Solving the LunarLanderContinuous-v2

Shoval Frydman (ID. 211777719), Aviv Alfandary (ID. 201133337)

Submitted as final project report for the RL course, BIU, 2021

1 Introduction

There are two different types of Reinforcement Learning (RL) algorithms: Model-Based RL (algorithms which use a model) and Model-Free RL (which do not). LunarLanderContinuous-v2 environment, has a continuous observation and action space with no model provided, therefore needs to be solved using a Model-Free methods, that can be divided into three main approaches: (Deep) Q-Learning (QL or DQL), Policy Optimization and a combination of them. In this work, we are going to solve the LunarLanderContinuous-v2 open-ai gym environment (1). First, we will elaborate on the different algorithms, alternatives and tricks we have tried and used (both theory and implementation). Second, we will find the best algorithm to solve the environment as fast as possible, compare running times and show appropriate graphs. Third, we will add uncertainty to the position of the agent and see how the best model with the fine-tuned hyperparameters reacts. Last, we will discuss our results and conclusions.

2 Solution

2.1 General approach

In this work, we chose to use RL algorithms based on DQL: DQN (2), Full-DQN (3), Double-DQN (4) and Dueling-DDQN (5), and algorithms which combine both DQL and Policy Optimization: SAC (6) and DDPG (7).DQN is efficient but it struggles to converge, thus we chose to use some variants of it for stability and efficient sampling. In addition, DDPG and SAC are algorithms that use both Q-Learning and Policy Optimization approaches advantages, therefore we assume they will have better performance. Our preferred approach to solve the environment is SAC for several reasons. First, it solves the environment as described below. Second, as the paper claims, it is not as sensitive to hyperparameters as the other methods, and converges pretty fast even without fine-tuning. Last, using the soft Bellman-Equation, the policy can have a wider exploration while giving up on clearly unpromising avenues (6).

2.2 Design

Our code is implemented using pytorch. At first, DQNs and DDPG had a hard time converging and were not stable with the initialized hyperparameters, therefore we have used a toolkit named "NNI" (8) to find the optimal hyperparameters (will be explained in details later). Note we have implemented DQL off-policy, temporal-difference algorithms: DQN, Full-DQN, Double-DQN and Dueling-DDQN, that are advanced versions of each other: DQN is the basic, Full-DQN is a DQN with target network, Double-DQN is a Full-DQN with double estimation and Dueling-DDQN is a Double-DQN with a dueling architecture. In addition, combined approaches were also implemented: SAC and DDPG, each with a different version of an actor-critic framework. Furthermore, at the beginning, we have also tried Policy Optimization based methods such as PPO (9), however despite its claim of being stable and easy to converge, even after strong fine-tuning of hyperparameters, the model has not converged for even 5000 episodes. Therefore, we will not present it in this work and will focus on the mentioned methods. Last, we will refer to running times in the experimental results (third section).

2.3 Methods

This section will start first with a short explanation of common frameworks we have used for different algorithms. Second, a short description of each algorithm, its implementation, learning process, differences and technical challenges we came across with. And last, a description of the training process which is quite similar to all algorithms. The hyper-parameters and networks architecture of all methods, are available at sub-section 3.1.

2.3.1 Common

Replay Memory - Learn from Experience. Experience replay is a tuple which stores the agent's action-observation through the steps of each episode. This tuple consists the current state, current action, reward, next state and whether

we are done or not, in a limited-length buffer known as replay memory. So that samples are drawn randomly from this memory. This demonstrates the balance between the exploration and the exploitation and is used for all algorithms. (10).

Epsilon-greedy and Noise. Epsilon-greedy or the greedy action (for discrete action spaces) and noise addition (for continuous action spaces), are common approaches which helps algorithms to explore their environments more effectively. In the DQNs, where the action space is discrete, epsilon-greedy means that given ϵ, sample a random number. If it is less than the current ϵ, choose a random action. Else, choose the greedy one. This ϵ is not fixed, but is changing each episode using epsilon decay. Usually, an exponential decay is used, however in our experiments the model did not succeed to converge, even after strong fine-tuning of hyper-parameters. Therefore, other decay options were tested and power-law decay seems to perform well and the models converged fast. For SAC and DDPG which assume continuous action space, a different type of noise is used for each algorithm and will be explained in details bellow.

Actor-Critic Framework. Actor-Critic is a common and useful framework that is used in methods that combine DQL and Policy optimization approaches with a continuous action space. The main idea is splitting the model in two: (a) Actor- A neural network that produces the best action for a given state. (b) Critic- A neural network which receives as input the state and the action from the actor, concatenates them and outputs the Q-value for the given pair. By learning the optimal policy (policy-based), the actor essentially controls how the agent behaves. By computing the value function (value-based), the critic evaluates the action. As the time passes, both models get better on their own role and the overall architecture is more efficient than the two models alone. The training of these networks is performed separately and uses gradient ascent to update both their weights and it accrues at each step and not at the end of the episode. This is the TD learning part- methods using it sample from the environment, and perform updates based on current estimates. (11).

From Continuous to Discrete Action Space. In this project we are handling a continuous environment. While SAC and DDPG are models which assume a continuous action space, the DQNs require a discrete action space, therefore a discretization was applied. To do so, we took our continuous action space which consists of two continuous elements in the range of [-1, 1], and with respect to its minimum and maximum, each element of the action space has changed so that it had 3 integers options that define its discrete value. Hence, 9 different combinations of the two action space elements were created in total. Next, we flatten the combinations of the multi discrete space so that each combination will be represented as a single discrete value. Finally, we have gotten a discrete action space with 9 different actions. (12)

2.3.2 Deep Q Network (DQN)

Explanation. DQN approximates a state-value function in a QL framework with a neural network. Instead of keeping the possible states and predicted Q-values in a table, the neural network takes as input the current state of the environment and outputs the predicted Q-values for each possible action. DQN is usually used with an experience replay. (2).

Code. DQN is implemented with a policy neural network and experience replay. In the learning process, experiences are sampled from the memory. Current and next Q-values for each action are predicted using the policy network. The target Q-value is calculated using Bellman-Equation with discount factor. Last, the Huber loss (13) between the current Q-values and the target Q-value is computed, and back-propagation and optimizer update are done. (14).

2.3.3 Full DQN

Explanation. The implementation of the standard DQN consists a single neural network in which both the Q-values and the target-values shift during the Q-learning update, leading to an unstable model struggling to converge. Therefore, a well-known improvement is to use a frozen target network, which is a clone of the standard policy network, and is used to calculate $Q(S_{t+1}, a)$ during the update step, i.e. every k updates (k is a hyper-parameter) the target network is cloned once again and the weights are refreshed. This makes the training more stable by preventing short-term oscillations from a moving target. We call this version of DQN "Full-DQN". (3)

Code. The Full-DQN has the same architecture as the DQN, it's agent inherits from the DQN agent, it has an addition of a target network. The learning process differs from DQN by having the back-propagation and the optimizer update, every k updates (hyper-parameter called "target_update") the policy network is cloned to the target network. (14)

2.3.4 Double DQN (DDQN)

Explanation. The DQN and the full-DQN choose an action with the highest Q-value for the next state, which may lead to an overestimation of the Q-values, and as a result an unstable model or a slow convergence. DDQN deals with this problem by making a double estimation: (a) The greedy policy is evaluated using the online (standard) policy network (b) The target network is used to estimate its value. Then, these estimations are used to compute the final Q-value. (15; 4).

Code. The DDQN's agent inherits from the Full-DQN agent, i.e. it has both policy and target networks. In the learning process, experiences are sampled from the replay memory, and current Q-values are estimated using the policy network. The next Q-values are calculated using the policy network by the next states and the maximum Q-values are chosen and

evaluated using the target network. This is the double estimation. Then, the overall Q-value is computed by Bellman-Equation, Huber Loss (13) is computed between the final Q-value and current one, and a back-propagation and optimizer update are done. Last, as in the Full-DQN, every k steps the policy network is cloned to the target network. (14)

2.3.5 Dueling Double-DQN (Dueling DDQN)

Explanation. It is unnecessary to know the value of each action at every step. The advantage $A(S_t, a_t) = Q(s_t, a_t) - V(s_t)$, subtracting the value-function from the Q-function Practically obtains a measure of the relative importance of each action. The Dueling network consists two streams that separately estimate state-value and the advantages for each action. They share a convolutional feature learning module, and are combined via a special aggregating layer. For each state, values and advantages are calculated and are then merged in a unifying layer. (5).

Code. The learning process is identical to the DDQN's, with the Dueling architecture which was described above. (14)

2.3.6 Soft Actor Critic (SAC)

SAC framework(6) is an off-policy deep RL algorithm. The actor maximizes both expected reward and entropy of the policy which is a measure of randomness in the policy, with a trade-off between them. Increasing the entropy yields more exploration, encouraging the policy to assign equal probabilities to actions with same or nearly equal Q-values and to ensure convergence. This can also accelerate the learning process. In addition, Experience replay is used here too. (16).

Framework. In practice, the SAC framework consists of five networks: Two soft critic networks for the soft Q-function, two for their target network, and an actor network for the policy approximation. The two critics and actor networks are trained separately with three different optimizers. Furthermore, the temperature parameter α that is related to the entropy, is not fixed, and has too an optimizer of its own.

Soft Critics and Their Target Networks. The algorithm makes use of two soft Q-functions (soft critics) to mitigate positive bias in the policy improvement step that is known to degrade performance of value based methods. In particular, the two soft critics are parameterized and trained independently to optimize the soft Bellman residual. Then, they use the minimum of the the soft Q-functions for the stochastic gradient for the parameters update and policy gradient. Two soft-critics are used because they speed up training. In addition, each soft-critic has its own target network with the same structure as their critics, and they are used for cloning and improving stability (same role as explained in Full-DQN). In our implementation, we initialize the critics and their targets to have the same parameters, and update the target network by Polyak averaging (a kind of moving averaging) itself and the main network. (6).

Actor. The actor network returns the action and the predicted policy. In SAC, the action is calculated by performing a reparameterization trick that is used to make sure that sampling from the policy is a differentiable process so that there are no problems in backpropagating the errors. Given a Gaussian distribution that is gotten from the forward process, the mean and standard deviation are used to calculate the log of the policy and the wanted action. In practice, the actor network takes as input a state and consists of: Two common linear layers, each followed by a non-linear activation function. The output is inserted as input to two different and separate linear layers: (a) a linear layer to get the mean (b) a linear layer to get the log standard deviation, both are parameters found to get a Gaussian distribution that is needed for the objective function the actor network aims to minimize. The log standard deviation we received is clamped to be in a same region. Then, the reparameterization trick is used: The mean and std (exponent of the log std) are parameters for a Gaussian distribution from which we sample randomly a noise vector. This noise is multiplied by the std and the mean is added to the result. Last, tanh non-linear function is applied on this term to finally find the action and in the training process, it is also normalized by the action space range. Furthermore, the log policy is predicted too using the sample Gaussian noise, calculated action and epsilon (a noise hyperparameter). (16).

Entropy - The temperature parameter fine-tuning. The temperature parameter α has an optimizer of its own because it is automatically tuned during the learning process. As explained in the paper, without it we might suffer from brittleness. It is updated while minimizing a loss depending on the (-log) of the policy and the entropy. The log of α is learned for numeric stability. (6).

Learning Process. As opposed to the methods listed so far, here the learning process happens every step, and there is no learning frequency parameter. The only condition is that the replay memory size will be large enough. When it is, the learning process goes as follows: Experiences are sampled from the replay memory, and the action from the policy (pi) and policy log ($\log \pi$) are predicted using the actor network. Then, we use the two target networks to evaluate the Q-values by the next states and new actions, and the target Q-value is the minimum of them minus $\alpha \log \pi$ (this is the entropy part). The expected Q-value is computed using the bellman equation (with the discount factor). Then, two Q-values are predicted using the two critics by the current states and actions, and a MSE loss is calculated between each Q-value to the expected Q-value. Back-propagation is then performed, along with the update of the two optimizers of the two critics. Again, new actions and $\log \pi$ are predicted using the actor network, but this time using the current states. If the current step is divided without residue by the delay_freq parameter (controls the frequency of the following update), the Q-values are again computed using the critics by the current and new actions, and the minimum is chosen to be the final Q-value. The actor loss is computed as the mean of the multiplication of $[\alpha \log \pi - final\_q\_value]$, and a back-propagation of the

actor loss and actor optimizer update are done. Then, we copy the current weights of the critics to the target networks. Last, the temperature parameter is optimized: A back-propagation is done on its loss that is depending on $\log \pi$ and the entropy, and its optimizer is updated. (6; 16).

2.3.7 Deep Deterministic Policy Gradient (DDPG)

DDPG (7) assumes continuous action space and is an actor-critic, model-free algorithm based on deterministic policy gradient. Both actor and critic have their own target network, and they are trained separately with two different optimizers as explained earlier. Therefore, DDPG can be treated as an expansion of Full-DQN to continuous environments which combines the actor-critic framework: The network is trained off-policy using replay memory and has a target network to give consistent targets during temporal difference backups. Moreover, "Ornstein-Uhlenbeck" is added to the actor policy multiplied by epsilon which also decays over time, to generate temporally correlated exploration for exploration efficiency in physical control problems with inertia.

Learning Process. The learning process happens every few steps, only for steps that are divided without residue by k (hyperparameter). It goes as follows: Experiences are sampled from the replay memory, and next actions are predicted using the actor target by next states. Then, Q-values are predicted using the critic network by the current states and actions, and next Q-values are predicted using the critic target by the new states and new actions. The targeted (expected) Q-value is calculated using Bellman-Equation with discount factor. Then, the MSE loss between the current Q-values and the expected ones is computed, and back-propagation, gradients clipping of the critic parameters and the critic optimizer update are done. Then, the actions are predicted using the actor network by current states, and the actor loss is minus of the mean of the predicted Q-values from the critic using the current states and predicted actions. Back-propagation and update of the actor loss and optimizer are done. Last, the parameters of the critic and actor are copied to their target network, the epsilon (that is multiplied by the OU noise) is decaying, and the noise is updated as well. (17; 18; 19).

2.3.8 Training Process

Initialization. For SAC and DDPG critics and actor parameters are initialized. For DQNs, the policy network parameters are initialized. For all algorithms, if a target network exists, it is initialized according to its standard network (critic or policy), and the replay memory is now empty.

Process. For the maximum number of episodes, for the maximum number of steps per episode, choose an action using greedy-epsilon trick (for each method as explained earlier). Then, perform a step in the environment to get the next state, reward and whether you are done, and add this experience (along with the current state and action) to the replay memory. The next state becomes now the current state. If the current step is divided without residue by the learning frequency parameter, perform the learning process (except for SAC in which it happens for each step), that is different for each model and is explained in details for each model separately. Last, in the DQNs and DDPG, apply epsilon decay, i.e. reduce the current epsilon according to the decay option. Continue doing it until the environment is solved or until the maximum number of episodes.

3 Experimental results

3.1 Hyperparameters - NNI

NNI (Neural Network Intelligence) is a toolkit to help users design and tune machine learning models (e.g., hyperparameters), neural network architectures, or complex system's parameters, in an efficient and automatic way (8). For each RL algorithm, NNI was used in two steps for a broad hyper-parameter tuning. First, a grid search of a wide range of parameters was performed to get the amplitude of the regularization. The second step was to refine the outcome by setting the tuner to Tree-structure Paezen Estimator (TPE) and running another search. By the end of this process, we have found the best hyperparameters, for them we have succeeded to solve the task in the minimum amount of episodes.

3.1.1 Deep Q-Learning Models

For all models, Huber loss (13), Adam optimizer and Relu activation function were used. Maximum number of episodes and steps are 400 and 1000, respectively, and the memory (buffer) size is 1e6. ϵ ranges between 1.0 and 0.01, and decays by power-law rule. The architecture of the networks of the DQNs agents is: input layer of size [8,64] where 8 is the dimension of the observation space, one hidden layer of size [64, 64] and an output layer of size [64, 4] where 4 is the dimension of the action space. In Full-DQN and Double-DQN, the target networks are identical structured. For Dueling-DDQN agent, both policy and target networks have: a common layer of size [8, 64] which is split into two output streams- an advantage and value stream- the value layer consists of two linear layers of sizes [128, 128] and [128, 1] and the advantage layer consists of two linear layers of sizes [128, 128] and [128, 8]. Relu activation function is applied on the common layer, on the first value layer and on the first advantage layer. Other hyperparameters are in table 1.

	DQN	Full-DQN	Double-DQN	Dueling DDQN
Batch size	128	128	128	64
gamma	0.98838	0.98604	0.98151	0.98859
Learning rate	0.001247	0.003153	0.003922	0.988594
Learning frequency	2	5	4	3
Epsilon decay	0.95237	0.97807	0.96335	0.97275
Target update	-	300	400	300

Table 1: DQNs hyperparameters

3.1.2 SAC and DDPG models

For these methods, the sizes of the observation and action spaces are 8 and 4, respectively. MSE loss and Adam optimizer are used for both critic and actor networks, and the maximum number of episodes is 1000.

DDPG. The actor (and target) consist of three linear layers: input layer of size [8,128], one hidden layer of size [128,128] and output layer of size [128,4]. Relu activation function is applied on the output of the first and second layers, while tanh is applied on the output of the third layer. The critic (and target) consists of three linear layers: input layer of size [8,128], one hidden size of size [128+4,128] and an output layer of size [128,1]. Relu activation function is applied on the output of the first layer, this output is concatenated with the given action, and again a relu is applied on the output of the second layer, and this is the input to the third layer. Furthermore, for both actor and critic the weights and bias of the output layer are uniformly initialized in the range [-3e-3,3e-3]. Other hyperparameters are listed in table 2.

SAC. All networks have the architecture described in the previous section: (a) Two soft critics and their target networks: input layer of size [8+4,256], a hidden layer of size [256, 256] and output layer of size [256,1]. Relu activation function is applied on the outputs of the first and second layers (b) Actor network: The common two linear layers are of sizes [8,256] and [256,256], and Relu activation function is applied on both their outputs. Then, the mean and log std layers (two separate layers) are of the same size [256,4]. Furthermore, the same initialization as described in DDPG is done to the third layer of the soft critics and targets, and to the mean and log std layers of the actor. Other hyperparameters are listed in table 3.

DDPG Hyperparameters	Values
Batch size	128
Buffer (memory) size	10000000
gamma	0.9829
tau	0.0035
Learning rate of actor	0.0006
Learning rate of critic	0.0014
Maximum number of steps	900
Epsilon decay	0.0000024
Epsilon	1.0233
Learning period	30
Update factor	30
Weight decay	0

Table 2: DDPG hyperparameters

SAC Hyperparameters	Values
Batch size	128
Buffer (memory) size	1000000
gamma	0.9853
tau	0.0882
alpha	0.2258
Learning rate of actor	0.0027
Learning rate of critic	0.0036
Learning rate of α	0.0020
Maximum number of steps	1000
Delay Frequency	5

Table 3: SAC hyperparameters

3.2 Experiments

Three experiments have been performed:

Train. Train the agent to solve the environment, i.e. to get an average reward of 200 over 100 consecutive trials, and do it in the minimum number of episodes. In our experiments we have used DQN, Full-DQN, Double-DQN, Dueling-DQN, SAC and DDPG, all implemented as previously explained and with the above fine-tuned hyper-parameters. The training process is described in section 2.3.8. The final plots are presented in figure 1. Table 4 shows the results- how many episodes and time in minutes it took each agent to solve the environment. From the figures and table it can be seen that in the aspect of number of episodes, The SAC agent solved the environment the fastest, significantly faster than the other, within only 118 episodes. DQN, Full-DQN and Double-DQN had approximately similar performance considering number of episodes and running time. Dueling-DDQN was the best of the DQNs with only 146 episodes. It seems that except for the Dueling-DDQN, the advanced versions of DQN has not done a significant change. Last, DDPG solved the environment within 200 episodes, the longest of all. In conclusion, the SAC agent performed best in the training mode.

Test. Given the best model of the actor, test the agent for 100 episodes to check success rate. This is a very important experiment because by it we can check whether the agent really learned something in the training mode, and if with the optimal weights and bias of the actor network it can succeed solving the environment. In practice, for each episode, for each step, we choose the greedy action and perform a step in the environment. The reward is added to the final score

of the episode. If done before maximum number of steps, we continue to the next episode. For each episode the final score is saved, and we plot a graph of the scores as a function of episodes. The goal is to get a score bigger than 200 for as many episodes as possible. The success rate (in percents) is computed as the number of succeeded episodes divided by 100. Figure 2 and table 4 show the results. Moreover, we give links to videos showing the agent's performance in 10 consecutive episodes when in test mode (see section 5). It can be seen that the SAC agent not only performed best in training mode and solve the environment in the smallest number of episodes, but also has the largest success rate- 96%! This means the agent did learn in the training mode and there was not overfitting. However, despite solving the problem, the DQN, Full-DQN and Double-DQN have a very small success rate. The Dueling-DDQN on the other hand has a large success rate- 80%. Last, although the DDPG agent solved the environment in the largest number of episodes and longest time, it has almost the biggest success rate (84%). In conclusion, the SAC agent performed best in the testing mode.

Uncertainty: Uncertainty is the addition of Gaussian noise to the agent's position with parameters $\mu = 0, \sigma = 0.05$. This addition was added only to the location of the lander (i.e. PositionX and PositionY). We measured the effect of this addition by applying it on each of our models using the hyper-parameters that achieved best results. Figure 3 depicts the training of the models with the uncertainty addition (i.e solving the environment). Figure 4 depicts the testing, which means the ability of the models to infer after solving the environment. It is noticeable that the addition caused the DQNs to perform poorly, while the DDPG shows a significant improvement and the SAC doesn't seems to be either significantly improve or worsen from the addition (a success rate of 95% compared to 96% without the addition of the noise). Table 5 summarizes the Number of episodes the training took, running time of the training and success rate of the testing.

	DQN	Full-DQN	Double-DQN	Dueling DDQN	SAC	DDPG
Number of episodes	195	191	181	146	118	200
Running time (minutes)	4.23	4.18	4.13	4.28	11.73	15.83
Success rate (%)	24	61	48	80	96	84

Table 4: Training and testing results of the agents: Number of episodes for solving the environment in training mode, running time in minutes and success rate in test mode

	DQN	Full-DQN	Double-DQN	Dueling DDQN	SAC	DDPG
Number of episodes	231	176	355	178	135	167
Running time (minutes)	6.42	4.00	5.38	5.93	12.13	20.25
Success rate (%)	0	32	7	71	95	88

Table 5: Training and testing results of the agents with uncertainty: Number of episodes for solving the environment in training mode, running time in minutes and success rate in test mode

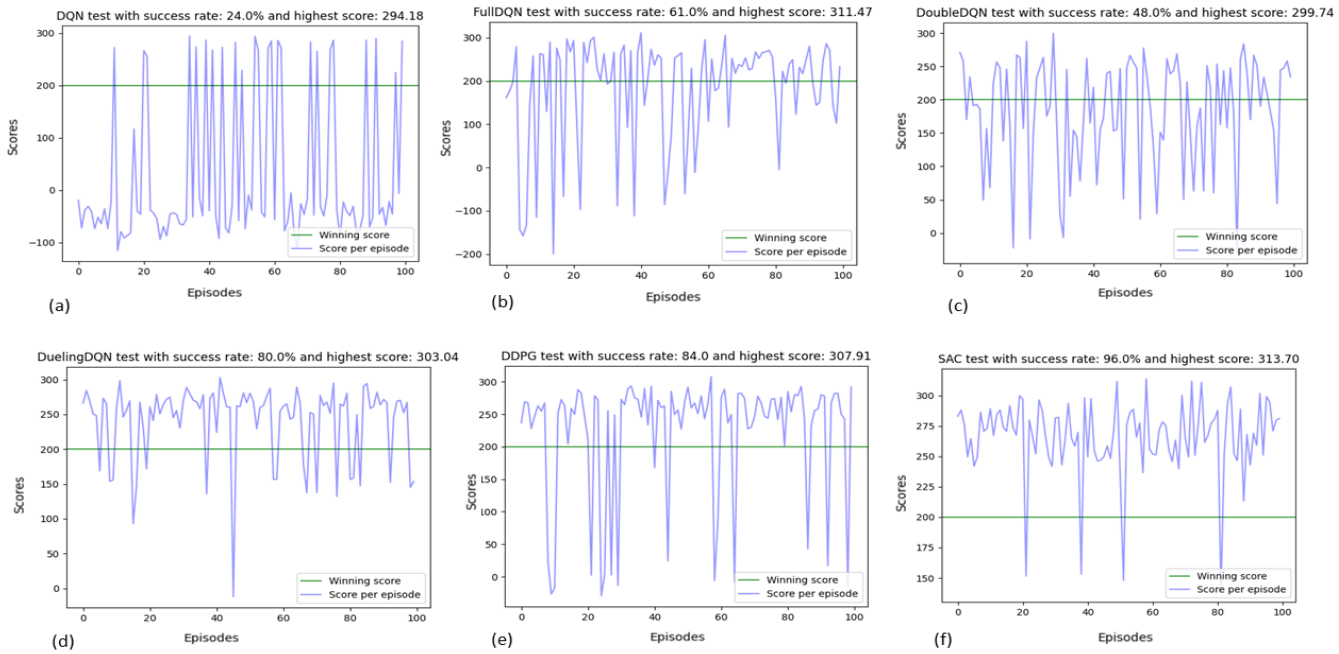

Figure 2: Testing. (a) DQN (b) Full-DQN (c) Double-DQN (d) Dueling-DDQN (e) DDPG (f) SAC. The green line is the winning score (200). The blue line is the reward per episode.

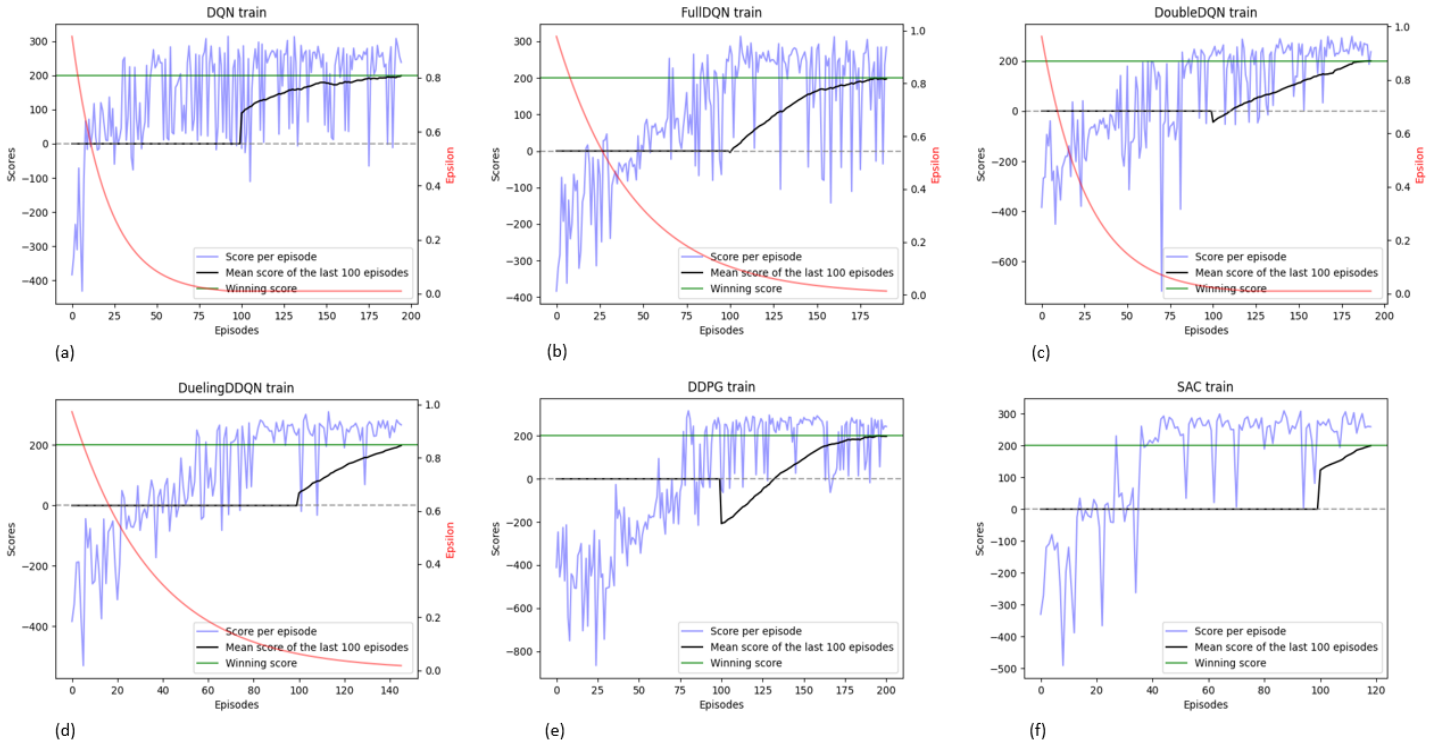

Figure 1: Training. (a) DQN (b) Full-DQN (c) Double-DQN (d) Dueling-DDQN (e) DDPG (f) SAC. The green line is the winning score (200), The black line is the average reward per 100 consecutive episodes (0 if less than 100 episodes are done), The blue line is the reward per episode. For DQNs only, the red line is the value of epsilon at each episode.

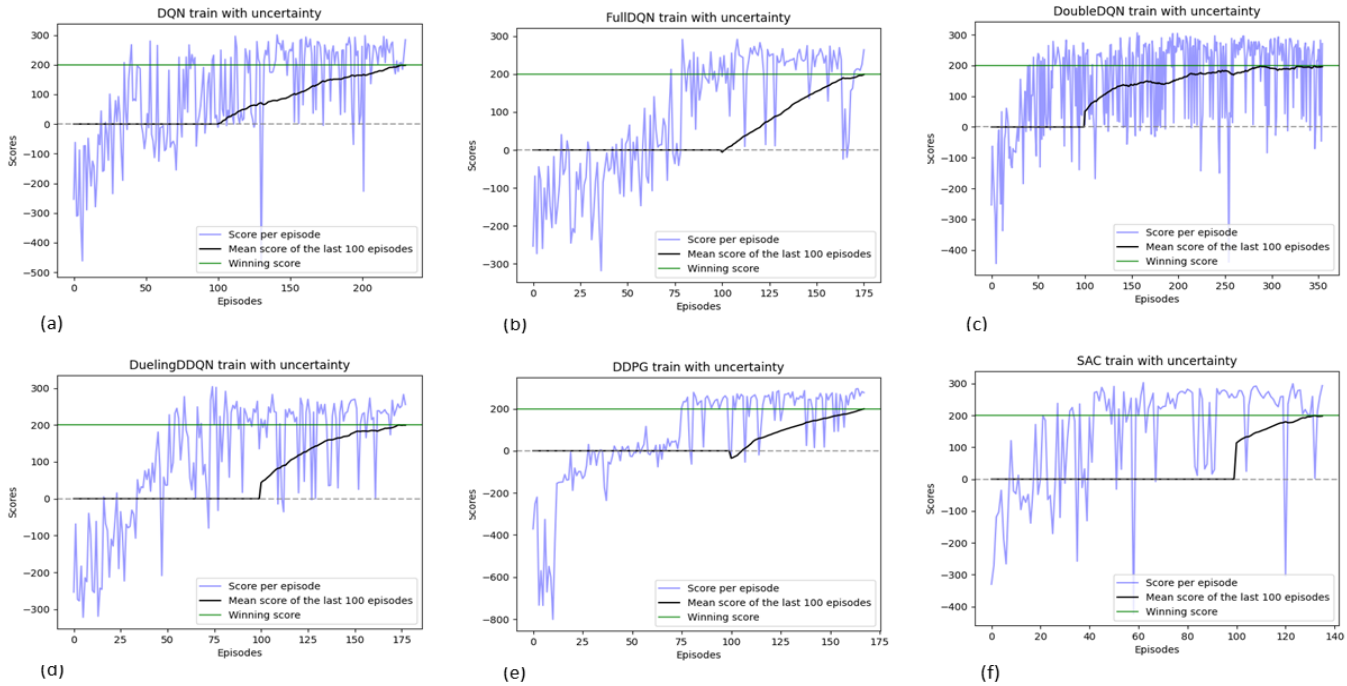

Figure 3: Training with Uncertainty. (a) DQN (b) Full-DQN (c) Double-DQN (d) Dueling-DDQN (e) DDPG (f) SAC

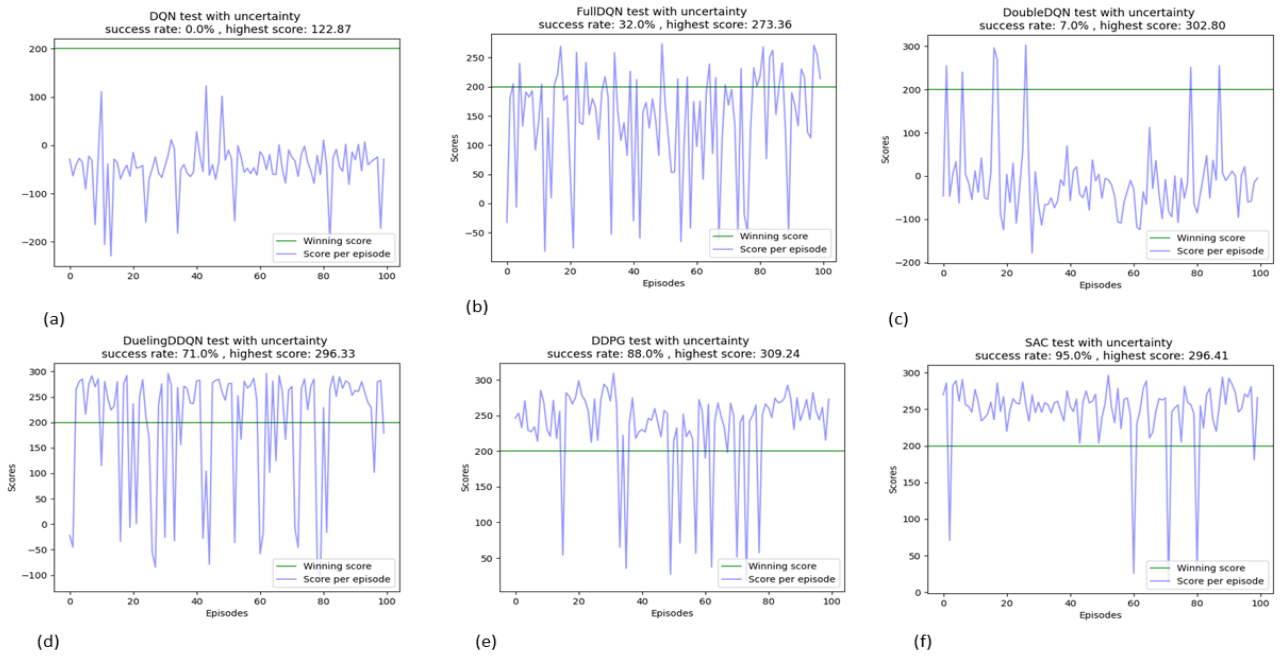

Figure 4: Testing with Uncertainty. (a) DQN (b) Full-DQN (c) Double-DQN (d) Dueling-DDQN (e) DDPG (f) SAC

4 Discussion

In this project we have learned about different models (agents) and their ability to infer and learn after achieving the goal of winning an average score of 200 over 100 consecutive trials (solving the LunarLanderContinuous-v2 environment). At the beginning, we have decided to use different Model-Free methods from each category: Deep Q-Learning (e.g. DQN and its variants), Policy Optimization (such as PPO) and algorithms that combine both worlds (SAC and DDPG). Most of our implementations goals were achieved except for the PPO model- We implemented it but the model wouldn't converge, even after a strong hyperparameter fine-tuning. Hence, at the end, we have used the following methods: DQN and its variants, SAC and DDPG. Our initial assumption was that the SAC and DDPG algorithms will yield better performance (because they combine the advantages of both approaches) and that the more advanced versions of the DQN will perform better than the basic one (but not as good as the continuous action space models). Indeed, SAC performed best: It has solved the environment in the smallest number of episodes, its success rate was the highest, adding uncertainty to the position of the agent did not seem to affect it, and it was very stable and insensitive to hyperparameters. DDPG agent may be the "slowest" agent of them all, but had a very good success rate. Also, surprisingly, adding uncertainty not only that did not have a bad effect on it, but the opposite- it took the DDPG agent even less episodes to solve the environment and its success rate increased. However, the bad thing about DDPG is that it was very sensitive to hyperparameters, and we needed to carefully fine-tune them (with NNI tool). On the other hand, the DQNs, except of the Dueling-DDQN, have surprised us for the worse. First, it seems that the advanced versions did not help very much- DQN, Full-DQN and DDQN converged almost in the same number of episodes and same running time. Second, they may converge fast, but their success rate was very low and not satisfying. Moreover, they were not stable, very sensitive to hyperparameters and strong fine-tuning was required (with NNI tool). As opposed to them, the Dueling-DQN achieved nice results, both on train and test. It is important to mention that the classic implementations of the DQNs use exponential ϵ decay, however when using it, our agent did not converge at all. Thus, we have looked for other decay options until we found the one that with it, the models did converge. About the uncertainty, The DQNs has worsen from the addition, it requires a larger number of episodes to converge and their success rate decreased. We would like to mention that by using the NNI tool the chosen hyperparameters are those which achieved convergence with the minimum number of episodes. Perhaps tuning with a different target function such as the loss function of the actor or allowing the agent to continue to learn even after solving the task may lead to a better inference performance. From this we conclude that adding uncertainty may actually be a good thing, making the agent to generalize better, hence learn better. but it in our case we did not see a significant evidence for that.

In conclusion, by working on this project we have been exposed to different reinforcement learning models. The project was very insightful, enlightening, and an enjoying process of learning and experimenting. From our experiments and results, we conclude that the SAC agent was best for our task in all aspects described above. Furthermore, we believe that the more stable and insensitive to hyperparameters a model is, the more better performance it will have.

5 Code

The code is available at a privet git repository which is available here
In addition, we created a short 10 episodes test video for each model, which is available here

References

[1] Q. Fettes, "Lunarlandercontinuous-v2 environment," 2021. [Online]. Available: https://gym.openai.com/envs/LunarLanderContinuous-v2/

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Human-level control through deep reinforcement learning.*, 2015.

[4] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[5] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[6] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," *CoRR*, vol. abs/1812.05905, 2018. [Online]. Available: http://arxiv.org/abs/1812.05905

[7] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2016.

[8] microsoft, "Neural network intelligence," 2017. [Online]. Available: https://github.com/Microsoft/nni

[9] Y. Tang and S. Agrawal, "Discretizing continuous action space for on-policy optimization," *CoRR*, vol. abs/1901.10500, 2019. [Online]. Available: http://arxiv.org/abs/1901.10500

[10] P. Barekatain, "Understanding stabilising experience replay for deep multi-agent reinforcement learning," 2019. [Online]. Available: https://medium.com/@parnianbrk/understanding-stabilising-experience-replay-for-deep-multi-agent-reinforcement-learning-84b4c04886b5

[11] S. Karagiannakos, "The idea behind actor-critics and how a2c and a3c improve them," 2018. [Online]. Available: https://theaisummer.com/Actor_critics/

[12] Y. R. Tang, "onpolicybaselines," 2020. [Online]. Available: https://github.com/robintyh1/onpolicybaselines/blob/master/onpolicyalgos/discrete_actions_space/trpo_discrete/wrapper.py#L17

[13] pytorch, "Criterion function," 2021. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html

[14] Q. Fettes, "Deeprl-tutorials," 2019. [Online]. Available: https://github.com/qfettes/DeepRL-Tutorials

[15] T. Simonini, "Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed...," 2018. [Online]. Available: https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/

[16] V. V.Kumar, "Soft actor-critic demystified," 2019. [Online]. Available: https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665

[17] M. Tylor, "ddpg-pytorch," 2019. [Online]. Available: https://github.com/MoritzTaylor/ddpg-pytorch/blob/master/utils/nets.py

[18] C. Yoon, "Deep deterministic policy gradients explained," 2019. [Online]. Available: https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

[19] M. Buchholz, "Deep reinforcement learning. deep deterministic policy gradient (ddpg) algorithm," 2019. [Online]. Available: https://medium.com/@markus.x.buchholz/deep-reinforcement-learning-deep-deterministic-policy-gradient-ddpg-algoritm-5a823da91b43