Abstract

The Q - learning algorithm is applied to a simulated warehouse environment. The effects of different parameter combinations (learning rate, discount factor, policy) are analysed and results provided.
In the Second Part we first implement vanilla Deep Q-learning to a second problem and compare and analyse the results to a number of modifications suggested in literature. We finally compare the results with our implementation of a Soft Actor Critic model.
In the Third Part we implement Asynchronous Advantage Actor Critic (A3C) to solve an Atari game.

1. *Q-learning and Environment*

Q -learning [1] is an off-line learning, model-free algorithm which seeks to find the best action, given a current state to get the largest expected reward. Given a state s and an action a at time t the function $Q(s,a)$ is recursively updated according to the following formula:

$$Q(a) \leftarrow Q(a) + \alpha\left(r + \gamma \cdot \max_{a}, Q(a') - Q(a)\right)$$

Where α defines the learning rate and γ defines the discount factor.
The Environment is adapted from the well known Taxi cab problem [2]. An illustration is provided on Fig 1. The environment is simulated on real-world situations where the agent, a warehouse robot, picks-ups a shelve from one location and delivers it to another. The environment is abstracted into a 5x5 grid where each cell is a position where the agent can deterministically move . There are four pre-defined cells, labelled with the letters R, G, Y, B as illustrated in Fig. 1. The task of the agent is to pick up a shelve from one of these four cells and drop it off to another. Cells labelled with O stand for obstacles representing shelves already in the environment. The agent can only move below these obstacles if it is not carrying the target (simulating the way robots move under shelves to pick them up in many large warehouses). The agent starts at a random cell each episode. The episode ends when the agent successfully picks up the target shelf and delivers it to the target location. There is only one target to pick up and one target destination each episode.

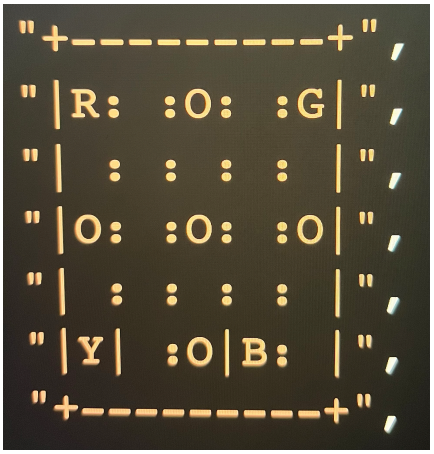

Figure 1: A representation of the Q-learning environment

The action space is defined by 6 discrete actions namely Up, Down, Left, Right, Pick-up, and Drop. Each action accounts for 1 time step.

2. *State - transition and Reward function.*

Each state is defined by a tuple in the form [row, column, target shelf location, destination location]. There are 5 rows and 5 columns in the environment. The destination and the target shelf location cell will be one of the four lettered cell. There is one additional location for the target shelf representing when it is being carried up by the agent.

Hence in total there are a total of 500 states (5 rows x 5 columns x 5 shelf locations x 4 destinations) which define our state space. The state transition function is given by $P(s' \mid a, s) = 1$ as our environment is deterministic.

The reward function is defined by $R(s,a,s')$. Each action accounts for one time step and agent receives a reward of -1 for every time-step unless one of the following applies:

bumps into O cell while carrying target : -2,
pick-up at right location: 10,
drop-off at right location: 20,
pick-up at wrong location : -10,
drop-off at wrong location : -10.

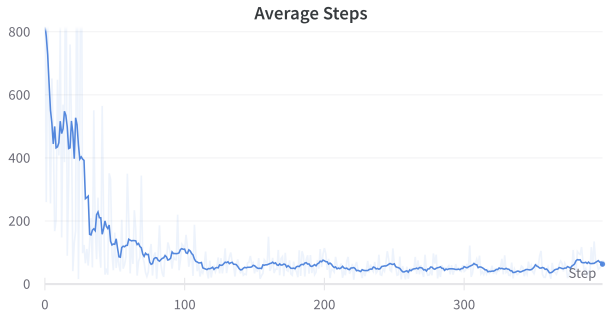

Figure 2: Illustrates the running average reward and steps per episode. The agent starts to converge around 50 episodes showing fully learned behaviour after around 100 episodes.

3. Q - learning initial parameters

To run the experiment, the policy and a number of parameters need to be set. A policy is a mapping π, at time t, from states s to probabilities of selecting each possible action [1]. Formally π (a|s) is the probability that A = a if S = s at time t for a in A and s in S. An epsilon-greedy policy is set: the agent takes a random action a with probability e or greedy action with probability 1-e.

The parameter Alpha is the learning rate which controls how much the latest update overrides the old update. Gamma is the discount factor which controls how much the agent values rewards received earlier than those received later.

The parameters are set as epsilon = 0.7, gamma = 0.7 and alpha = 1

5. Tuning parameters

Fig 3. illustrates the returns and steps per episode (10 episode running average for both) for different gamma values keeping epsilon and alpha constant. Lower gamma values provide greater stability with the learning of the agent, while higher values are more unstable but converge and receive a higher average reward earlier. This might be because our agent receives a positive reward only the end of the episode. So the lower the decay rate of a reward from the final, successful state is beneficial in our environment.

The effects of different Alpha values are seen in Fig 3. The values generally follow the same trend, however alpha values less than 0.5 clearly converge later and complete episodes with more steps. As our environment is deterministic a high alpha value works better as there is no stochasticity in the rewards and state transitions.

Fig 4 illustrates the average returns and steps for different policies. Generally the trend follow the same patten with higher epsilon values naturally bringing about a great variance in the rewards received. As our environment is generally small, a lower epsilon value will be a better fit to exploration-exploitation trade-off. The best policy is clearly using a decaying epsilon (reduce epsilon after every episode: here we use a decay rate of 0.005) which provides the largest convergence rate with the least steps per episode.

6. Results

The environment and the model are best suited to a higher alpha value as receiving the reward at the end requires that the agent receives more feedback from initial states for quicker convergence (least steps per episode). The learning rate (alpha) represents the amount of weight assigned to last update. A value of alpha of greater than 0.75 (and less than 1) will work well as shown by the previous experiments keeping the average number of steps low and the agent still receiving a good steady reward per episode.

A higher gamma value is also optimal for the agent to receive quicker feedbacks due to the nature of the environment with a delayed reward function. As the problem horizon is longer, a a delayed gamma suits the environment more. A gamma less than 1 has been shown to work well.

An epsilon decay policy has been shown to be superior. This controls the rate by which the minimum epsilon value is reached. Optimally the agent should explore more in the initial episodes (hence higher epsilon value and rate of exploration) and use the learned experience for most actions in later episodes.

The agent converged to the optimal policy eventually with most optimal policy.

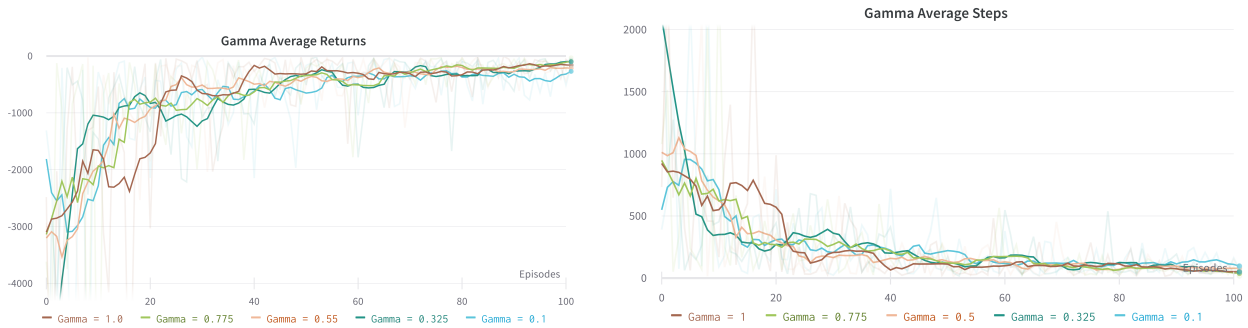

Figure 3: The running average of the reward (left) and number of steps (right) per episode for different gamma values

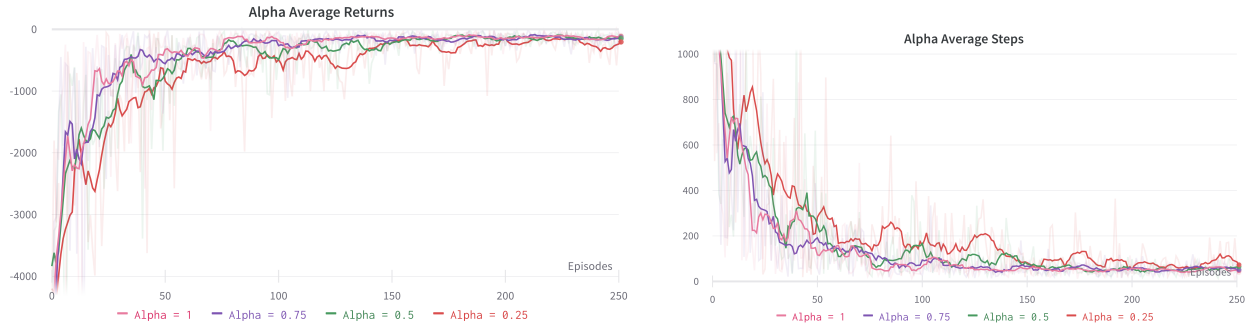

Figure 4: The running average of the reward (left) and number of steps (right) per episode for different alpha values

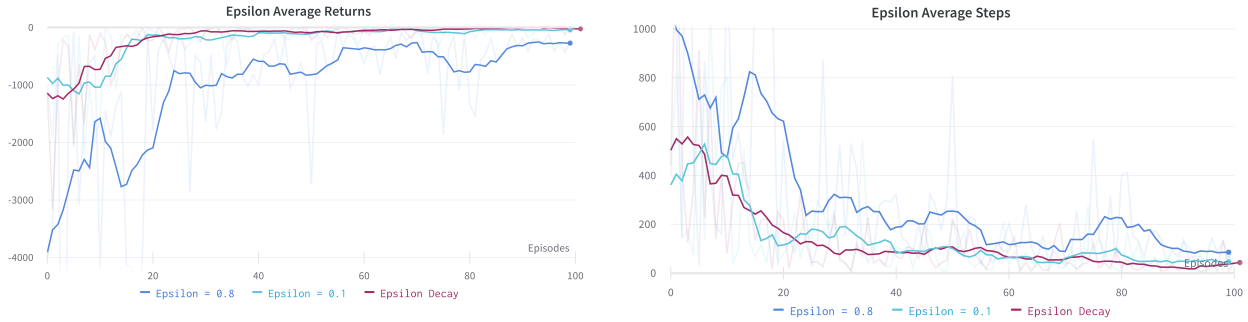

Figure 5: The running average of the reward (left) and number of steps (right) per episode for different epsilon values

This suggests that any sensible parameter configuration would work with this environment. Arguably this is because of the not complicated nature of the environment. However as has been shown different parameter combinations is important for the rate at which the agent converges to the optimal policy. A relatively high gamma and alpha with an epsilon decay policy will be able to make the agent converge with the least amount of steps per episode.

Part 2: Advanced

We apply the Deep Q learning algorithm to the Lunar Lander environment and analyse the results. We further train the agent (with continuous action space) on the Soft-Actor Critic algorithm. The results analysed thereafter.

7. Deep Q-learning

DQN extends the Q-learning algorithm by using a neural network as a functional approximate for the state value functions. If the combinations of states and actions are too large, the memory and the computation requirement for *Q-learning* will be too high.This allows the Q learning algorithm to scale up and generalise to environments with infinitely many states or actions.

However using a function approximate brings about it's own problems: the DQN algorithm is usually implemented with an experience replay mechanism which randomly samples from previous experiences so the learning phase is logical separate from the gaining phase.

Double DQN:

The DQN algorithm has been known to overestimate action values [3] as it uses the same values for both selection and evaluation. Double DQN decouples this by using a second set of weights for the target network.

In DQN, the Q-values based on parameters and the max is taken over actions based on these Q-values.The problem with this is that it leads to an overestimation bias, especially at the beginning of the training process, where the Q-values estimates are noisy.

Double Q learning decomposes the the max operation in the target into action selection and action evaluation. Two functions Q1 and Q1 are independently learned and one is used to determine the maximising action and the second to estimate the value [7].

Duelling DQN (DDQN):

[4] present the novel duelling architecture which explicitly separates the representation of state values and state-dependent action advantages via two separate streams.

The key motivation behind DDQN is that it leads to greater generalisation. DDQN has two streams to separately estimate state value and advantages of each action. The two streams are combined to produce an estimate of the state value function. This way the model can estimate which states are valuable or not, without having to learn the effect of each action for each state.

Environment

Lunar Lander is a reinforcement learning environment extensively used in literature. The environment stimulates a situation where a lander needs to land on a landing pad. [fig 2]. The state space is continuous. There are four actions available in each time step namely fire main engine, fire left engine, fire right engine, and do nothing. The reward is designed so that the agent needs to land as quickly as possible while keeping a an upright position. The agent always starts at the same position every episode.

We expect the double and duelling to do better in this environment as the goal of the agent is to land to a small section of the environment. Hence actions do not always effect the environment in meaningful ways and this this the type of generalisation estimates these two improvements solve.

1. Results

The agent was trained for 100,000 steps in the environment. An epsilon- greedy policy was used with the epsilon starting from 1 and a decaying factor of 0.0005 each step. The batch size was 32 for both models and the target update was every 200 steps. The average reward and average loss curves are shown in Fig 5. Both Double and Duelling trained the agent well with the the average reward functions showing similar trends. Overall Duelling was more stable with a linear rise in the reward function and a loss function which was steep in the beginning but stabilised as the training continued. The Double on the other hand learned quickly and was on the whole reliable expect for the dip in the middle. This might be just because our experience replay is set at an un-optimal level which leads to moments of catastrophic forgetting: the agent learns only experiences success and those are the samples saved in memory and the agent predicts high values for everything. Further experiments with the size of the replay buffer and the time update interval for the target network os necessary. A further improvement for the Double agent could be to integrate a prioritised experience replay as which will provide more stable learning than random sampling as the agent will have always have a few examples of what not to do. The Duelling agent with greater generalisation ability due to the decomposition of the state-value and advantage function is less prone to such forgetting.

As is in clear in Fig 5. the Duelling architecture had a greater decrease in the loss curve and the loss stayed at a more stable rate thereafter. This implies there was greater learning from the Duelling agent: The double agent has a pretty stable loss curve which suggests slow and incremental learning.

Even thought the agent was able to complete learning with these modifications in a reasonable amount of episodes. Performance could be improved.A further line of enquiry would be to run multiple runs on these two algorithms to get a better estimate of their training quality. Moreover Prioritised Express Replay could also be implemented so that important transitioned could be replayed more frequently and learning could be improved.

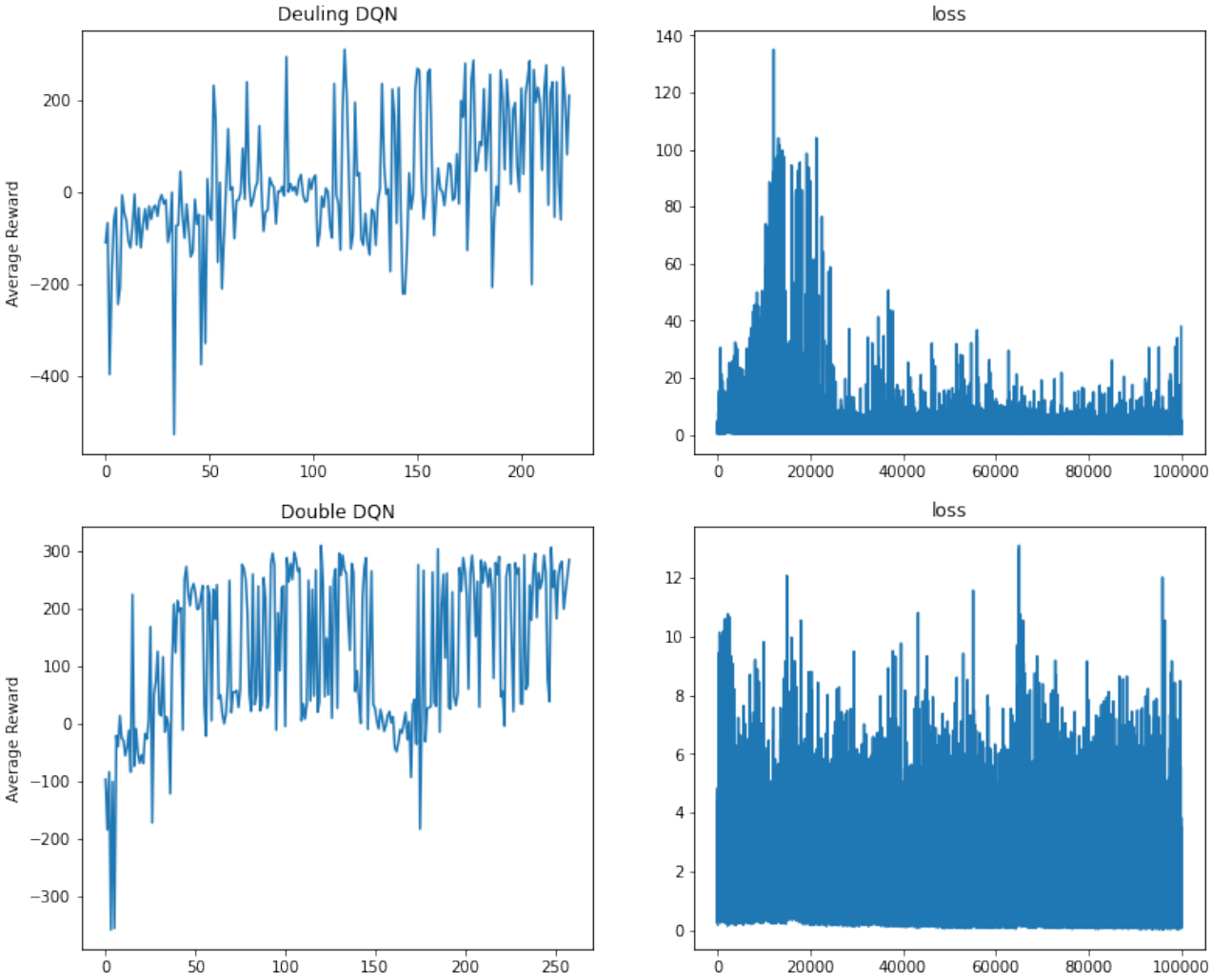

Figure. 5 Average Rewards/500 episodes and loss curves/steps for Double DQN and Duelling DQN

Soft Actor Critic.

In the Soft Actor Critic (SAC) algorithm the agent aims to to maximise expected reward while also maximising expected entropy. A policy network is used to predict the mean and standard deviation of each action and sample is taken from a Normal distribution from these values. Hence the policy is trained to maximise the trade-off between expected return and exploring as much as possible. This leads to accelerated learning [8].

The average reward curve for an implementation of the Lunar Lander environment with a continuous action space is illustrated in Fig. E1. The agent was trained on 100,000 steps in the environment for greater comparability with the DQN agents. As can be seen from Fig. E1 the algorithm worked well on this environment with a the agent getting seeing a steady increase in the average rewards. The agent sees a steady increase in the average rewards per episode for the first 50 episodes and the reward curve mostly converges after 100 episodes. The fluctuations thereafter are mostly due to the exploration in-built in the algorithm. However the reward curve remains mostly stable.

ATARI

9. Pong on A3C

We applied the A3C, **Asynchronous Advantage Actor Critic,** algorithm for on the Atari Pong game. A3C is a policy gradient algorithm which maintains a policy and an estimate of the value function[6]

The main advantage of this algorithm is that several agents run in parallel, each with their own copy of the environment. This decreases correlation as different agents will experience different states and transitions. It is also memory efficient as samples don't need to be

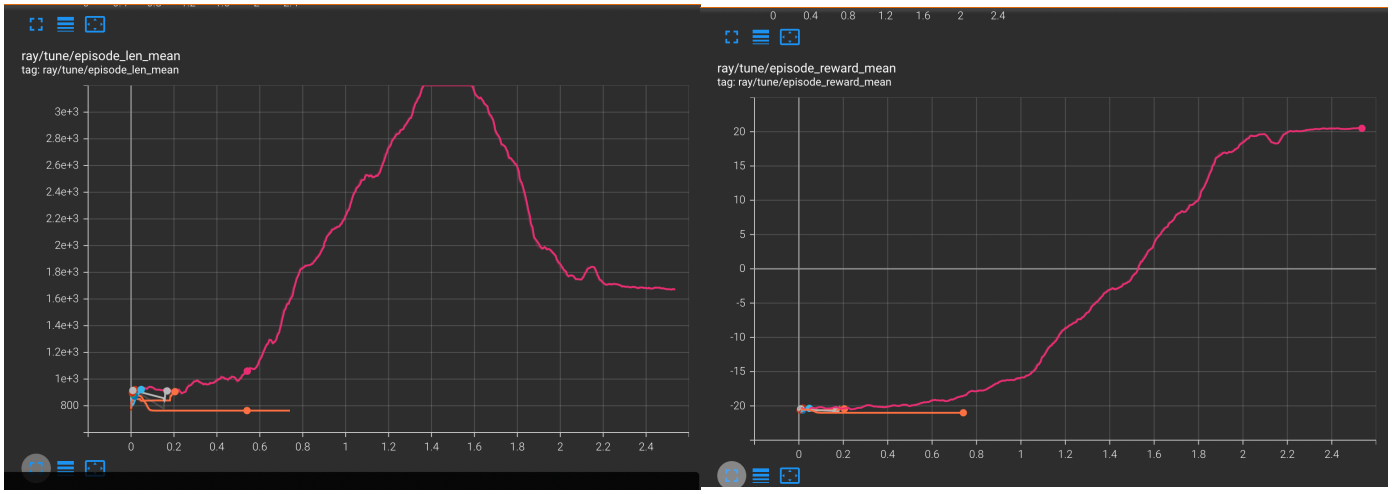

Figure 6: The mean reward and episode length for A3C

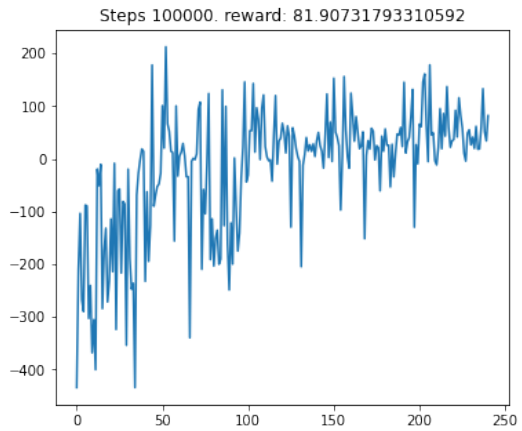

Figure E1: Average reward per episode for the SAC agent.

saved as the agents use them to learn as they arrive.

This algorithm has been proven to give good benchmarks on many games. However the main justification of using this algorithm was that it can train Asynchronous amongst several CPU's thus saving training time. Each worker runs independent runs of the environment and use their samples as they arrive. This also leads to a less memory requirement as the samples do not need to be stored. One main master computes the gradient update and transfers the new parameter weights to the workers. The driver will then use the new model updates to run the simulations. We used the free tier of the goole cloud cluster to train the agent with 8 vCPUS. The hyper-paramters were mostly default configurations on RLLIB after running a hyper parameter search for an hour. We also used the RLLIB pre-processing filter for Atari games. An LSTM framework was set for the Neural net, the activation functions were RELU, the learning rate was set at 0.001, gamma was set at 0.99 , the images were converted to greyscale. Further details are in the coding file. The training was then run for 3 hours. The results are illustrated on Figure 6.

10. Results

The training results are illustrated in Fig. 6. The training starts off slowly with small episodes lengths (game-over) and negative rewards. However this might be because of the delay in updating weights in an asynchronous algorithm. Once the workers had simulated a number of batches and the learner had concerted and updated weights from the individual workers the reward curve starts to increase in a linear fashion. The episodes lengths also starts to increase side-by-side as the agent learns to stay alive for some time. The episode lengths then start to decrease as the agent learns to optimise its play by defeating the AI agent quickly with the reward reaching a maximum score of 20. This suggests the agent learns the best actions for the respective states.

References

[1]R. Sutton and A. Barton, *Reinforcement learning*.

[2]"Gym: A toolkit for developing and comparing reinforcement learning algorithms", *Gym.openai.com*, 2022. [Online]. Available: https://gym.openai.com/envs/Taxi-v3/. [Accessed: 24- Apr- 2022].
[3]H. Hasselt, "Double Q-learning", *Papers.nips.cc*, 2022. [Online]. Available: https://papers.nips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html. [Accessed: 24- Apr- 2022].
[4]"Gym: A toolkit for developing and comparing reinforcement learning algorithms", *Gym.openai.com*, 2022. [Online]. Available: https://gym.openai.com/envs/LunarLander-v2/. [Accessed: 24- Apr- 2022].
[5]V. Mnih et al., "Asynchronous Methods for Deep Reinforcement Learning", *arXiv.org*, 2022. [Online]. Available: https://arxiv.org/abs/1602.01783. [Accessed: 24- Apr- 2022].
[6]"Papers with Code - A3C Explained", *Paperswithcode.com*, 2022. [Online]. Available: https://paperswithcode.com/method/a3c. [Accessed: 24- Apr- 2022].
[7]J. Janisch, "Let's make a DQN: Double Learning and Prioritized Experience Replay", ヤロミル, 2022. [Online]. Available: https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/. [Accessed: 01- May- 2022].
[8]"Soft Actor-Critic — Spinning Up documentation", *Spinningup.openai.com*, 2022. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/sac.html. [Accessed: 01- May- 2022].