Using Deep Reinforenment Learning methods for solving OpenAI's Highway Environments

Matan Velner , Ophir Benjamin

Efi Arazi school of Computer Science, Reichman University

Matan.velner@post.idc.ac.il, Ophir.benjamin@post.idc.ac.il

Abstract: OpenAI's Highway Environments provide a set of 2D-Autonomous Driving simulations designed for testing and researching ML algorithms. Previous works have shown great success in solving these environments using Kinematic Observations. In this paper, we attempt to solve said challenges using screen pixel information and environment rewards alone. We utilize and compare various RL algorithms and tools to determine the best approach for this challenge.

1. Introduction

Reinforcement Learning (RL) has shown tremendous progress in recent years following DeepMind's seminal papers [Mnih+13][Mnih+15] that displayed the potential of integrating Deep Neural Networks (DNN) into RL algorithms. Since then, many approaches and improvements have built upon this idea and are often compared using video games benchmarks, not unlike the comparisons made in this paper. The OpenAI Gym environment we attempt to solve is modeling one such video game. Here, a driver has to complete some section of a multi-lane road as fast as possible, without crashing into other vehicles and preferably driving on the rightmost lane. This game presents a classic RL scenario: At each time frame the environment presents the agent with an observation as a sequence of 4 matrices holding gray-scale pixel values. These values represent the game screen at the current time frame as well as 3 frames before it. The agent then chooses any of 5 available actions (Lane Left, Idle, Lane Right, Speed up, Slow down) to perform. The environment, in turn, presents the agent with the next observation, as well as a reward for the action it has just performed. An episode ends after 500 such steps or upon arriving at a terminal state (agent crashed into another vehicle). The agent's goal is to arrive at a policy such that the expected sum of rewards over an episode is maximal.
This paper compares different RL algorithms and tools over 3 challenges comprised of different variations in the Highway environment:

1. Exercise 1: **Highway Environment (Easy)**
   Here, the agent only navigates a 3-lane road, completely devoid of curves and turns. In this challenge the other drivers (bots) are relatively passive and sparsely placed on the road. Here we also encounter a stochastic transition model: for every time step there is a 15% chance that the action chosen by the action will be replaced randomly by a different one.
2. Exercise 2: **Highway Environment (Medium)**
   Same scenario as before, only on a harder setting- here the agent only navigates a 6-lane road, straight as before. In this challenge there are more drivers (bots). They are also more aggressive and more densely packed on the road.
3. Exercise 3: **Super Agent**
   Here, the agent should be able to navigates any of these 3 scenarios, and their different settings:
   - Highway Environment: As described before.
   - Merge Environment: The agent's road has a lane merging into it. The agent should allow seamless merger into its lane.
   - Roundabout Environment: The agent's road enters a roundabout. The agent must enter and leave the roundabout seamlessly.

For each of these challenges we have two goals:

- Solve the challenge with the least amount of steps taken by the agent in the environment. (A precise definition for 'solve' to be discussed later on).
- Solve (or attempt to solve) the environment with different approaches and compare their performances.

1.1. Related work

The OpenAI Gym is a popular testing ground and includes a large variety of games. As such, many of its environments have been tried by many. Notable mentions are:
Stable-Baseline3[1]: Open source collection of implementations of RL algorithms in PyTorch. This collection supports the Highway Environment we review here.
Phil Tabor[2]: An open source implementation of many DQN variations. These are usually accompanied with a video walk-through and analysis of the details, pros and cons of each algorithm, which we have used as a significant aid in this work- both for implementation details and high level insight on the ways of the algorithms.
Rainbow Is All You Need[3]: An open source step by step implementation of DeepMind's Rainbow model [Hes+18], designed to act on any OpenAI Gym environment.

2. Solution

Before choosing the algorithms and tools to solve this environment, we must dissect the components of the task and formalize them as a reinforcement learning scenario. The environment is a game system where a player can control a vehicle driving in some segment of a road alongside other (bot controlled) vehicles.
The learning agent will play this game and attempt to maximize its score (sum of rewards). In this scenario, the agent's observations of the environment are comprised solely of the game screen at the current time frame as well as at three previous time frames (presented in grey-scale mode). That is, for each time frame t, s_t is a set of 4 $128x128$ matrices with integer values in the range $[0,255]$.
For each time frame, the agent chooses an action a_t from a discrete action space $\{0,1,2,3,4\}$ representing (Lane Left, Idle, Lane Right, Speed up, Slow down) respectively. After performing a_t the environment responds with a reward $r_t \in [0,1]$, and the next observations s_{t+1}.
It is worth noting that taking an unavailable action (e.g turning right from the rightmost lane) is equivalent to taking the Idle action as far as affecting the environment. The agent is rewarded for driving fast and avoiding collisions, with some benefit to driving on the rightmost lane. Our goal is to train an agent, based on pixel information and rewards alone, that arrives at a policy that maximizes the expected rewards on an episode.

2.1. General Approach

The problem we attempt to solve is very similar in nature to the challenges solved in DeepMind's Atari saga. This, together with the vast amount of knowledge and success that have accumulated in recent years in using temporal difference value-based methods, have caused us to focus on the Deep Q Network (DQN) approach. Specifically, we intend on starting with a simple version of DDQN, building upon it and improving it using modern variations and enhancements such as Dueling, Prioritized Experience Replay and more.
The decision to skip the most basic DQN was made considering its subpar performance and the high instability it has shown in previous works (see model comparison on [Hes+17] page 6).
An additional approach we would like to explore is using Actor-Critic, policy gradient algorithms such as A3C [Mnih+16] and its synchronous counterpart A2C. From this family of methods we wish to explore Proximal Policy Optimization algorithms (PPO) as they showcase improved results over many benchmarks [Sch+17] compared to A2C and A3C.
Throughout our experiments we ensured that the only information the agent can infer from is given by the environment (raw pixel data and rewards), and not to inject our own understanding of the task into the process (e.g forbidding unavailable actions, cropping irrelevant parts of the image).

2.2. Design

Our implementation started by constructing a general framework to accommodate different approaches. This framework is composed of 5 categories:

1. **Agents** - These code cells include a parent *Agent* Class from which all other DQN Agent classes inherit. This parent class holds common DQN actions such as interacting with the replay buffer and decrementing the epsilon value . The other agent classes each represent a single algorithm variation (e.g DDQN, D3QN) that can run on any environment and implement algorithm-specific actions such as taking a learning step or sampling an action.

2. **Components** - These code cells hold the building blocks required to compose a learning agent. Here we implement classes for the Convolution Neural Networks, ICM Module [Pat+17], Replay Buffer, etc. These are attached to agents using composition and are designed to be generally used.

3. **Train & Evaluation Scheme** - This code cell holds the training and evaluation schemes. The training scheme receives an agent, a list of environments and their settings (e.g stochastic transitions set on/off) and a number of episodes to run. It trains the agent over all the given environments, choosing one randomly each episode. This scheme also collects and returns information on the progress (e.g rewards and step counts) made by the agent.

4. **Attempts** - These code cells represent an agent's attempt to learn a given one or more environments. Here we set the hyper-parameters of the agent, describe on what environment we wish to train it and for how long.

5. **Utilities** - These code cells provide some common services such as displaying training history information, generating a video of an agent or implementing some data structure used elsewhere in the project.

For our initial DDQN attempt we turned to the network architecture seen in [Mnih+15;Has+15]: 3 convolution layers and a fully-connected hidden layer (approximately 1.5M parameters in total), using Mean Squared Error loss (MSE) and Adam optimizer. Later on, we implemented a dueling heads architecture (D3QN), Prioritized Experience Replay buffer (PER [Sch+16]), an Intrinsic Curiousity Model (ICM [Pat+17]) and more.

3. Results

3.1. Experimental Results

We decided to treat the highway environment as our testing ground and move forward once we have a better understanding of the task at hand and the challenges involved.

Exercise 1 and 2 are based upon the exact same highway environment under different configurations:
Exercise 1 has only 3 lanes and other vehicles are sparse and relatively passive. The major contributing factor to its difficulty is the stochastic transition model.
Exercise 2 has twice the lanes (6) and other vehicles are more tightly packed, as well as more aggressive. The deterministic transition model is a great aid in helping the agent converge to a decent policy.
It is for this reason that we decided to combine these exercises in this section of the paper. Our goal is to finish this section with an agent that performs well on both exercise 1 and 2, and pass it over as a starting point to exercise 3.

3.1.1. Double Deep Q Network (DDQN)

Our first experiment was a standard DDQN agent without any improvements attempting the 1st environment (Highway, easy mode). The model was run for 200 episodes, with $T = 200$ (terminating an episode after 200 steps, if it the agent hasn't crashed before that), $\mu = 1e-4$, $\gamma = 0.99$ and decrementing ε from 1 to 0.1 over 120 episodes. The run lasted for about 5,700 steps. We display the results averaged over the last 100 episodes performed (no data for the first 100 episodes).

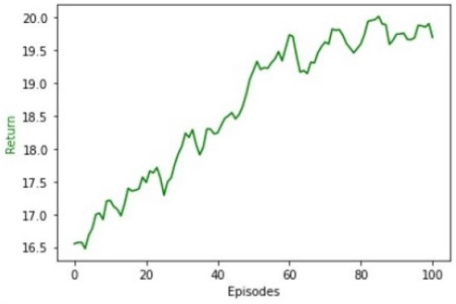


Fig. 1. Average reward per episode (RPE)

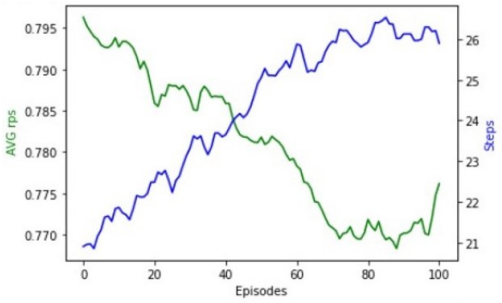


Fig. 2. Average reward per step (RPS), Step count

We see the positive trend clearly in Fig. 1. However, this only tells half the story. Fig. 2 shows the step count (averaged over 100 episodes) as well as the average reward per step (RPS). This is crucial to determine the willingness of the agent to take risks in order win over the maximal reward at each step. A careful agent will attempt to increase its rewards be staying in the game as long as possible, avoiding any risks (speeding, overtaking other vehicles, etc.). This is effective, but results in sub-optimal rewards. In Fig. 2 we can see a problematic behavior that will continue to appear in many of our experiments: the agent learns to slow down (indicated by the decreasing average RPS), in exchange for playing the game longer (indicated by increasing average step count)

and ultimately increasing its key performance indicator: average RPE.
It is already clear that besides having an agent learn to *survive* in the environment, our job must include ways of exciting the agent to explore riskier, yet more rewarding policies.
Additionally, this initial trial gives us a rough baseline of the progress we can expect in about 6000 steps- not much.
After this trial we also experimented with different network architectures: larger models, smaller kernels, different activation functions and batch norms. Our main conclusions are as follows:

- Long Networks are a bad fit- While bigger networks have more expressive capabilities and can potentially learn more complex behaviors, our goal to solve the environments quickly drives us to consider the smallest networks that are able to learn our task.
- Batch Norms don't seem to work- Even though batch norms are designed to speed up the learning process, in our experiments they did not display any improvements.

We decided to opt for a network design very similar to the one in the original paper:

Parameters	Value
Q network: channels: channels	32, 64, 64
Q network: filter size	8 x 8, 4 x 4, 3 x 3
Q network: stride	4, 2, 1
Q network: hidden units	512
Activation function	Relu
Q network: output units	5 (number of actions)

At this point we decided to move on and add a dueling head mechanism to the agent.

3.1.2. DuelingDDQN (D3QN)

The dueling architecture is introduced in [Wan+16] as an "alternative but complementary approach... better suited for model-free RL", and is shown to outperform basic DDQN models in many of the Atari games benchmarks.
Implementing some lessons from our first trial, we set up a larger experiment for the D3QN agent: The model was run for 500 episodes, with $T = 200$, $\mu = 5e-4$, $\gamma = 0.99$ and decrementing ε from 1 to 0.1 over 200 episodes. The run lasted for about 15,600 steps or about 13 minutes (GPU).

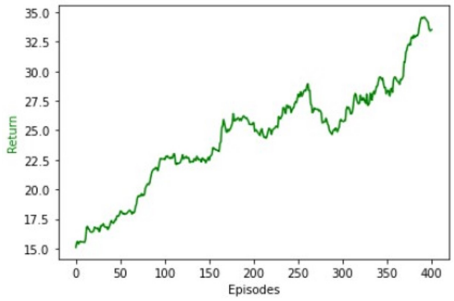


Fig. 3. Average reward per episode (RPE)

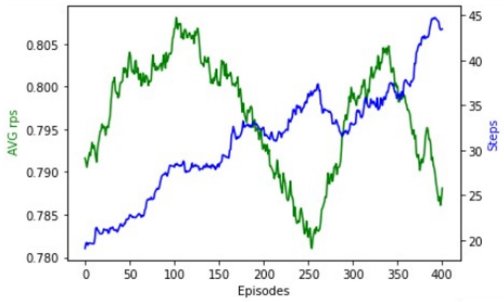


Fig. 4. Average reward per step (RPS), Step count

Despite the slight increase and general upward trend in key performance metrics, we still observe the oscillating pattern negatively correlating the average number of steps taken before collision and the average RPS, indicating that the agent still tends to benefit more from low risk strategies, rather than riskier high RPS ones.
At this point we have started experimenting with 3 additional concepts to enhance the agent's performance and lower its training time:

- **Action Repeat** - In [Mnih+13] the agent is only presented with a choice once every 4 frames. For some repeat value k this means that after choosing an action, the agent will inevitably attempt to take this action for k consecutive time frames (in exercise 1 the stochastic transition model still might affect the actual action taken). We experimented with saving and discarding the transitions that occur between the agents' choices, arriving at a design that emulates lowering the temporal resolution for the agent: given a state s_t

, the agent chooses an action a_t and attempts to act on it for 4 frames. This generates a sequence of states $(s_t, s_{t+1}, ..., s_{t+k-1})$, and rewards $(r_t, r_{t+1}, ..., r_{t+k-1})$. we save the past k frames into the replay buffer as one transition: $(s_t, a_t, \sum_{i=0}^{k-1} r_{t+i}, s_{t+k-1}, d)$, where d is true if any of the states $\{s_t, ...s_{t+k-1}\}$ is terminal.
The idea is to run k times more episodes without significantly increasing the overall run time of the training phase. Since the vast majority of the computation is involved in feeding through the neural networks, reducing the rate in which we access them decreases the run time. Additional upside of this setup is that the agent experiences the environment in larger time frames, with the intuition that we gain more control over the magnitude of the consequences of its action.
It is worth noting that our trials show that increasing k too much, could cause the agent to act overly cautious and avoid speeding up or overtaking other vehicles. So far a value of $k = 2$ has served us best.

- **Buffer Start-Up** - Another useful technique is shown in [Hes+17] and suggests playing several "dummy episodes" (paraphrasing) to fill the replay buffer before learning starts: "DQN and its variants do not perform learning updates during the first 200K frames, to ensure sufficiently uncorrelated updates". Therefore we decided to fill the replay buffer with 500 transitions before staring the learning phase (we don't count these steps when comparing models as they do not require any updates to the weights).

- **Quick Episodes** - In order to speed up the learning process we experimented with increasing exploration using smaller T values (max steps per episode). Similarly to action repeat, the idea is letting the agent experience a larger variety of states within a similar step count. We hope this will help the agent understand its influence on the environment better.

In order to single out the effects these improvements have on the learning process, we have set up a long training session for 3 DDQN agents over Exercise 2 (to allow easier training without a stochastic transition model) with identical hyper-parameter apart for the tested property. We first tested the effects of different action-repeat on the learning process over 3000 episodes. The hyper-parameters used here are: $T = 40, \mu = 2.5e-4, \gamma = 0.99$.

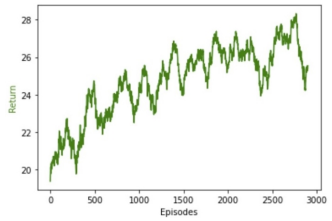


Fig. 5. *Repeat* $= 1$

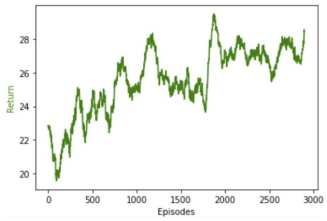


Fig. 6. *Repeat* $= 2$

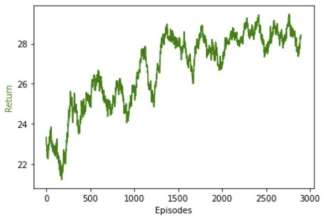
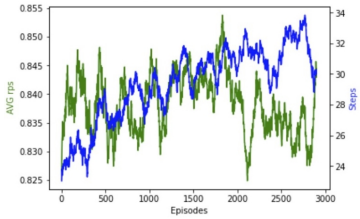
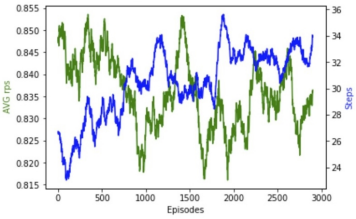
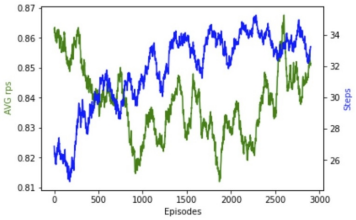


Fig. 7. *Repeat* $= 4$







While the general performance over the different action repeat values is similar, the higher value (4) displays higher episode returns and more a consistent high episode step count. This might seem insignificant but recall that for a repeat value 4 the agent only chooses an action and performs a learning step once every 4 moves, this means reduction of the the training time by up to a factor of 4. Therefore, in this trial we observe an agent slightly outperform its counterparts (where $repeat \in \{1,2\}$, while taking 25%-50% less steps- a significant improvement. Further tests have shown for that while in exercise 2 and 3 $repeat = 4$ works best, for exercise 1 $repeat = 2$ works better, perhaps because the agent has a harder time correlating its choices with the actions it ends up taking.
Testing the importance of short episodes and buffer start-up are not shown in this paper. Testing the effects of these parameters showed slight preference towards shorter episodes ($T = 40$), and a start-up value of around 200 transitions. We expect the effects of different values of T to be more visible when considering longer training sequences than we can run in the scope of this paper ($Overall - Steps > 1,000,000$).
At this point we decided to continue and implement a prioritized replay buffer for the agent to use.

3.1.3. Prioritized Experience Replay (PER)

The PER is first described in [Sch+16] as a method to :“replay important transitions more frequently, and therefore learn more efficiently”. Later on, in [Hes+17], PER is mentioned as one of the most significant improvements to a DDQN agent (together with multi-step on which we will elaborate later on), demonstrated by great reduction in performance when being removed from a complete Rainbow model. The PER is using a data structure called ”Sum Tree” for quick sampling of transitions from the replay buffer according to their respective priority. We used the Sum Tree implementation, as well as some guidance for the PER seen here [4].
The priority of a given transition is the size of the Temporal Difference Error (TD-Error), and the gradient descent step taken when learning from said transition is scaled with proportion to probability of sampling it in order to compensate for the non-uniform probabilities of the transitions in the buffer. Comparing the PER D3QN variant against the basic D3QN agent did not clearly show the PER variant learns faster. However, the idea that the differences between the two would be more clear over longer training sessions is consistent with the findings in [Hes+17] (where the gap begins to emerge at around 50 million steps, though playing a more complex game). This supports the idea that the PER agent is better at replaying crucial moments (such as collisions or near collisions) and learning how to handle them better.

3.1.4. Internal Curiosity Model (ICM)

ICM was initially introduced in [Pat+17] as a way to handle learning in environments where rewards are relatively sparse or absent altogether. However, in the original paper ICM is also shown to have some benefit in environments where rewards are relatively dense (such as our environment) so we were curious to test it. The main idea behind ICM is providing the agent with a second source of rewards that is managed intrinsically and is proportional to the difference between (an embedding of) the agents prediction of the next observation and (an embedding of) the actual next observation. Intuitively, the agent is rewarded for arriving at states it did not predict well, causing it to seek out unfamiliar states.
We experimented with various setups and hyper parameters and arrived at a network with the following design: 5 convolution layers fed in 2 fully connected heads each with 2 hidden layers. Details:

Parameters	Value
ICM network: channels	32, 32, 32, 32, 16
ICM network: filter size	3x3
ICM network: stride	2
ICM network: Inverse	256, 5
ICM network forward	256, 256
ICM network: state embedding space	256

conclusion best displayed by the results of this relatively long training session (3000 episodes, 80k steps). Hyper-parameters: $T = 40, \mu = 2.5e-4, \gamma = 0.99$.

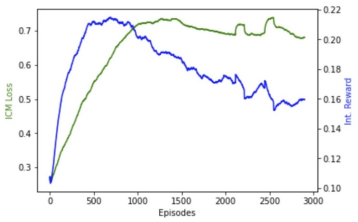


Fig. 8. ICM loss, Int. Reward

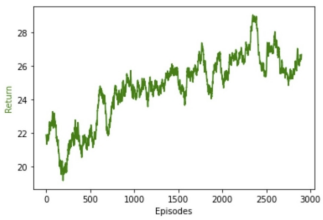


Fig. 9. RPE, Step Count

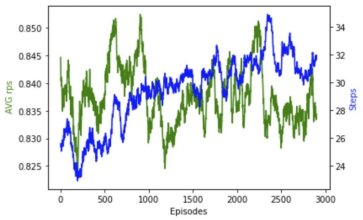


Fig. 10. RPS

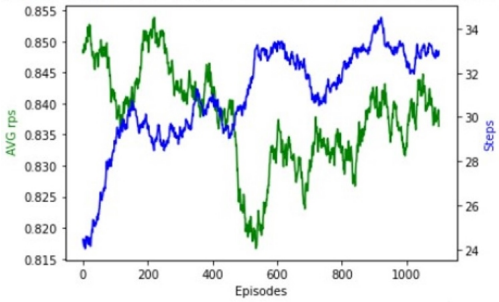
The key metrics in Fig 9, 10 read similar to the achievements we have seen in our repeat value tests before but took significantly longer to produce: the ICM module includes its own learning process over a CNN with 2 fully connected heads. This is very expensive computationally compared to our previous attempts. However, Fig 8 completes the picture: the steady decline in intrinsic reward indicates that the agent is slowly but surely running out of unfamiliar states to visit. We expect that as this process continues we will observe increase in the agents returns. However, since our task is raising an agent to solve the environment as quickly as possible (time-wise and step-wise), we have decided this method was not a good fit for our particular challenge.

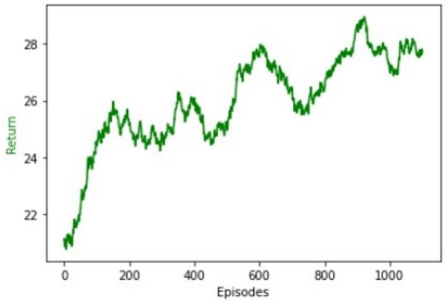
3.1.5. Multi-Step

The idea of Multi-step in reinforcement learning is described in [Sut88][Sut+98], and its purpose is described in [Hes+17] as being used to "shift the bias-variance trade-off and help to propagate newly observed rewards faster to earlier visited states". This is done by considering for each state, not only the reward that has been granted for it, but also the rewards granted in next n steps, as such:

$$R(n)_t = n \sum_{k=0}^{n-1} \gamma^k{}_t * R_{t+k+1}$$

As stated before, multi-step is described in [Hes+17] as one of the two most important improvements to a DDQN model. Since the other (PER) has been implemented we set out to add Multi-step to the agent. The results in the paper present $n = 3$ as the best choice.
To asses the improvement made by using Multi-step, we present the following training process, made over only 1200 episodes with hyper-parameters: $n = 3, Repeat = 4, T = 40, \mu = 2.5e-4, \gamma = 0.99$.





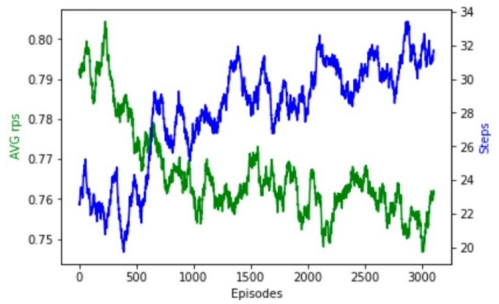
3.1.6. Choosing a champion

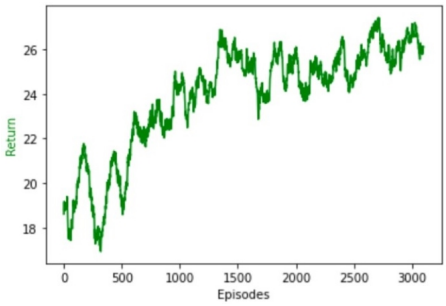
In [Hes+17] it is shown that many of the improvements that can be made to a DDQN model are in fact complimentary in that their advantages can be stacked. Therefore, and considering our experiments so far, we decide to use a D3QN model with a prioritized replay buffer, multi step mechanism and an action repeat of 4 (2 on exercise 1) as our preferred model for the task.

3.2. Final Results

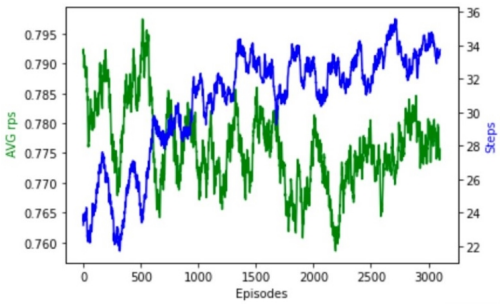
For each of the given exercises, we have trained our agent for 3200 steps using the same set of parameters (except exercise 1 where $repeat = 1$) . These are: $n = 3, repeat = 4, T = 40, \mu = 2.5e-4, \gamma = 0.99$.
Recall that n is the number of multi-step performed, $repeat$ is the number of actions repeat, T is the maximum number of steps taken in a learning episode, μ is the learning rate and γ is the discount factor. We have also load the buffer using 200 transitions before starting the learning process.
For each exercise we display the training process (as seen through the average reward per episode (RPE), average reward per step (RPS), and average step count). For evaluation information see colab notebook.

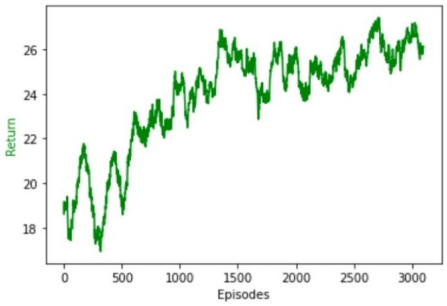
3.2.1. Exercise 1



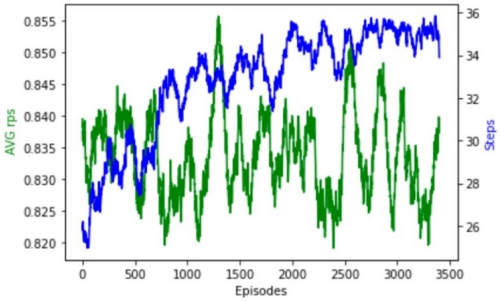


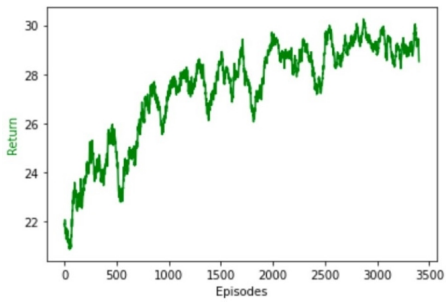
3.2.2. Exercise 2





3.2.3. Exercise 3

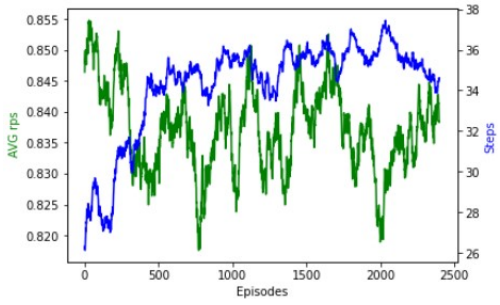


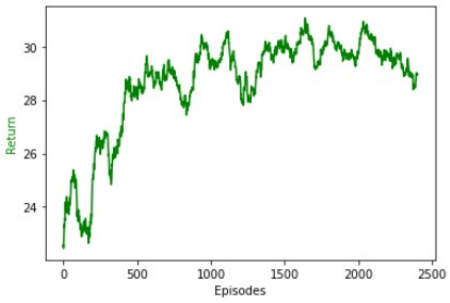


4. Discussion

While this paper displays a few high-point and crossroads we have passed through the process of solving this problem and tackling our first deep reinforcement learning task. However, the actual path contained many more than we could elaborate on. This is an opportunity for us to quickly go over some of those points before moving to a little more high-level view of our process:

- **A better model**- Since we write these words towards the submission of the paper, we can't help but feel a little frustrated that a small but significant improvement has presented itself right when we don't have the proper time to thoroughly compare it. Reading into jackyoung96's implementation[5] of the Rainbow model, we noticed the addition of heavy max pooling in the main Q network. Testing this over 2500 episodes showed great results, even when compared to our agent's 3000 episode learning sessions:





  This training was done over exercise 2, perhaps the easiest one (no stochastic model, only one scenario). However, at this point it was to late to properly compare to other models and in other scenarios. But definitely an interesting path forward.

- **Stopped to soon**- Many of the trials and experiments we have run throughout this work have left us with a sense of "what would happen x episodes down the line?". This is because the return graphs often would end with a clear upward trend (this is true, for example, to our final results). Many of the classic papers display training sessions of tens, sometimes hundreds of millions of training steps. However, since many of the games they attempted to solve were often more complex, it was hard for us to arrive at a reasonable estimation of how long should a successful training last. We were then bound with what tests were reasonable

to run given our setup using Google Colab and given our goal to solve using as few steps as possible. This means about 2 hours or more for a 3000 episodes training session, which did not seem like it was getting the agent to its best possible shape.

- **Comparing Actor Critic Methods**- From the beginning of this work we wanted to explore Actor Critic methods (such as A3C, A2C, PPO) and have read extensively on the PPO algorithm. In our code you can find an implementation of this algorithm. Unfortunately, we were not able to extract significant results from our implementation- perhaps due to sneaky bugs in our code or some misunderstanding of the algorithm. In the future we will definitely consider putting more emphasis on the AC approach.

Aside from the technical issues, our own learning during this word has led us to 2 significant realizations:

- **Discovering how to communicate with the agent**- Despite the extensive knowledge we have gained during the semester, it seems that there is no recipe for a clean ”out-of-the-box”, ”one-size-fits-all” RL solution, even for a relatively simple problem. The specific reward system, environment state spaces and action spaces have everything to do with the way we steer our agent towards impactful learning. Many hours of thinking and discussing are required to internalize the ways in which the agent experiences its goal and moves towards it, and more importantly, how we can influence this process in a positive way.

- **RL is art but also engineering**- Again, in spite of the countless hours of studying and internalizing done during and after the semester, RL (much like most other ML fields) eventually comes down to implementation and performance of the algorithms. We have discovered that even 8-line-long algorithms could cause a seemingly endless stream of errors and complications when attempting to implement them correctly (that's PPO for us). In the future we will definitely approach RL tasks with more regard and attention to the engineering side of the problem.

To sum up, the process has been fascinating in its ability to help us hone down our understanding of the magic of deep RL, as well as its (often hidden) complexity and the incredible human ingenuity driving it forward for the past years. We have internalized more deeply many important RL concepts seen in class through our attempts to guide the agent in the right direction (even when it seemed to enjoy mostly slowing down), and have improved in our ability to translate academic papers to code in a useful way.

5. Code

[6]

6. References

[Mnih+13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra , M. Riedmiller
"Playing Atari with Deep Reinforcement Learning". (2013)

[Has+15] H. V. Hasselt, A. Guez, D. Silver
"Deep Reinforcement Learning with Double Q-learning". (2015)

[Mnih+15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, et al.
"Human-level control through deep reinforcement learning".(2015)

[Wan+16] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, N. d. Freitas
"Dueling Network Architectures for Deep Reinforcement Learning". (2016)

[Mnih+16] V Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, et al.
"Asynchronous Methods for Deep Reinforcement Learning". (2016)

[Sch+16] T. Schaul, J. Quan, I. Antonoglou, D. Silver, et al.
"Prioritized Experience Replay". (2016)

[Sch+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, et al.
"Proximal Policy Optimization Algorithms". (2017)

[Pat+17] D Pathak ,P. Agrawal, A. A. Efros, T. Darrell, et al.
"Curiosity-driven Exploration by Self-supervised Prediction". (2017)

[Hes+18] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, et al.
"Rainbow: Combining Improvements in Deep Reinforcement Learning". (2018)

[Sut88] Sutton, R. S
"Learning to predict by the methods of temporal differences.". (1988)

[Sut+98] Sutton, R. S., and Barto, A. G.
"Reinforcement Learning: An Introduction". (1998)

References

[1] Stable-Baseline3. . URL https://github.com/DLR-RM/stable-baselines3.

[2] Phil Tabor. . URL https://github.com/philtabor/Deep-Q-Learning-Paper-To-Code.

[3] Rainbow is all you need. . URL https://github.com/Curt-Park/rainbow-is-all-you-need.

[4] Alexander Nikulin. . URL https://github.com/Howuhh/prioritized_experience_replay/blob/main/memory/tree.py.

[5] jackyoung96. . URL https://github.com/jackyoung96/RainbowDQN_highway/tree/master/Rainbow.

[6] Our code. . URL https://colab.research.google.com/drive/1GojNKKom8CX92rUJUSevVBE6yIj1aU4z?usp=sharing.