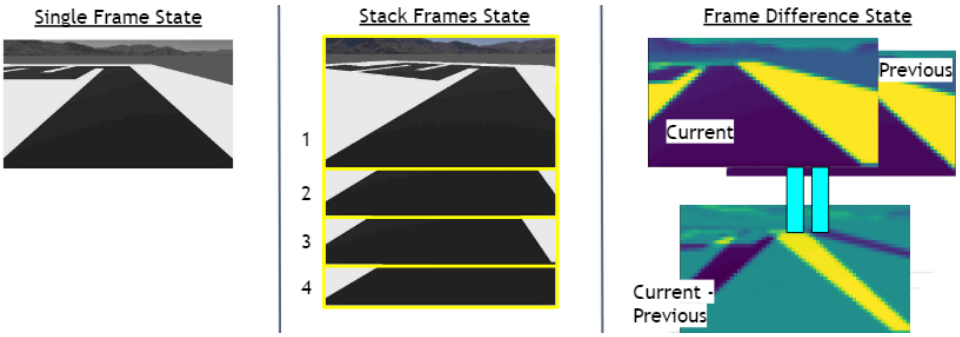


Figure 9: 3 pictorial depictions of how frame(s) could be used as a state representation

8. Training and Challenges Faced

This section describes the main challenges faced when training and the parameters changed to try overcome the challenges.

8.1. Challenge 1: Spinning

The first major problem faced was that the agent determined spinning on the spot was the best strategy. The spinning problem occurred when the ResNet 50 variant was first used in training. This behaviour was observed after a significant number of episodes had occurred and the agent chose to keep spinning on the spot especially at the starting point.

It was later identified that it could be caused by the average pooling at the end of the ResNet structure. Average pooling is a function that reduces spatial size to improve computation and to extract low level features from the surrounding [7], which is good for image classification tasks. However, in this case, average pooling might remove features that could be important for the fully connected layers to deduce the optimal action to take. Consequently, the model stopped spinning after the pooling function was removed.

8.2. Challenge 2: Irreconcilable Regions of Navigation

Another issue faced were regions where the agent was unable to properly move forward. For this issue, both basic DQN and ResNet algorithms has issues clearing the first straight path and turning left subsequently into a turn, seen in the figure below.

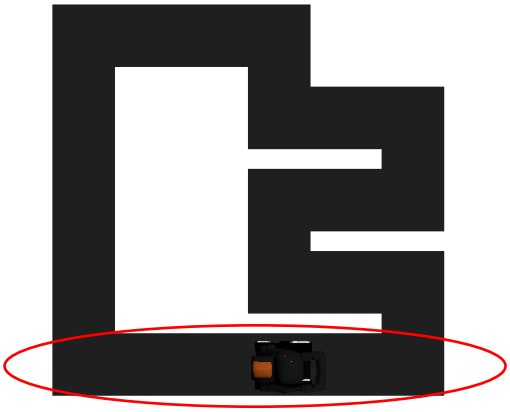


Figure 10: The circled region shows the area where the agent has trouble navigating properly

Instead of going straight, the agent fails to navigate the straight path circled in Figure 10. The following sub-sections describe the various tunings to the training code to overcome said problem, with each tuning stacking with the next unless specified.

8.2.1. Tuning 1 – New region, batch size, PReLU

To tackle this issue, an additional region at the start of the track was added to facilitate learning to go straight.

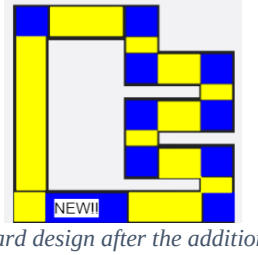
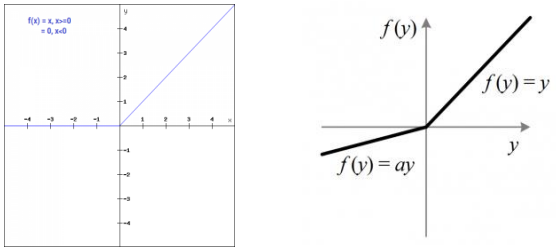


Figure 11: Reward design after the addition of 1 region

In addition, batch size was also changed from 32 to 256. The increase in batch size refers to the number of samples that is taken from the replay memory to train the NN. This should allow more important samples, such as experiences of moving to the other side of the track, to be used in training. Furthermore, all activation functions were changed from ReLU to parameterized rectified linear (PReLU). The reason for this change is due to the negative regions of the ReLU graph, as seen in the graph below.



ReLU PReLU

Figure 13: Graphical comparison between a ReLu (Left) function and PReLu (Right) function

As seen in Figure 12, any negative value in the ReLU graph will be multiplied with a zero and cause the node to be permanently deactivated [8]. This could result in the function not mapping the negative values correctly. Therefore, a PReLU function was used to solve that problem by giving the negative region a gradient that is mapped to the variable 'a', which is another trainable parameter. Doing so allows the activation function to be more robust since it now also consider values not possible using ReLU [8].

However, both basic and ResNet variant remains stuck in the first stretch of the course, with the reward graph having a negative sloping trendline after about 3.8k episodes as seen below.

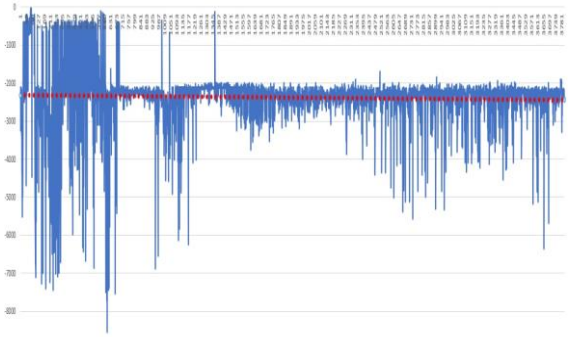


Figure 12: Basic DQN's Reward against 3,800 Episodes plot with linear trendline

8.2.2. Tuning 2 – Frame difference state representation, Hyperparameter

The next parameter to change is to use frame difference as state representation and updating the hyperparameters. The idea of using DQN to train a self-driving agent is not new and has been done many times in other setups like The Open Racing Car Simulator (TORCS), which simulates a more racing game-like environment. After scouring through some works, I decided to update my hyperparameters to as follows, but the problem still persists after 18 thousand episodes.

Table 1: Updated hyperparameters based on works done in TORCS

Hyperparameters in DQN	
Target net update	4
discount factor	0.98
learning rate	0.005
batch size	2048
decay rate	0.00003
ϵ_{min}	0.01

8.2.3. Tuning 3 – Reward Function

Next, two other reward function was explored. The first reward function replaces penalising staying in current region by 1 to 0 but fails to produce a working model.

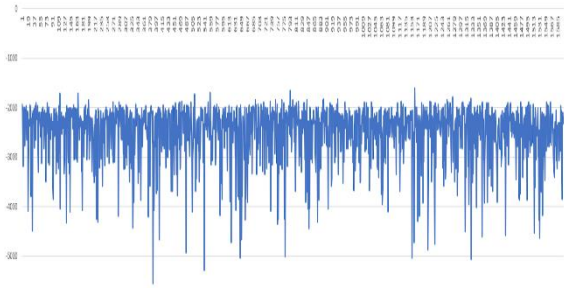


Figure 14: Basic DQN's Reward against 1,600 Episodes plot

The second reward function uses a staircase function:

$$Reward = -20 + N_{regions} \quad (2)$$

where $N_{regions}$ is the number of times the agent cross to the next region, but also fails to produce a working model.

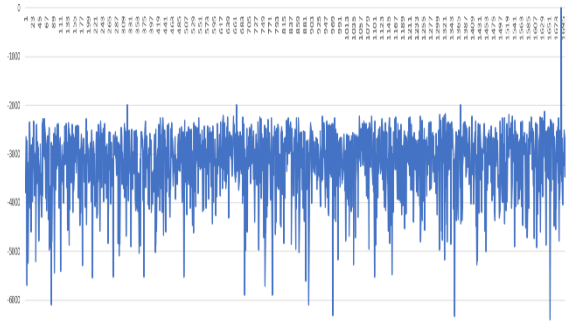


Figure 15: Basic DQN's Reward against 1,700 Episodes plot

Note that the 0 reward spike at the end of Figure 15 was due to simulation errors when the virtual Vector interact weirdly with the environment. Further training uses back the reward function in Figure 8.

8.2.4. Tuning 4 – Stack frames state representation in ResNet variant

While the basic DQN uses frame difference state representation, the ResNet variant explores using the stack frames state representation. A stack of four frames was experimented, with the idea of using four frames originating from the Atari paper that utilized a DQN to train an agent to play a wide variety of Atari games [9].

However, this implementation significantly increases the computational complexity and reduces training speed as each experience now carries eight times more image data than using one frame. To cater to this increase in computational requirements, the batch size was reduced back to 64 for a reasonable training speed.

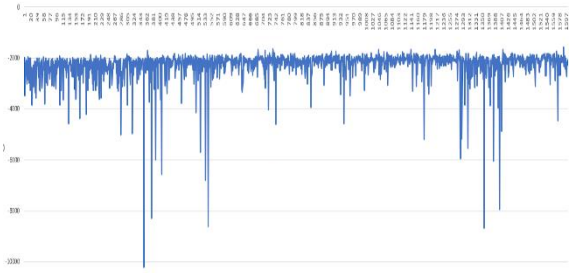


Figure 16: ResNet50-DQN's Reward against 1,600 Episodes plot

As seen in Figure 16, though the agent was unable to complete the track, the reward graph seemed to have a slight upward gradient, which nods to possible convergence, but at a very slow rate. Hence, it was decided to stop training the ResNet variant and to focus on the faster and lighter basic DQN.

8.2.5. Tuning 5 – Randomised start

Randomised start was an interesting implementation adopted from one of the works done using TORCS. The author noted that implementing such strategy can improve convergence rate while also helping the agent to not over-fit its policies to the start of the track [10]. In this project, randomised start is adopted such that the virtual Vector model will start from any point within the first region.

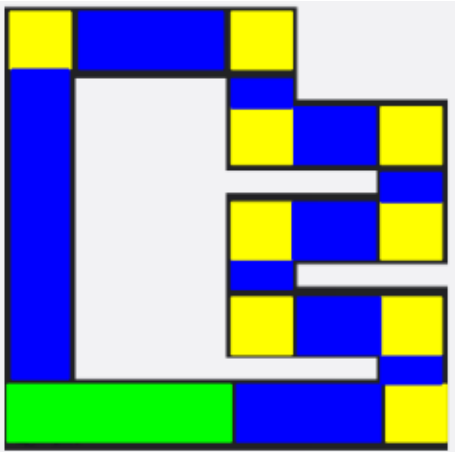


Figure 17: Randomised start within the green starting region

As seen in Figure 17, randomised start was used with the initial reward design of the track to give it a larger area of freedom. The virtual Vector model was allowed to start each episode at any point within the green region, but always facing to the right.

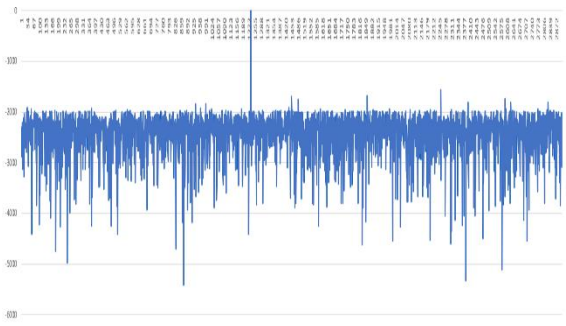


Figure 19: Basic DQN's Reward against 2,900 Episodes plot

Though it was an interesting strategy, it was unable to solve the problem and this strategy was abandoned.

8.2.6. Tuning 6 – New regions

Seeing how the agent was still having trouble clearing the straight path, two new regions were added to further facilitate learning to go straight.

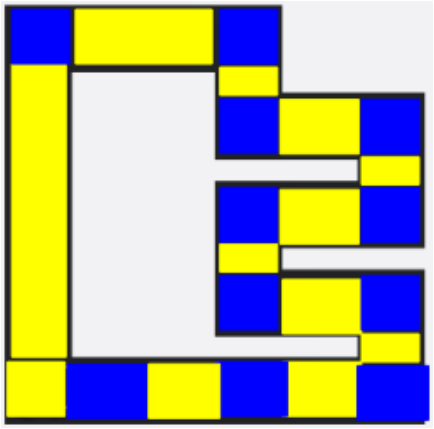


Figure 20: Reward design after adding two more regions to the design in Figure 11

As seen in Figure 19, two more regions were added to the reward design in Figure 11 but doing so doesn't help in the long run as the variance gets larger as shown below.

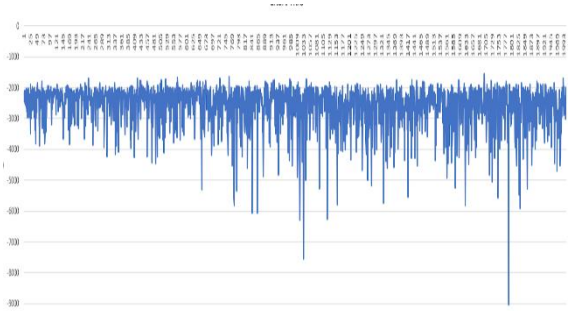


Figure 18: Basic DQN's Reward against 2,000 Episodes plot

8.2.7. Tuning 7 – Stack frames state representation in Basic DQN

Inspired by the possible success of using stack frames as state representation in the ResNet variant, four stacked frames were

also used. Although the increase in computation to process extra frames decreased the training speed, it was not as bad as the situation in the ResNet variant, and batch size was reduced to 512 instead of 64.

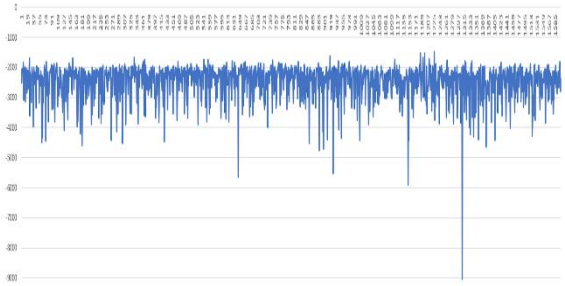


Figure 21: Basic DQN's Reward against 1,600 Episodes plot

While it wasn't able to solve the problem, using four frames as the state representation seems to have lower reward variance compared to Figure 20.

8.2.8. Tuning 8 – Pytorch DQN structure, Stricter reward, Adaptive epsilon

Seeing how changing parameters doesn't improve the situation, the basic DQN structure was revisited. It was found that Pytorch's own basic DQN structure had an addition of batch normalisation (batch-norm) layer between each convolution layer and its activation function.

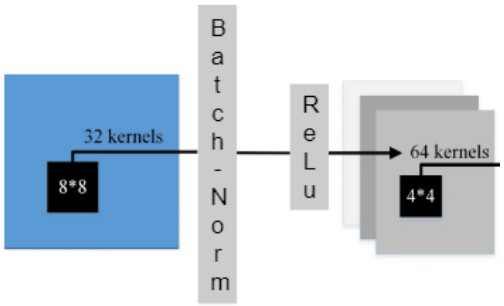


Figure 22: Example of Pytorch's implementation of a basic DQN structure with batch-norm after a convolution layer

Batch-norm helps with convergence rate while allowing for more room during initialization [11], which could be especially important in this project as the time frame doesn't allow for more testing of hyperparameters.

The reward function was also further refined by making it stricter for the agent.

Reward:
1) +100 , go next region
2) -100 , go previous region
3) -1 , stay in current region
4) -2,000 , going off track or take >90s
5) +2,000 , complete track

Figure 23: Stricter reward function with 1 less reward condition compared to previous version

As seen in Figure 23, the new reward function discourages the agent from going anywhere off track, in hopes of faster convergence rate with this reduce complexity.

Finally, an adaptive epsilon method was explored to further facilitate training the straight path.

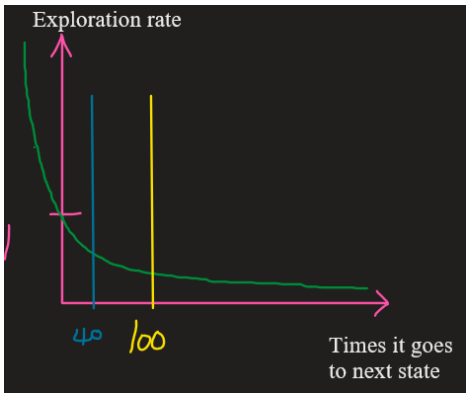


Figure 24: Graphical example of the different exploration rate in yellow (current) and blue (next) region against the number of times it goes to the next region or state

Imagine a blue and yellow region like in Figure 19, where the blue region is the next region from the yellow region. As seen in the example in Figure 24, if an agent is able to cross from the yellow region to the blue region 100 times while the blue region only manages 40 times, the yellow region would have a lower exploration rate than the blue region. This effectively trains the agent one region at a time in hopes of learning the proper route to take at each region.

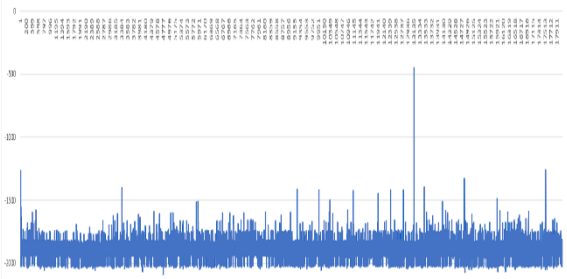


Figure 26: Pytorch DQN's Reward against 21,700 Episodes plot

However, even after 21 thousand training episode, the agent was still stuck at the first part of the track.

8.2.9. Tuning 9 – Boltzmann Exploration

On the topic of epsilon, another exploration method called the Boltzmann Exploration method was investigated.

As seen in Figure 26 [13], the main difference for the Boltzmann Approach is that the probability of choosing an action is based on its Q-value. The higher the Q-value, the higher the probability of that action to be chosen. This could be beneficial when the Epsilon-greedy strategy chose to explore, and the worst action has equal chance as the second-best action [13].

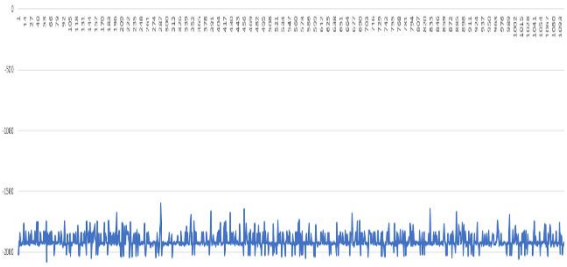


Figure 25: Pytorch DQN's Reward against 1,100 Episodes plot

As seen in Figure 27, it was unable to show an increase in reward after 1,100 training episodes. Thus, Epsilon-greedy strategy was used instead.

8.2.10. Tuning 10 – Rotation based movement, Reward function

While looking closer at how the virtual Vector robot move, it was observed that turning was not perfect and after a series of turns on the spot, the robot would be slightly out of place and unable to return to the starting direction before turning. This could potentially cause instability in training as the

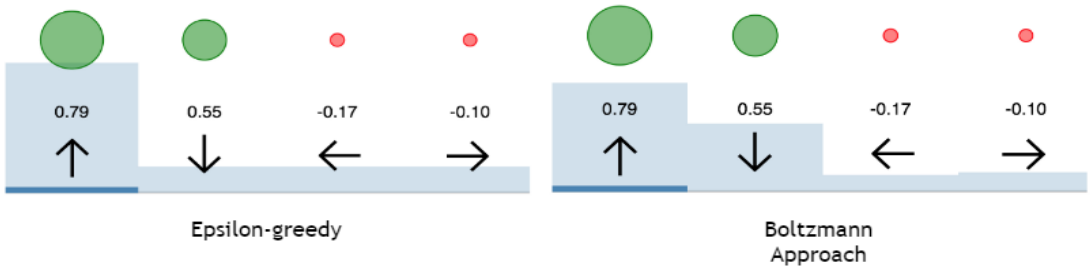


Figure 27: Pictorial depiction between Epsilon-greedy (Left) and Boltzmann Approach (Right). The height of the light blue bars corresponds to the probability of choosing that action. The dark blue bar at the bottom corresponds to the chosen action. The numerical values correspond to the Q-values of that action

actual turns would either overshoot or underestimate its prediction.

Initially, the virtual robot moves based on velocity control. Turning left would cause the left wheels to rotate backwards while the right wheels rotate forward at the same speed. To allow for more precise turns, movements were changed to rotation based where the virtual robot would just change its axis angle rotation when turning left or right. This allows for better control of the rotation resolution while also ensuring precise turns. For this implementation, the virtual robot is only allowed to make 15 degree turns in either direction.

Unfortunately, using this movement policy made using four frames for state representation invalid. When using velocity control, intermediate states while the robot was turning could be captured. Since the rotational policy moves the robot instantaneously, this caused it not to have any intermediate states. Consequently, only 2 frames of before and after moving could be stacked for state representation.

To further reduce complexity and facilitate travelling in the correct direction, a new reward function was used to encourage staying in the middle of the track and travelling in the direction of the next region.

$$Reward = \cos\theta - \left|\frac{P_y}{W_d}\right| \qquad (3)$$

Where θ is the angle between where the virtual robot is facing and the direction to the next region, P_y is the distance from the robot to the middle of the track and W_d is half the track length.

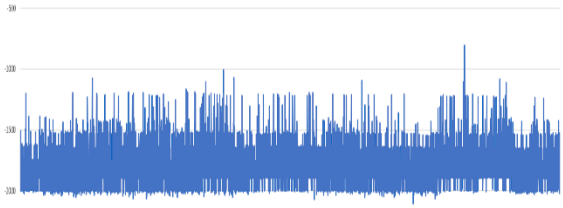


Figure 28: Pytorch DQN's Reward against 11,300 Episodes plot

However, even with so much tuning, the agent could not overcome the challenge

9. Model 5300 and Discussion

While looking back at earlier trainings, it was observed that reducing action space and using one frame as the state representation allows the agent to complete the track in some episodes.

In earlier trainings, the action space was not setup properly that caused the DQN to only have two outputs, forward and turn left. Surprising, the agent shows that it can navigate the track well even at right turns where it would just do a full left turn until the desired angle to carry on. One possible theory could be that since the action space has one less dimension, this allows for faster convergence.

After correcting the action space from two to three, the agent shows signs of being able to complete the track regularly after 5,300

training episodes, hence the name Model 5300.

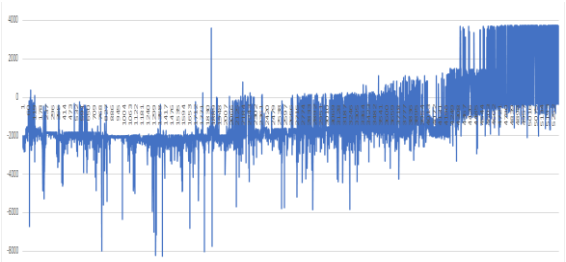


Figure 29: Basic DQN's Reward against 5,300 Episodes plot

As seen in Figure 29, the agent was able to regularly hit close to 4,000 reward value after about 4,500 training episodes. The hyperparameters used is as follow while the reward design follows Figure 8.

Table 2: Hyperparameters of Model 5300

Batch Size	32
Discount Factor	0.99
Epsilon Decay	0.001
Target Update	10
Replay Memory Size	100,000
Learning Rate	0.00025

To test if Model 5300 was able to generalise and not just run on the training tracks, three other tracks with similar themes, but never seen before by the trained agent, were used to test the model.

Figure 31: 3 new tracks to test if the agent will be able to navigate turns (Top), navigate a slanted path (Bottom-Left) and navigate an oval shape similar to a basic racetrack (Bottom-Right)

In addition to Model 5300 being able to clear the three new testing tracks in Figure 30, using the same training code, this behaviour was also reproducible after training for 9 thousand training episodes.

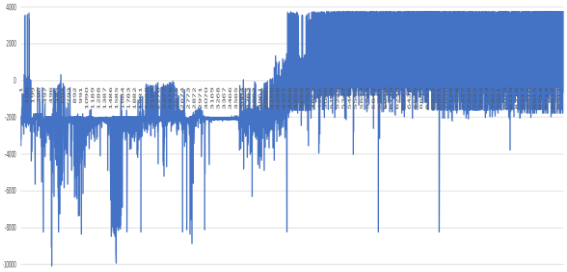


Figure 30: Basic DQN's Reward against 9,000 Episodes plot

The results are definitely surprising as it was initially thought that using a single frame as state representation would not be enough to convey variables like velocity and the direction it is going. However, this project provides an insight that it might not be the case and in this specific experiment, using more frames or altered frames could cause more harm than good. With that being said, more testing would definitely be needed to properly analyse such contradicting behavior.

10. Hardware

This section covers the hardware and Sim-to-Real implementation of this project

10.1. Robot

Figure 32: Product picture of the Vector robot by Anki

The hardware used in this project is a product from Anki, a US robotics company, called Vector. Vector is designed to be an autonomous robot, where the user will be interacting with the robot as though with a pet, or at least that is the general intention behind the product design.

Within the robot itself, it houses a Qualcomm 200 Platform, a HD camera with a 120 FOV, beamforming four-microphone Array, infrared Laser Scanner, 6-Axis IMU, a color IPS display, and WiFi capabilities.

10.2. Sim-to-Real Transfer

After successfully obtaining Model 5300 to enable the robotic agent to drive itself, transferring to the actual robot itself will be implemented as seen in the flowchart below.

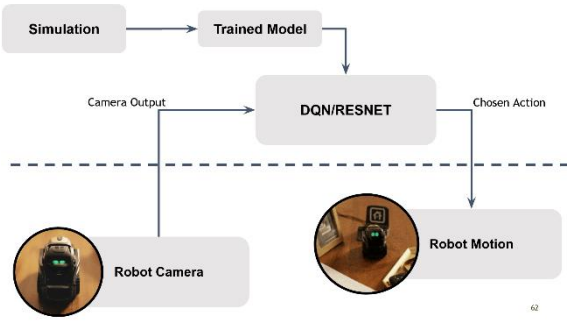


Figure 34: Flowchart of the implementation of Model 5300 with Vector's SDK

As seen in Figure 33, Model 5300 is loaded within the robot's code. The DQN model will receive images from the robot's actual camera and process those images using the same process as during simulation. The output of this DQN algorithm will be a chosen action as mentioned in the previous sections, but this time it will be sent to the robot's SDK where Vector will execute that action.

However, trying to plug-and-play Model 5300 did not work. It was later identified that the cause of this problem was the mountains in the background in the Webots simulation environment, where the NN might have used those mountains as one of the features of interest to determine which action to take. An example of the Webots environment with the mountains is shown below.

Figure 33: A frame captured by the virtual robot while in training. The background has a mountainous scene that might have used been extracted during training

This is a serious problem as the actual robot also expects to see mountains in the background or it will be unable to determine how to move with its current model. This issue, called the Sim-to-Real transfer problem, is precisely the reason why researchers have meticulously replicated their simulation environment to their actual environment where they are going to test the robot, such as the Duckietown environment [14].

To prevent the need of having a complicated background in actual testing, the

background of the simulated environment was changed to a white backdrop as seen below, and the entire model was retrained from scratch with the white backdrop.

Figure 35: A frame captured by the virtual robot while training with a white backdrop

After 14,700 training episodes, a few models were tested to have comparable or better performance than Model 5300.

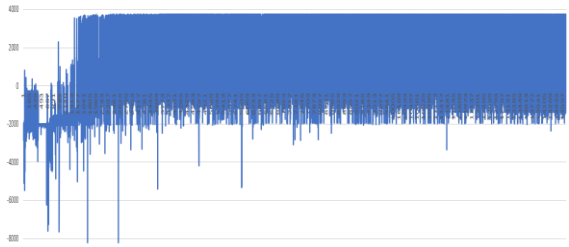


Figure 36: Basic DQN's Reward against 14,700 Episodes plot

As seen in Figure 36, using a white backdrop improve convergence rate as the agent was able to hit close to 4,000 reward value after about 1,477 training episodes, compared to the 4,500 training episodes when using a mountainous background. Out of the 147 models saved, eight models managed to navigate the training track when tested. The final chosen model was selected based on the fastest clear time and stable behaviour when navigating the track.

11. Results and Discussion

After obtaining the new Model 12400, the virtual environment where Vector was trained in was also replicated in real-world to be as close as the simulated environment. This was done by printing out the entire simulated track in life size and using four white corrugated boards as walls to replicate the white backdrop in the Webots environment. These adaptations proved to be enough for Vector to successfully completed the course as seen in the images below.

Aside from the successful completion of the training track, it was also observed that the NN contains some tolerance not intentionally designed into the architecture. These tolerances came in the form of slight differences in the camera input, as the camera input within the simulated environment is well-lit and has good contrast, while the real camera input is not as well lit and crisp and the inputs are affected by the built-in fisheye lens on the physical camera. In addition to these deviations from the simulation, the actual

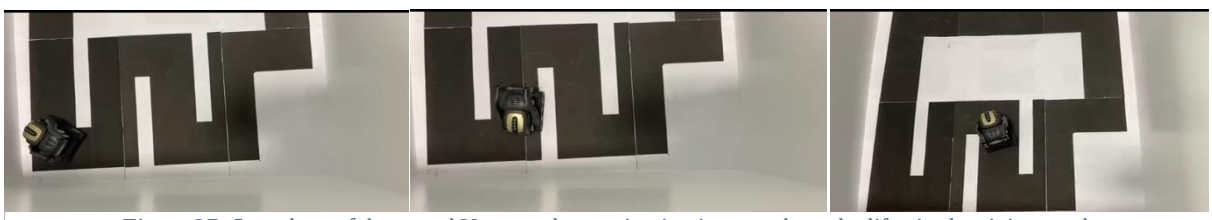


Figure 37: Snapshots of the actual Vector robot navigating its way about the life-sized training track

tracks used in the actual implementation is not printed to a very high standard, and in Figure 37, lines from the paper boundaries could be seen. However, these physical deviations from the simulation might have been omitted by the image pre-processing algorithms such as the resolution reduction and grayscaling which therefore helped to approximate the input from the actual camera to something that is close enough to the simulated images that the DQN model was trained on.

12. Conclusion

In this project, we explored various tuning methods of DQN and the results it bring, and also attempted to apply the resulting DQN model in real-life. This project first explored the implementation of a basic DQN structure and a ResNet variant that replaces the three convolution layers. However, we ultimately went only with the basic DQN structure as it was computationally the least intensive and allowed us to run the number of simulations we had in such a short amount of time, coupled with the staggering limitations of our laptop hardware. Subsequently, we faced the challenge of Sim-to-Real transfer and had to retrain our model using a white backdrop so that it could be easily replicated in real-life using white corrugated boards. With these experiences, we gained a deeper understanding of the inner-working of DQN and RL, and the challenges faced with autonomous driving, in particular where Sim-to-Real transfer is concerned. To allow autonomous driving to occur in real life, large simulations have to be computed where every single possible scenario is considered and trained upon. As shown with our limited computational power and success, it seems that perhaps there is still a possibility where autonomous driving can occur in our lifetimes.

13. References

[1] "Webots: robot simulator", Cyberbotics.com. [Online]. Available: https://cyberbotics.com/.

[2] "PyTorch", Pytorch.org. [Online]. Available: https://pytorch.org/.

[3] R. Liu and J. Zou, "The Effects of Memory Replay in Reinforcement Learning", 2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton), 2018. Available: 10.1109/allerton.2018.8636075.

[4] M. Sewak, "Deep Q Network (DQN), Double DQN, and Dueling DQN", Deep Reinforcement Learning, pp. 95-108, 2019. Available: 10.1007/978-981-13-8285-7_8.

[5] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition", 2016 IEEE Conference on Computer Vision and Pattern Recognition

(CVPR), 2016. Available: 10.1109/cvpr.2016.90.

[6] T. Tan and E. Brunskill, "CS234 Notes - Lecture 6 CNNs and Deep Q Learning", Stanford University, 2018.

[7] H. Gholamalinezhad1 and H. Khosravi, "Pooling Methods in Deep Neural Networks, a Review."

[8] D. Pedamonti, "Comparison of non-linear activation functions for deep neural networks on MNIST classification task", arXiv, 2018.

[9] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning", arXiv, 2013.

[10] M. Vitelli and A. Nayebi, "CARMA: A Deep Reinforcement Learning Approach to Autonomous Driving", Stanford University.

[11] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift", arXiv, 2015.

[12] A. Juliani, "Simple Reinforcement Learning with Tensorflow Part 7: Action-Selection Strategies for Exploration", Medium, 2016. [Online]. Available: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf.

[13] J. Groot Kormelink, M. M. Drugan and M. Wiering, "Exploration Methods for Connectionist Q-learning in Bomberman", Proceedings of the 10th International Conference on Agents and Artificial Intelligence, 2018. Available: 10.5220/0006556403550362.

[14] A. Kalapos, C. Gor, R. Moni and I. Harmati, "Sim-to-real reinforcement learning applied to end-to-end vehicle control", 2020 23rd International Symposium on Measurement and Control in Robotics (ISMCR), 2020. Available: 10.1109/ismcr51255.2020.9263751.