Solving LunarLander-v2 with Deep Q-Learning Methods

Daniel Kuknyo

2022 Spring Semester

Abstract

This report is based on my research regarding Deep Reinforcement Learning techniques. The paper will compare state-of-the-art reinforcement learning techniques on OpenAI gym's LunarLander-v2 environment such as Deep Q-Learning, Double Deep Q-Learning, Dueling Double Deep Q-Learning and Actor Critic methods. The task was solved using the aforementioned methods and will be focusing on the operation of these techniques using a modern deep learning library called Torchvision, acting inside the OpenAI gym interface serving as an environment for the agent. The research will compare these techniques using different methodologies and evaluate the performance of each algorithm.

Part I
Introduction

LunarLander is a whole genre of video games where the player has to land a spaceship on an uneven surface using the spaceship's thrusters. The game was originally made in 1969, the year of the Apollo moon mission, and was loosely inspired by it as the task is very similar. The spaceship has to land on a specific area given a rough surface. The designated area is marked by flags. If the spaceship lands anywhere else the agent will receive a negative reward from the environment. The goal of the agent is to execute a smooth landing based on its policy. If the landing is too sudden, or in the wrong place the task is considered to be failed. The LunarLander-v2 environment changes the roughness and shape of the surface every time for extra variance and challenge.

Part II
Related work

The basis and theory of the applied algorithms is laid out in two editions of the same book: Sutton and Barto 's Introduction to Reinforcement Learning, the 2017 unfinished draft version [4] and the 2018 finished second edition [5]. This paper uses the detailed algorithm descriptions and pseudocodes from these books as guides and baselines. The techniques and structures were implemented using the techniques described in the O'Reilly book called Hands-on Reinforcement Learning [2]. The algorithms and data structures were implemented using the Torchvision library and by the guidelines in this book. Lanham also describes in detail the basis of reinforcement learning and the agent-environment interface from a programming standpoint which can serve as a reference for fast and efficient learning algorithms.

The specific task of LunarLander-v2 was solved by Xinlin Yu's paper [8] where he describes neural network architectures that can solve this specific task efficiently and precisely, and also compares different hyperparameter values like activation functions of hidden layers, learning rate, network size, discount rate and specific values for the exploration/exploitation tradeoff parameter, ϵ. Yu also made some very relevant experiments regarding the memory and replay buffer size, the conclusion of which was employed in this research. There are some notable approaches covered by other authors as Deep-SARSA and hybrid approaches, in the paper by Z. Xhu [7], which proposes new approaches to measure performance on different learning environments provided by OpenAI's gym interface like mean Q-values or the standard deviation in rewards, including on LunarLander-v2.

Part III
Problem description

Environment

State space

The inputs to the agent's policy is 8 variables that describe its state: the x, y coordinates of the lander on the screen, its linear velocities in x, y, its angle, its angular velocity and two booleans that represent whether the each leg of the spaceship is in contact with the ground or not. Most of these are continuous variables, this makes the problem a continuous control task. The landing space is at the coordinate $[0, 0]$, this is where the ship has to land. Thus, the observation vector is as follows:

$$(x, y, v_x, v_y, \theta, v_\theta, left, right)$$

The spaceship starts at the top-middle of the screen at the beginning of every episode, with a random amount of force applied to its center of mass. This for the agent to learn how to balance out a force with the thrusters.

Action space

The action space is discrete, and has 4 elements: do nothing and turn on left/right/main thruster. The task for the Deep-Q models is to estimate the state-action value function for each of the possible states and actions that can be taken from the states.

Rewards

The reward for moving from the top of the screen to the landing pad and coming to rest is about $100 - 140$ points. If the lander moves away from the landing pad, it loses reward. If the lander crashes, it receives a penalty of -100 points. If it lands smoothly, it receives an extra $+100$ points. Each leg with ground contact is $+10$ points. Thrusting the middle engine is -0.3 points for each step, so it's not optimal to just fire the main thruster to land from the top. Firing the side engine is -0.03 points each frame. Solved is 200 points, and the v2 environment also counts the energy spent in the episode. More spent energy will result in less overall episode reward.

Episodes

The agent will play an episode until it lands on the pad. We can classify the task a fail if the ship crashes into the moon, gets outside of the screen (x coordinate larger than 1) or the lander is floating in a stationary position.

Agent

The goal for the agent is to agent is to figure out how to play the game in such a way that results in the most reward in the long run. The agent's action is dependent on a policy, which is a mapping between a state and an action. However, as the state space is continuous, and the variables can span in a wide range, it's not effective to create a Q-table that stores the $Q(s, a)$ state-action values explicitly. The $Q(s, a)$ value measures how good it is to be in a state s, take action a, and thereafter follow the policy π. However, this is not possible for a task like LunarLander, as it would require discretization of the state space, which would result in an ill-formed, suboptimal policy. The solution in cases like this is to estimate the $Q(s, a)$ using function approximation methods that use neural network to predict the state-action value function given a specific state, and decide on an action for the agent to take. The common in all the following approaches is that they all give estimations on what action should the agent perform, and they are iteratively improving their predictions based on the experiments that the agents are playing in the simulated environment.The only difference in the approaches is how they achieve this goal. The research will be focused on the differences and the measurement of performance of each agent created according to a specific methodology.

Part IV
Methodology

Basics

In this research I have ran and measured the performance of four deep reinforcement learning algorithms, namely Deep Q-Learning (DQN), Double Deep Q-Learning (DDQN), Dueling Double Deep Q-Learning (DDDQN) and Actor-Critic (AC) methods. In the following section the paper will explain the main differences and advantages of each method, so the reader can understand better why the results are different. These methods are used to build models that serve as a policy of an agent, helping him make decisions on what action to take at each time step t.

The basic function for the model to estimate is the state-action value function, which is the expected value of the sum of the cumulated discounted future reward, given state s_t and action a_t:

$$Q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$$

Where γ is the discount factor, and R is the immediate reward received by the agent for its action. This expectation value is the one that the model will have to estimate. Before diving into that, first we will look at another approach, which is the Bellman optimality equation for the state-action value function. This is a very compact formula that uses the dynamics of the environment to calculate the $Q(s,a)$ value in a closed-form equation:

$$Q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma V_\pi(s')]$$

Where the term $p(s',r|s,a)$ denotes the probability that the agent will receive reward r, end up in state s', given that it's currently in state s and takes action a, and thereafter following the existing policy, π. $V_\pi(s')$ is the state-value function in the following state, s'. The issue with this formula is that despite it being very compact, it might take a lot of processing power to compute in complex environments, and also that the dynamics of the environment is not known for most cases (like this one). This is why it's reasonable to turn to deep learning methods to solve task to estimate the $Q(s,a)$ value function. With the help of effective code and fast deep learning libraries, good accuracy and speed can be achieved.

Deep Q-Learning

Generic DQN

The deep Q-learning is an extension to the original method of calculating the state-action value function explicitly, using a (deep) neural network. In tasks where the state is defined by an image of the gameplay screen, the go-to solution is a deep convolutional network that processes the raw pixel data and outputs the action to be taken by the agent. However in the case of LunarLander it's possible to skip the raw image data processing and stick to the state variables provided by OpenAI. In classical Q-learning the state-action values are updated iteratively by the following rule:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r(s,a) + \gamma \max_{a'} Q_t(s',a') - Q_t(s,a))$$

Where α is the learning rate hyperparameter. and $r(s,a)$ is the immediate reward received for taking action a from state s. As stated before, this is not possible as the LunarLander-v2 environment has an infinite state space. However, the Q-learning update rule can be directly implemented using a neural network. In this case instead of the Q-table, the model will be updating the weights of the neural network called parameters. The original Temporal Difference loss will be replaced by a Loss function L:

$$L(\theta_t) = E_{(s,a)\sim D}[(y_t^Q - Q(s,a,\theta_t))^2]$$

Where y_t^Q is the target value. The term D stands for an experience buffer which contains important data on the episode like state, action, next state, reward and dones (booleans to tell if the agent has terminated with either win or fail). The experience buffer is sampled uniformly each time the model updates its weights and then emptied out. This originally was the idea of Long-Ji Jin [3] and is now widely accepted as a method of improving the model's performance. The loss function is calculated based on the sample and gives the experience for the agent to learn upon. This function measures how good the model performed on a given training iteration (episode). The procedure corresponding to the traditional Q-table update is gradient descent on the parameters of the neural network. It takes the following form:

$$\theta_{t+1} \leftarrow \theta_t + \alpha E_{(s,a)\sim D}[(y_t^Q - Q(s,a,\theta_t))\nabla_\theta Q(s,a,\theta_t)]$$

Where ∇_θ is the gradient of the loss function.

Double DQN

In classical Q-learning and DQN, the algorithm always uses the same maximum values to select an action and to evaluate that action. This is what is referred to as optimistic estimation and this is something that is better to be avoided. The algorithm for Double Q-learning was originally proposed by Hasselt [1]. In this case two estimators are used synchronously to retrieve the the predicted action value. When updating the state-action value function, one of the models is used to predict the best possible action, and the other one is used to evaluate the taken action. For this reason Double Q-learning has to store two Q-tables called Q^A and Q^B. The values from one network are copied over to the other one every defined couple of iterations. This results in lower variance and more robust models. The update rule is the same as in case of the DQN, but right now the model has to update the online network. However there's a slight change that has to be introduced as a result of having multiple estimators in the system. The target value in case of DDQN is:

$$Y_t^{DDQN} = r_{t+1} + \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a, \theta_t), \theta^-)$$

Where θ^- is the set of parameters of the target network, the one thats weights get updated every couple of iterations (defined by the programmer). The update frequency is usually denoted by τ.

Dueling DQN

Before discussing the DQN viewpoint for dueling network architectures, let's establish what the advantage function is. This measurement is one type of value function that's obtained by subtracting the state-value function from the state-action value function and tells us how good is it to be in a state s, take action a and thereafter follow the given policy relative to the other actions that can be taken at that given state:

$$A(s,a) = Q(s,a) - V(s)$$

The dueling neural network architecture was first presented by Wang et. al. [6], which explicitly separates the inner calculation of the state-value function and the state-action value function. The network's inputs are the same as in the cases before, the observations of the agent, and the hidden layers can be customized as the user wishes. However the main difference is in the output. The network will get two heads (output neurons): one of the size 1, for estimating the state-value function of the current state and another one that's the same size as the action space (4 in case of LunarLander) for estimating how good is it to take each possible action. Then after these values are calculated, the network reassembles them with the following forward mapping in order to maximize the Q-value:

$$Q(s,a,\theta,\alpha,\beta) = V(s,\theta,\beta) + (A(s,a,\theta,\alpha) - \max_{a' \in |A|} A(s,a,\theta,\alpha))$$

Actor Critic

A2C methods also use two heads for the neural networks just like dueling architectures. One is called the actor, and the other is called the critic. The critic will estimate the value function (this can be the state-value and the state-action value as well, in this experiment the state-value function was used) and the actor will estimate a policy distribution (logarithmic probability of actions) in the direction suggested by the critic. In advantage actor critic methods the actor head will predict the advantage function $A(s, a)$ in the head. The update equation for actor-critic methods that uses the advantage function is as follows:

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

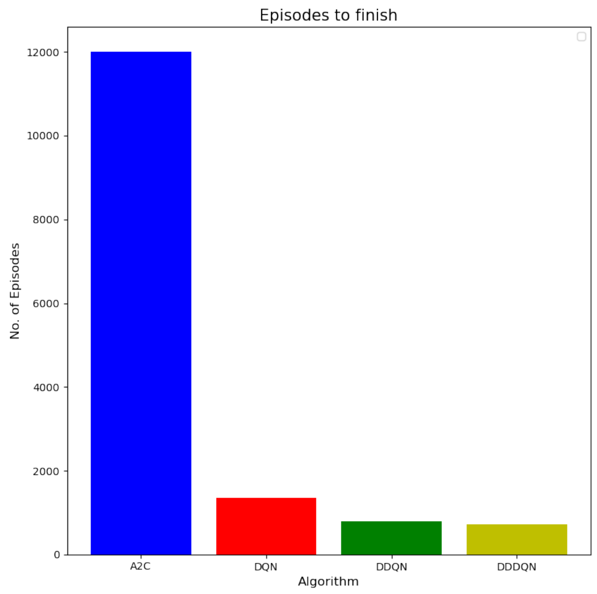
Where the term $\log \pi_\theta(a_t|s_t)$ is the logarithmic probability distribution: the probability of the agent taking each action given that it's currently in state s_t, and thereafter following policy π.

Part V
Results

According to DeepMind's GitHub page on LunarLander-v2, the task can be considered solved when the agent reaches the reward of 200 in a single episode. In this research however, the criteria for solved was set to the last 100 episodes having $average_{reward} \geq 200$. This measure is able to ensure that the models received are robust and the agent has found an optimal policy.

Time to solved

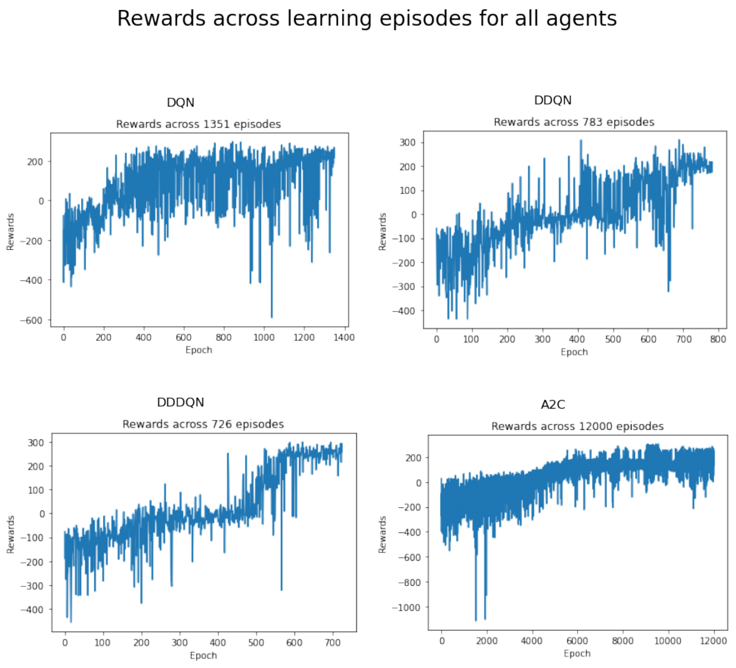
At first the time steps to completion will be investigated. It was measured how many episodes did each of the agents took until reaching the set criteria. All of the DQN variants were able to achieve this goal, with relatively similar setup. Each neural network had one hidden layer with 64 neurons, and they only differed in their heads that were necessary for the specific task to estimate a different value function or distribution. The A2C method hasn't reached convergence in over 12000 episodes however, it has reached the original goal set by DeepMind such that the reward of an episode has to be over 200. The time steps to completion were as follows:



According to the experiments, the dueling network architecture had the highest performance, with just 726 learning iterations until the average reward of the last 100 episodes converged to 200. The worst performer according to this metric was the A2C, which couldn't reach the required amount of average reward in reasonable time, so the training was stopped after 12000 episodes. It's worth noting that even the A2C has reached the goal set by DeepMind, which is 200 reward for a single episode, despite obviously being the worst performer among the bunch.

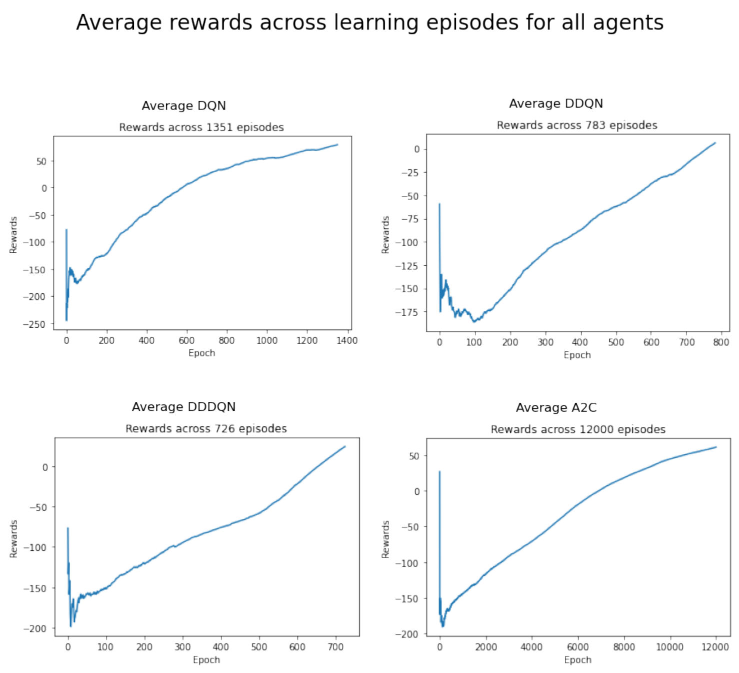
Episodic rewards

The next viewpoint of performance measurement is the rewards for each episode. This diagram plots the cumulated discounted reward for each episode of each learning agent. It's interesting to notice that the architecture that has reached the cumulated reward of 200 the soonest is the generic DQN algorithm, at around episode 300. However after this, no significant improvement can be seen in its performance regarding the reward. This model also possesses the most deviation in the rewards, e.g. the -600 score at around episode 1050. The least deviation is that of the DDDQN model's, which corresponds to the idea that multiple neural networks are supervising each other in order to avoid overly optimistic prediction values. The large deviation is a result of overly optimistic predictions, or overfitting. The largest negative reward is that of A2C's, which has successfully made less than -1000 not one, but two times.



Mean rewards

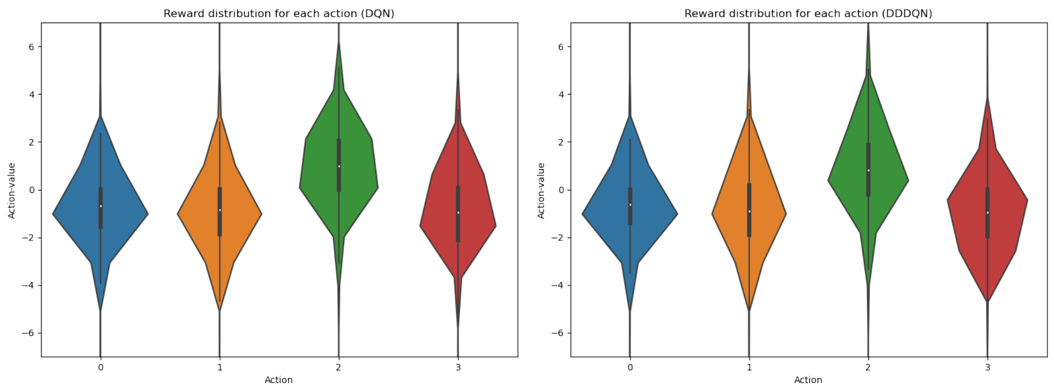
The mean of the rewards was also measured, but not in a rolling fashion. The mean reward $\overline{r}$ at time step t: $\overline{r} = \frac{r_0+r_1+...+r_t}{t+1}$. This value was calculated for every time step and plotted as a continuous function for each of the learning algorithms:



In the case of DQN, the rate of growth of the mean rewards is decaying. It can be seen that it's reaching the most of its performance as the time steps are moving along. The growth would most likely reach a plateau if the experiment was continuing for more episodes. However previous experiments have shown that it's capable of reaching an average last-hundred reward of 250 even. In the case of DDQN and DDDQN it's not hard to note that the growth is not close to stopping. These architectures could more than likely be capable of getting to a last-hundred reward of over 300. The A2C method has started off with a very strong negative reward but then it started growing steadily. The episodic rewards plot shows that the rate of growth reaches a plateau. The plot can fool the eye at first, because the scale of the x axis spans in a much wider range than that of the DQN variants'. The mean reward turned positive only after episode 7000, while this number of iterations wasn't at all necessary for the other models. Most likely the A2C model would need more than 20000 training episodes to match the performance of DDDQN in less than 1000. This can probably be accounted for by the fact that the A2C implementation doesn't use experience replay with uniform sampling, but instead has an entropy term as an input for the advantage function calculation.

Distribution of rewards

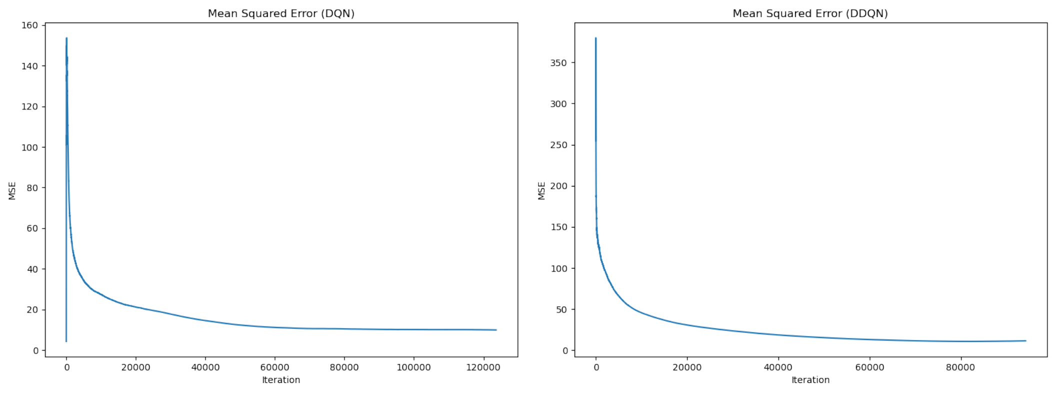
The distribution of rewards with respect to each of the four possible actions was also investigated in case of the DQN and DDDQN algorithms. The following diagram shows the distributions of the rewards, with the mean at the widest value on each violin plot.



It's interesting to note that in both cases action 3 was the one that yielded the most reward in the long run. Action 1 and 3 were roughly in the same place for both of the algorithms, and it's fairly easy to see why: they are the actions to turn on the left and right thrusters of the lander. Because of the random trajectory initiation are used in the same frequency. Action 0 yielded a larger reward in the case of DQN, which can be explained by two things. One is the hover tactic, which makes the agent receive continuous, small reward for not landing. The main thruster is used excessively over here, and it's proven that DQN has been experimenting a lot with this tactic until it has figured out how to land effectively. The other one is that DDDQN learned to land more quickly, and for this the main thruster had to be used only for slowing down, in sync with the left and right thrusters.

MSE

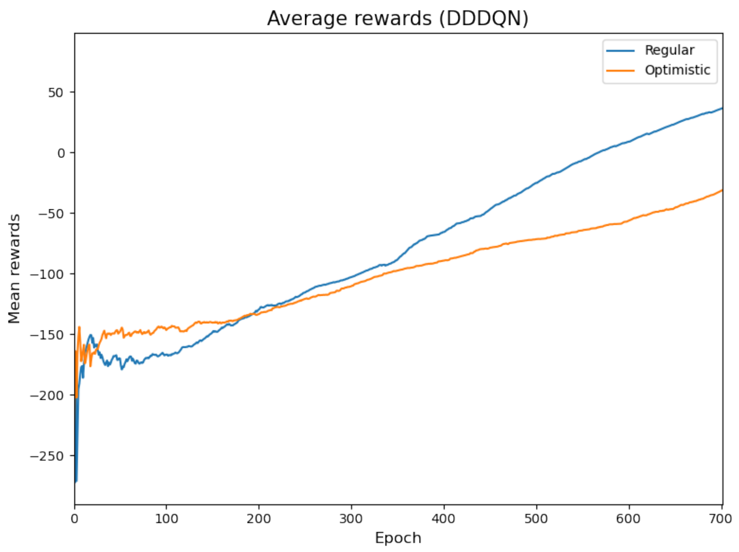
Mean Squared Error was measured on DQN and DDQN, then averaged out according to the aforementioned methodology ($avg(r_0...r_t)$ for each time step t) in both cases. The error follows a steady decrease, which is expected from an agent that learns to solve its task iteratively. There are two plots as the datasets are different in length and visualizing them on the same plot would require distorting the values.



It can be shown that the DQN architecture took a longer time to reach a stage of low MSE, while the DDQN started off with higher errors, but could successfully decrease the error in a shorter amount of iterations. Keep in mind that the iterations here mean number of error calculations, not the number of training episodes.

Optimistic initialization

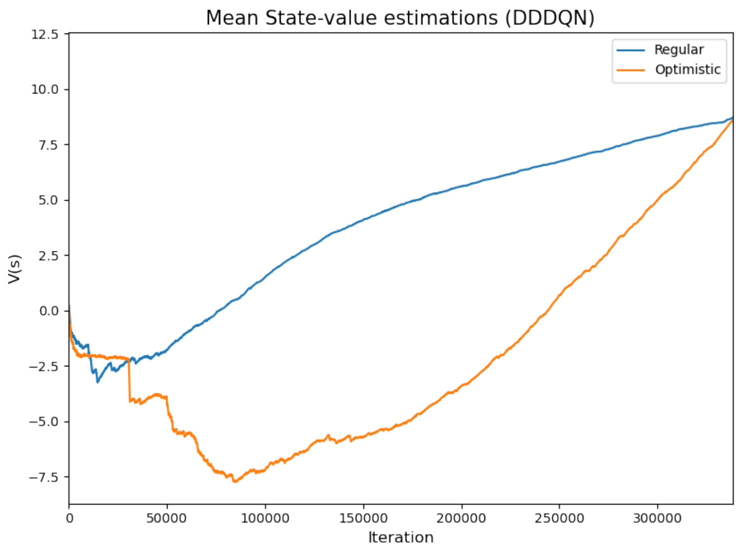
Optimistic initialization means giving the agent some large positive initial state-action value in order to encourage exploration. In the case of LunarLander all action values were set to +200, which counts as a fairly good score in a single episode. In this case whichever action is selected at first, the reward is less than the starting value. This effect will result in a "disappointed" agent, thus switching to other actions (explore). As a result the agent will try all actions several times to see which one is the best suited for the task. In the beginning the optimistic method performs worse because it's exploring more, but eventually it starts to perform better because the exploration will decrease over time. This was tested on the DDDQN network for execution speed. The result can be seen on the following plot:



It can be seen that in this case the result was exactly the opposite as that of the expected. Optimistic initialization is not well-suited for stationary problems, because exploration is temporary here.

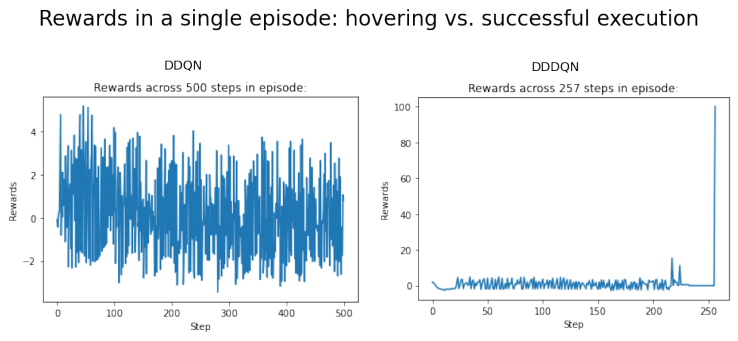
State-value estimation

The state-value function in the case of the Dueling architecture was measured and averaged out with rolling mean, with both optimistic and regular initialization as well. The optimistic initialization started out even better than the regular one, but took a dip in value very quickly, and for the most part was worse than the realistic method. It has however reached a quicker rate of growth at around iteration $150k$. It has matched the performance of the non-optimistic method at around iteration $350k$, and would probably overcome its value in the upcoming iterations. However the task was solved by that time, so this is not part of the plot. The results are the following:



Other metrics and notes

It's interesting to note that in some cases the agent has realized that if it hovers a few meters above the ground it will receive rewards for being close to the landing pad and above it, and it kept receiving positive rewards for this state. In later stages of the training however, it has learned that the energy spent by lander counts as a negative term in the rewarding scheme, so it started landing as soon as possible. The maximum number of steps inside an episode was set to 1000, but tests were run with 500-step episodes as well. Let's compare the rewarding timeline of a training episode where the agent was hovering vs. one where the landing was executed successfully.



On the left is a single training episode rewards from the experiments ran on the DDQN network. In this case the Lander is hovering above the ground, and keeps on receiving rewards of ± 3 for its actions. This is proven to be a suboptimal strategy as the agent receives a large positive reward for successfully landing both legs on the landing pad, as it can be seen in the right side subplot. Note that it took the DDDQN a lot less steps to complete the task, only 250, as opposed to the 500 steps of the DDQN.

For more information on the experiments, implementation and rendered videos of episodes please open the Jupyter notebooks included with the project.

Part VI
Conclusion

In this project multiple deep Q-learning algorithms were implemented and tested on OpenAI's LunarLander-v2 environment. It was shown that there are multiple factors that can influence the performance of a deep Q-learning agent like the usage of experience replay, network architecture such as dual or dueling networks. Target networks have vastly helped in bringing robustness into the system by avoiding overly optimal predictions. Although double and dueling architectures used in the research perform significantly better than the generic and AC methods, there's still room for improvement like in the neural network architecture or creating new, custom variables from the existing ones using neural feature extraction or dimensionality reduction. Also some thought could be given to custom loss functions such as a custom capped Huber loss instead of MSE. As a conclusion regarding the libraries used, Torchvision has again proved itself to be a state-of-the-art library capable of doing everything that a programmer could require.

References

[1] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[2] Micheal Lanham. Hands-on reinforcement learning for games : implementing self-learning agents in games using artificial intelligence techniques, 2020.

[3] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.

[4] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction, draft 2017 nov.5. *Trends in Cognitive Sciences*, 3(9):360, 2017.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Leaning*. 2018.

[6] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[7] Zhi-xiong Xu, Lei Cao, Xi-liang Chen, Chen-xi Li, Yong-liang Zhang, and Jun Lai. Deep reinforcement learning with sarsa and q-learning: a hybrid approach. *IEICE TRANSACTIONS on Information and Systems*, 101(9):2315–2322, 2018.

[8] Xinli Yu. Deep Q-Learning on Lunar Lander Game. (May):5, 2019.