Machine Learning Engineer Nanodegree

Capstone Project

Jean Ricardo Rusczak

April 17th, 2019

I. Definitions

Reinforcement Learning

Reinforcement learning is a subset of Machine Learning where an agent learns how to operate in an environment by trial and error. The agent is rewarded or punished accordingly its actions, and the main objective is to find an optimal policy that allows the agent to decide how to act and accumulate the most amount of rewards.

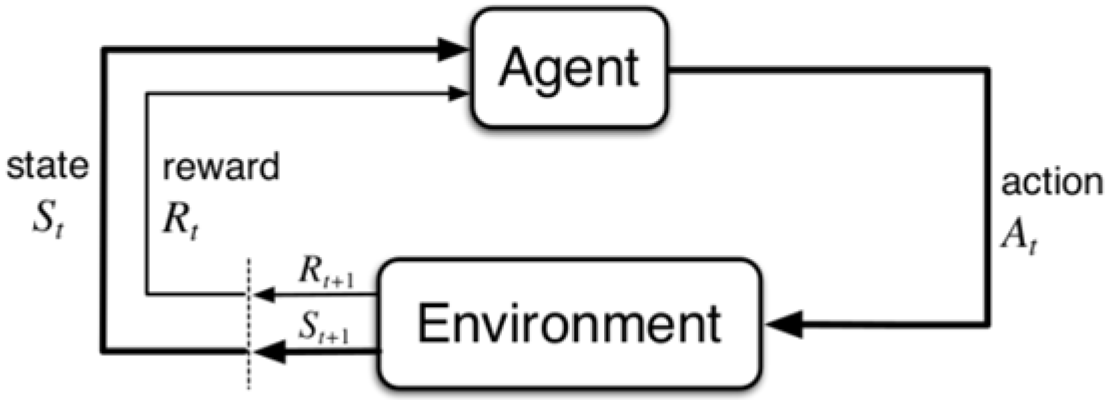


Figure 1 - Reinforcement Learning Problem (https://i.stack.imgur.com/eoeSq.png)

Some definitions that should be defined for this type of control are:

- Action (A): it’s an array of possible actions (i.e. Duty cycle, engine power, applied voltage, direction, angle, etc.)
- State (S): an array with all the identified states of the system being controlled (i.e., position, velocity, etc.)
- Reward (R): environment response from the applied action.
- Policy: it’s a strategy used by the agent to define the next actions based on the current state.
- Value (V): the expected long-term return considering the discount ()
- Q-Value (Q): the expected long-term return of the current state s, for an action a, under policy Pi.

Model-free v.s. Model-based

In model-based reinforcement learning, the agent has access to a model of the environment. In this case, it can compare its performance with a ground-truth performance. The model will give the state transitions and rewards can have.

With a known environment, the states can be divided into a grid and be slowly explored by the agent. Once it's explored, the agent will always find the optimal route that solves the environment. The efficiency of this method is due to the ability to plan all the agent actions after the initial state is known.

Unfortunately, the ground-truth model of the environment is usually not available to the agent. In this scenario, the agent has to acquire a set of experiences and use them to respond to stimuli from the environment.

Model-free algorithms have to explore the environment, as they do not have any knowledge about what works and what does not work in the environment.

Off-Policy versus On-Policy

In on-policy algorithms, the agent learns the Value based on a current action A that was generated based on the current policy. Therefore, the agent will only be able to learn a new value when the policy gets updated.

Whereas in off-policy algorithms, the agent learns the value based on an action A that can be obtained from another policy. A key benefit of this separation is that the behavior policy can operate by sampling all actions, whereas the estimation policy can be deterministic.

Observation Space

A state s is a complete description of the state of the environment. Whereas an observation is just a partial description of that state. And they can be discrete or continuous.

In discrete spaces, the environment will be divided into a grid that will be explored in order to find the grid address that will end the episode. And for continuous spaces there isn’t such division, so the agent has to start to learn an action-state pair that gives him the best return in every situation.

Action Spaces

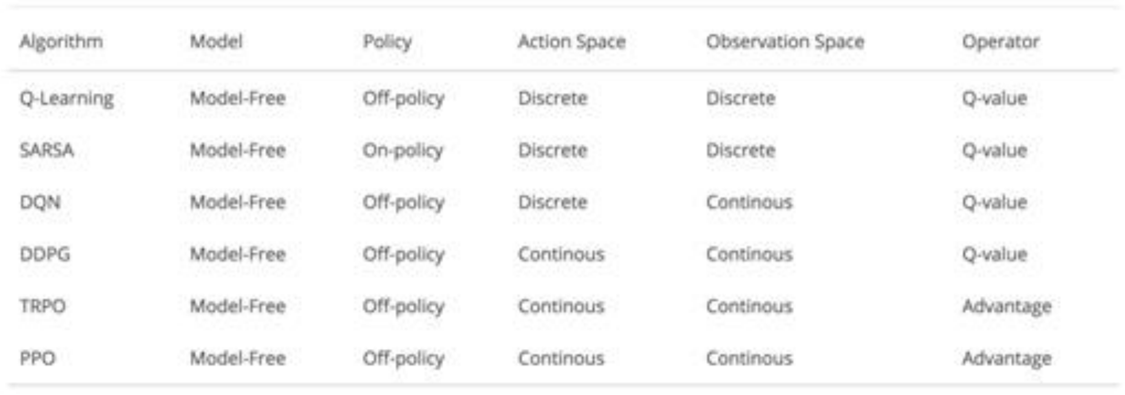
The set of valid actions for a given environment are called action space. As the observation space, it can be discrete or continuous.

Discrete action spaces have a finite number of actions the agent can execute, for example, (left, right, up, down). This type of action space is useful when the agent is not required to have great movement precision. A good example is video games, where the agent should follow the same set of commands a physical controller would have.

For other environments, especially the ones that describe the physical world, all actions must be continuous. This type of action is common in robots, where a PWM is applied to the motors so the robot can smoothly move around the space. So for continuous actions the number of possibilities is extremely large or infinite.

Algorithm Types

Depending on the type of observation and action spaces required for that particular environment, a different type of algorithm should be used:



Algorithm	Model	Policy	Action Space	Observation Space	Operator
Q-Learning	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA	Model-Free	On-policy	Discrete	Discrete	Q-value
DQN	Model-Free	Off-policy	Discrete	Continous	Q-value
DDPG	Model-Free	Off-policy	Continous	Continous	Q-value
TRPO	Model-Free	Off-policy	Continous	Continous	Advantage
PPO	Model-Free	Off-policy	Continous	Continous	Advantage

Figure 2 - Main Reinforcement Learning Algorithms

Source: https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-part-ii-trpo-ppo-87f2c5919bb9

As the proposed environment has a continuous observation space with continuous actions, three main algorithms could control the lander:

- Deep Deterministic Policy Gradients (DDPG)
- Trust Region Policy Optimization (TRPO)
- Proximal Policy Optimization (PPO)

In this project, at least DDPG will be investigated and tuned to perform the landing task based on the study proposed by (Lillicrap et al, 2015).

Project Overview

The ability to have a controlled descent in any surface has been for a long time object of study in the aerospace industry. For manned missions, it is possible to rely on the skills of an experienced pilot, but for remotely control, led missions the vehicle has to be able to land by itself. This is mostly because of the communication delay between Earth and the spacecraft, which can be from several minutes to hours, depending on where the spacecraft will land.

In recent years, the company SpaceX proved it is possible to improve the rocket reusability by performing a soft landing by using advanced adaptive algorithms (Açıkmeşe, 2012). The rocket family Falcon 9 has been able to reuse its boosters due to 33 landings of 40 attempts.

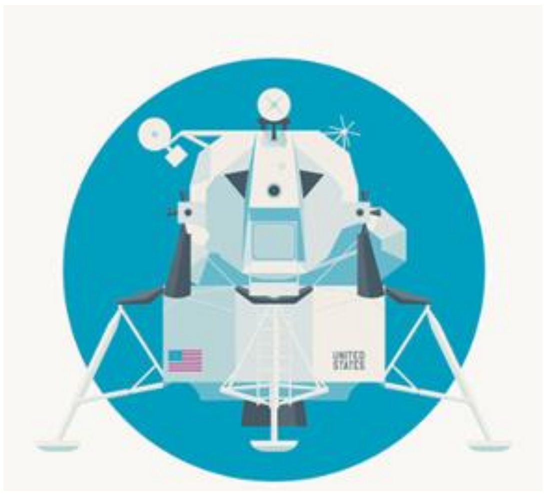


Figure 3 - Apollo 11 Lunar Lander Artwork

Source: https://fineartamerica.com/featured/apollo-lunar-module-lander-minimal-text-cyan-ivan-krpan.html?product=art-print

Other important milestones are the spacecraft Philae that managed to land on comet 67P/Churyumov–Gerasimenko in 2014, and the rovers and probes sent to Mars. A good example of powered descent was the InSight lander, which had a smooth and precise landing on Mars on Nov 2018.

Problem Statement

The main objective of the algorithm will be landing a simulated lunar module that is generated by the OpenAI Gym environment.

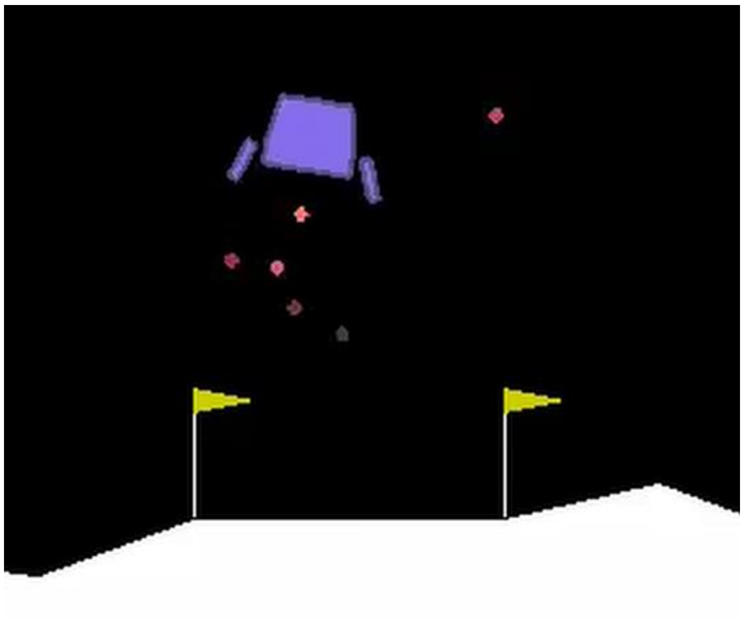


Figure 4 - Lunar Lander in Gym Environment

This simulated lunar lander has to land on the landing pad that is defined by the location between two flags. The landing pad has fixed coordinates (0,0) and the lander can start at random positions.

The lander continuous state is described by its horizontal position (x-axis), vertical position (y-axis), horizontal linear velocity (Vx), vertical linear velocity (Vy), orientation (θ), angular velocity (Vθ), state of each landing leg (left and right).

To control the lander it is required 3 engines: a main and two directional, which can be activated to move the rocket to left and right. The main engine control can accept numbers from -1.0 to +1.0, but it will only fire when power is equal or greater than 50% (+0.5 to +1.0). And for the directional engine control, it can accept values from -1.0 to +1.0, being (-1.0 to -0.5) to fire the left engine and (+0.5 to +1.0) to fire the right engine. Values between (-0.5 to + 0.5) the engine will be in OFF state. Given that the state space is continuous, it’s considered infinitely large.

The lander is rewarded when it performs specific tasks like moving from the top of the screen to the landing pad and landing with zero speed. For this case it can be rewarded from 100 to 140 points.

When the episode finishes the lander will receive -100 or +100 points if it crashes or comes to rest, respectively. For each leg that made contact with the ground, the lander receives +10 points.

In this simulation, fuel is infinite, but the lander is penalized in 0.3 points each time (step) the main engine is ON.

Finally, the lunar lander can land anywhere, even outside the landing pad but it will not receive the maximum reward (+200 points).

Metrics

One of the main metrics this problem can use is the amount of rewards the agent can receive per episode. As this value usually can vary a lot, a moving average was applied in order to best track if the algorithm is learning or not.

II. Analysis

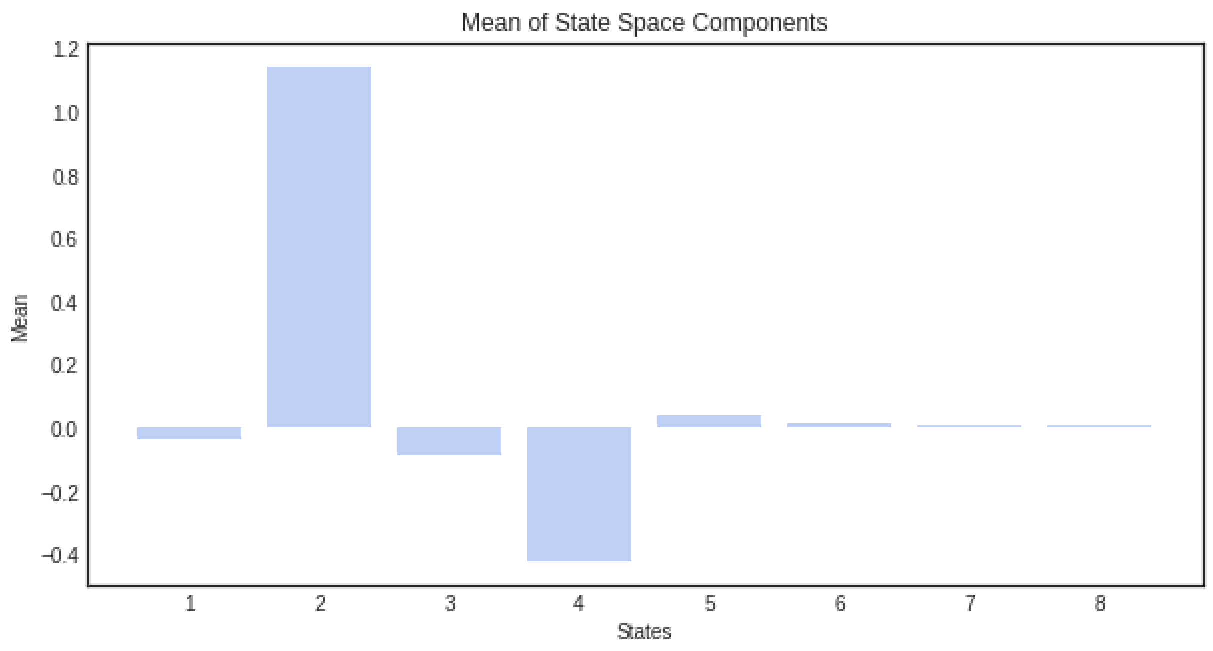
Data Exploration

There is no previous dataset and the agent will learn based on interactions with the environment. So, the first approach is to understand how much variation the networks should expect in their inputs by creating a random agent that will apply random actions to the environment.

The idea is to understand if there is any outstanding state space component and if the input requires any sort of normalization.

Exploratory Visualization

The following charts show the mean and standard deviation of the 8 state space components with random actions being applied to the environment over 1000 episodes and 100 steps per episode.



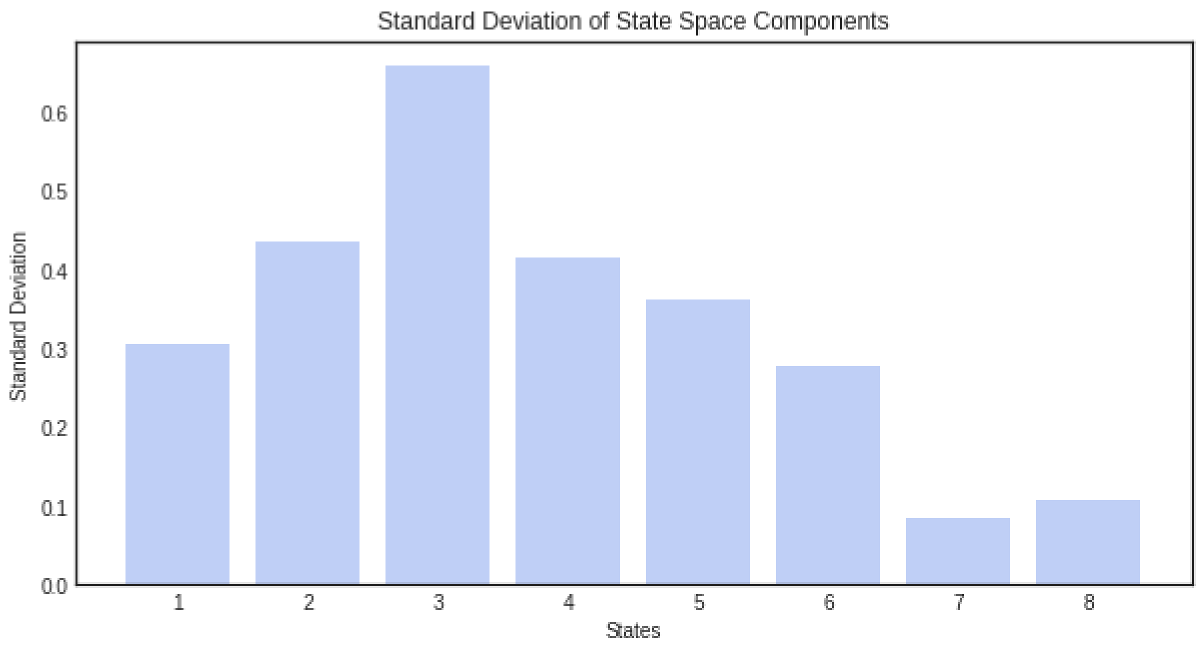


Figure 5 - Mean and Standard Deviation of the state space components under random actions

Based on these results, it could be interesting to apply a normalization factor to the inputs of the policy network as the second and fourth state space components (Y-axis and Vy) can be larger than the other components when sampling from the replay buffer.

Algorithms and Techniques

Deep Deterministic Policy Gradient (DDPG)

The base of the Deep Deterministic Policy Gradient (DDPG is a learning algorithm architecture called Actor-Critic, which can decouple the policy function from the value function (Q-Function). In this architecture, the policy function structure is known as the actor, and it will generate the actions to be applied to the environment during that current step. And the critic, will observe these actions and the environment state to estimate a Q value.

DDPG is an off-policy, model free algorithm that works for continuous action spaces that uses a stochastic behavior policy to exploration, but estimates a deterministic target policy. It introduces a few new concepts:

- Replay Buffers
- Target Networks
- Ornstein–Uhlenbeck applied to the policy

Replay Buffers

Reinforcement learning algorithms have to explore their environment in order to understand what is the best set of actions for different states of the system. And the agent will have different experiences during its training and might happen with different levels of sparsity. A set of previous and current experiences is called replay buffer.

It should be large enough to contain a wide range of experiences, but cannot have too much otherwise it will slow down the training and limit the learning performance as well.

As the Bellman equation doesn't evaluate which transition tuples are used or the actions being selected, the experiences can be randomly sampled from the buffer, even that the experiences was obtained with an outdated policy.

Target Networks

Target networks come to try to solve the condition where the Q-function to target error is being minimized but they depend on the same parameters that are being trained making the MBSE minimization unstable.

They are a time-delayed copy of the original network (actor and critic) and they are updated at the same time of the main network by polyak averaging (running average).

Action Noise

As the DDPG policy is deterministic but trained in an off-policy way, so the agent has to try to explore the environment in order to learn from it. As per the original DDPG paper, the authors recommend a time-correlated OU noise to be added to the controller actions.

By doing this, the agent can get different experiences and try to learn from it. And the amount of noise has to be tuned depending on the type of environment the agent is exploring. In this implementation, there is another factor Epsilon that slowing decreases the amount of noise as the agent becomes more experienced. And the balance of how much the action must vary versus how much the agent is actually learning from the experiences is really important during training.

Combining all blocks

The DDPG architecture can be described as the image below:

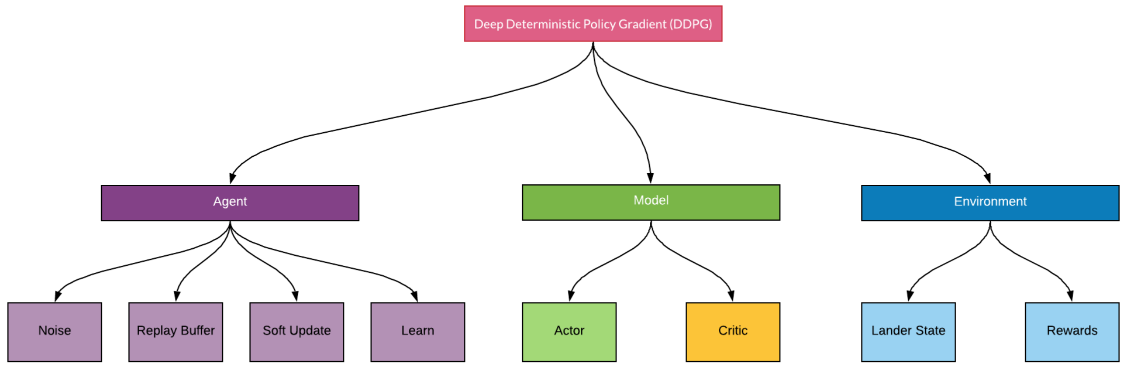


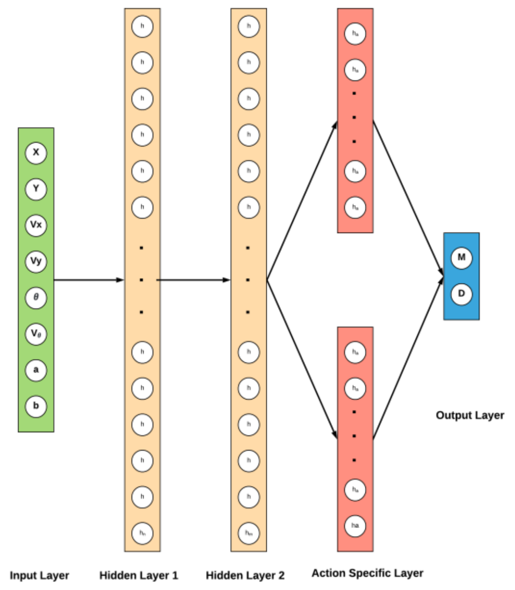
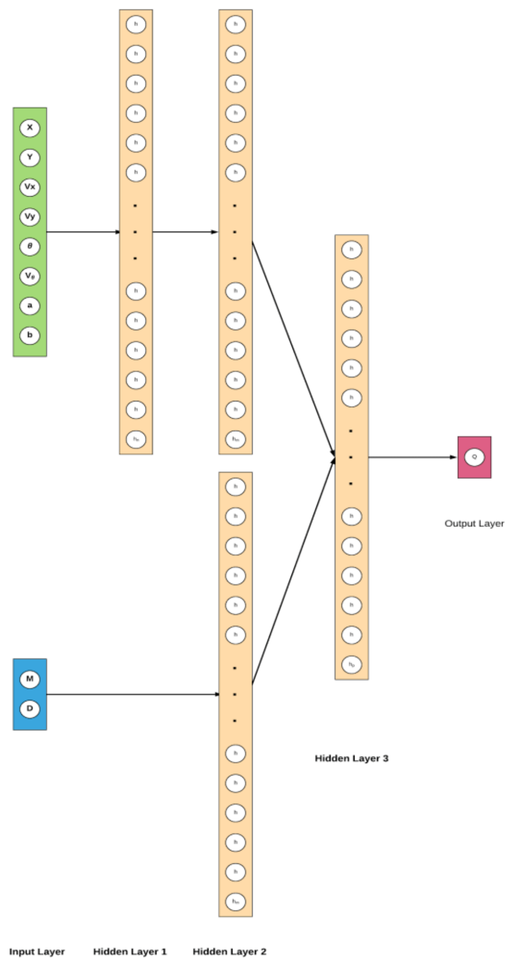
Figure 6 - DDPG Main Architecture

And the complete algorithm that implements the behavior of this architecture as proposed by (Lillicrap et al., 2015) will be as follows,

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
**for** episode = 1, M **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    **for** t = 1, T **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$
        Update the target networks:
            $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
            $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
    **end for**
**end for**
```

As the actions have different requirements, so for the actor network they were split into two final layers, after the basic structure proposed by the original paper. The idea behind this approach is to use a customized activation function and network complexity for each output.

The Actor Network	The Critic Network
 X Y Vx Vy θ Vθ a b M D Output Layer Input Layer Hidden Layer 1 Hidden Layer 2 Action Specific Layer	 X Y Vx Vy θ Vθ a b M D Output Layer Hidden Layer 3 Input Layer Hidden Layer 1 Hidden Layer 2

Benchmark

For the specific environment there are only a few benchmarks available (OpenAi Gym, 2019), which gives the best result a score greater than 200 for 100 trials. The author uses the Proximal Policy Optimization algorithm to achieve that.

Mostly the documents available solve this environment with discrete actions, so the benchmark for this project will be a random agent trying to control the spacecraft.

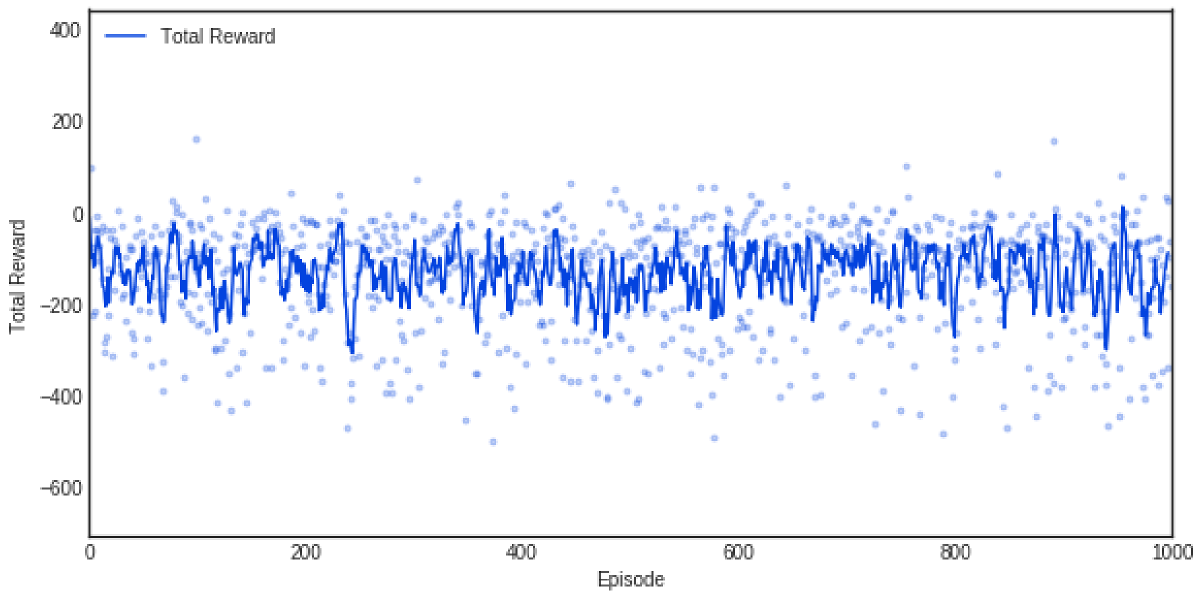


Figure 7 - Lunar lander rewards with random actions

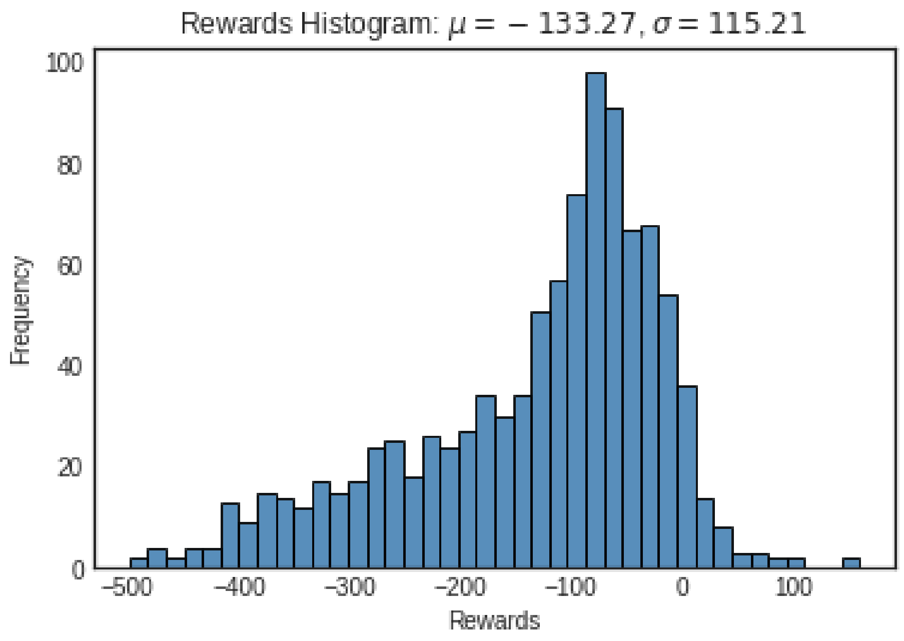


Figure 8 - Lunar lander continuous rewards histogram with random actions

As can be observed from the charts above, the agent just applies random actions and fails to land every time, getting really low scores. The rewards histogram has a normal distribution shape and there are only two scores in 1000 episodes near 200 - but still below this mark.

III. Methodology

Reproducibility

One of the main challenges of RL is how to reproduce results. Usually, papers give a superficial description of the algorithm and do not detail all hyperparameters and randomizations that the writers used while studying that particular task.

The stochasticity can occur both in the environment and in the learning process (i.e. random weight initialization, sampling experiences from a buffer, etc). Accordingly to (Henderson et al. 2019), for some environments it is possible to get learning curves that do not fall within the distribution at all.

In order to reduce this effect, the vanilla DDPG proposed by Udacity had to be modified to have a common random seed. The same seed was applied to the Uhlenbeck & Ornstein noise, replay buffer sampling, both actor and critic random initializers.

Data Preprocessing

Weight Initialization

The actor neural network used the rectified non-linearity (Glorot et al., 2011) for all hidden layers. The critic neural network hidden layers were randomly initialized using a uniform distribution between [-0.003,0.003].

Implementation

The algorithm was implemented in Keras with a main training function implemented on Jupyter notebook.

The episode can finish if lander starts to tilt too much that it probably won't recover, the idea is to have a smooth descent to the target.

A random uniform initialization was also applied to the lander in order to prefill the replay buffer with samples that are different than just zeros.

Refinement

The vanilla DDPG structure was modified to include action specific layers that help the network to have a better fit in actions that have a different range.

In this case, the actions for the main engine and the directional thrusters operate from 0.5 to 1.0 for the first and (-1.0 to -0.5, 0, +0.5 to + 1.0) for the second. So, being able to explore different activation functions is really interesting to limit the amount of useless variation each output can generate.

The exploration noise parameters sigma and theta also were modified to enable the network to explore more. And to control the noise reduction when the network is becoming stable, a decay factor (epsilon) is being applied to each noise step and slowly reducing the amount of action noise.

Algorithm	Hyperparameter	Value
DDPG Agent	Discount (Gamma)	0.995
	Soft-update (Tau)	0.0001
	Batch Size	128
	Action Repeat	3

Algorithm	Hyperparameter	Value
Model - Actor	Learning Rate	1e-4
	Hidden Layer 01 Dimension	400/ReLU
	Hidden Layer 02 Dimension	300/ReLU
	Action Specific - Engine	1/Sigmoid
	Action Specific - Direction	8/Tanh

Algorithm	Hyperparameter	Value/Dimension/Activation
Model - Critic	Learning Rate	1e-3
	Hidden Layer 01 Dimension	400/ReLU
	Hidden Layer 02 Dimension (action/state)	300/ReLU
	Hidden Layer 03 Dimension (merged)	300/ReLU

	Q Layer	1/Linear

Algorithm	Hyperparameter	Value
Replay Buffer	Buffer Size	1e4

Algorithm	Hyperparameter	Value
Ornstein-Uhlenbeck Noise	□_main engine	0
	□_directional_engine	0
	Theta	0.4
	Sigma	0.3
	Epsilon	1.0
	Epsilon Detay	1e-5

IV. Results

The algorithm was trained for different number of episodes depending of the network structure that was used.

For a replay buffer with size around 10000, the number of episodes was set to 750, as the algorithm started to overwrite previous experiences by episode 200. And for replay buffer size 100000, the number of episodes was set to 1500 to 3000, depending of the network complexity.

The first approach was to use the vanilla DDPG with replay buffer and OU noise and the network was slowly tuned with small variants of the original DDPG.

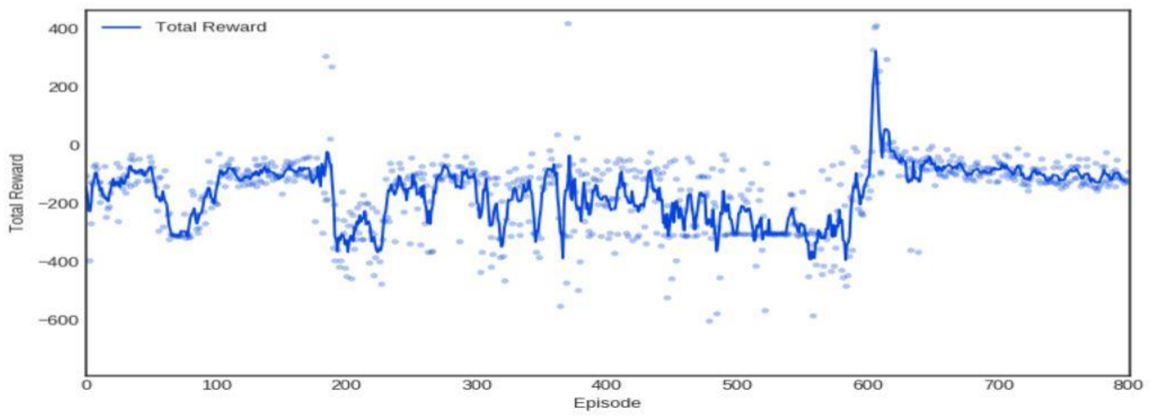


Figure 9 - Solved environment with 400x300 networks (Original DDPG)

Adjusting the replay buffer size and modifying the discount factor to 0.995, the soft-update ratio to 0.0001, and buffer size to 128 started giving more stable results.

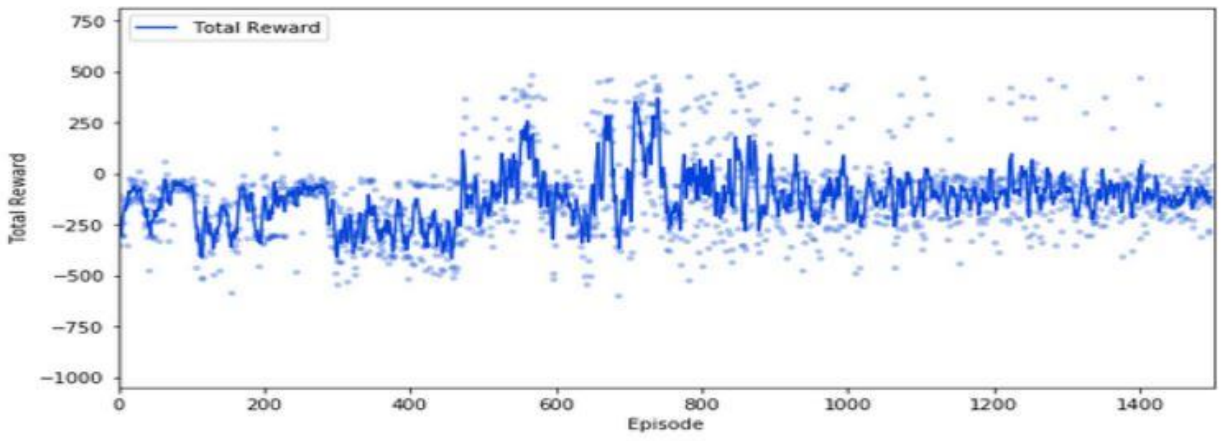


Figure 10 - Solved environment with 400x300 networks and action specific layers

With a final tweak of adding more nodes to the second hidden layer of the actor and second and third of the critic, made the algorithm increase its rewards average after 1200 episodes as shown below.

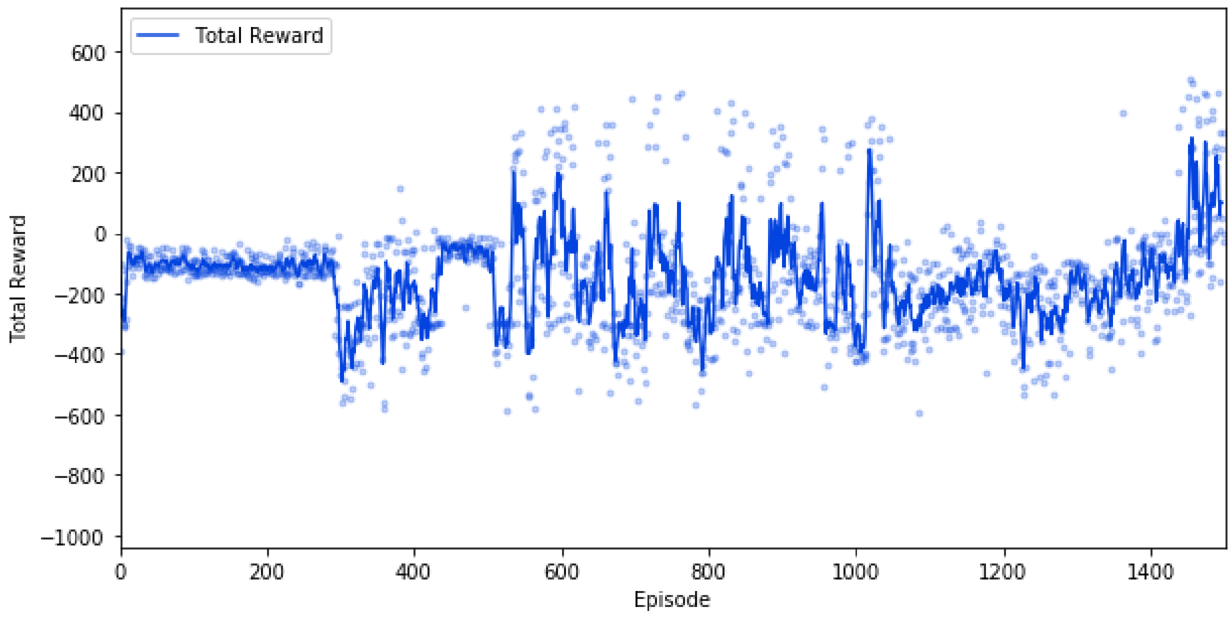


Figure 11 - Solved environment with 400x400 networks and action specific layers

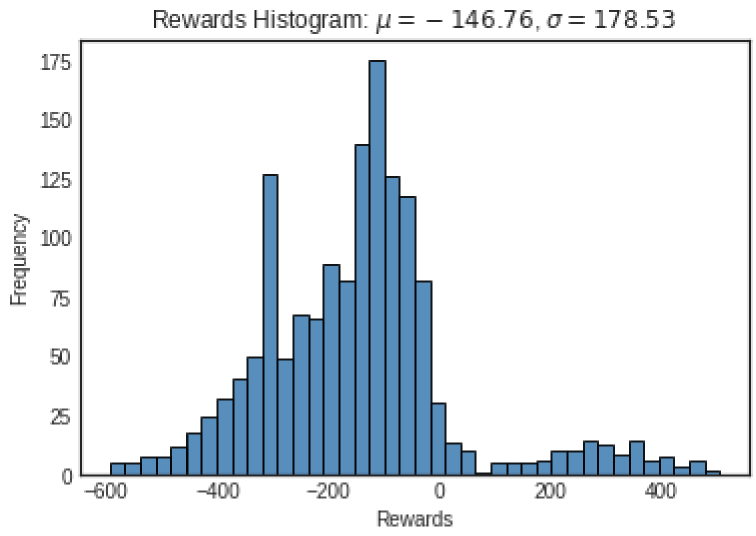


Figure 12 - Solved environment with 400x400 networks and action specific layers

In this scenario, the environment it is considered solved as the average reward become stable on +200 points.

Model Evaluation and Validation

The training process can be very time consuming and requires an early stopping when the agent starts to have a steady learning. By the observations, it usually happens a little bit after the replay buffer start being overwritten. Which is an interesting fact and helped guide to define the final replay buffer size.

It seems that this technique still requires a lot of work so the training can be performed faster as convergence can happen only after thousands of episodes and becomes hard to do a fine tuning hyperparameter comparison.

As it was observed in other problems during the course, problems with discrete action spaces tend to be easier to control and have a more stable learning process when compared to continuous action spaces.

Justification

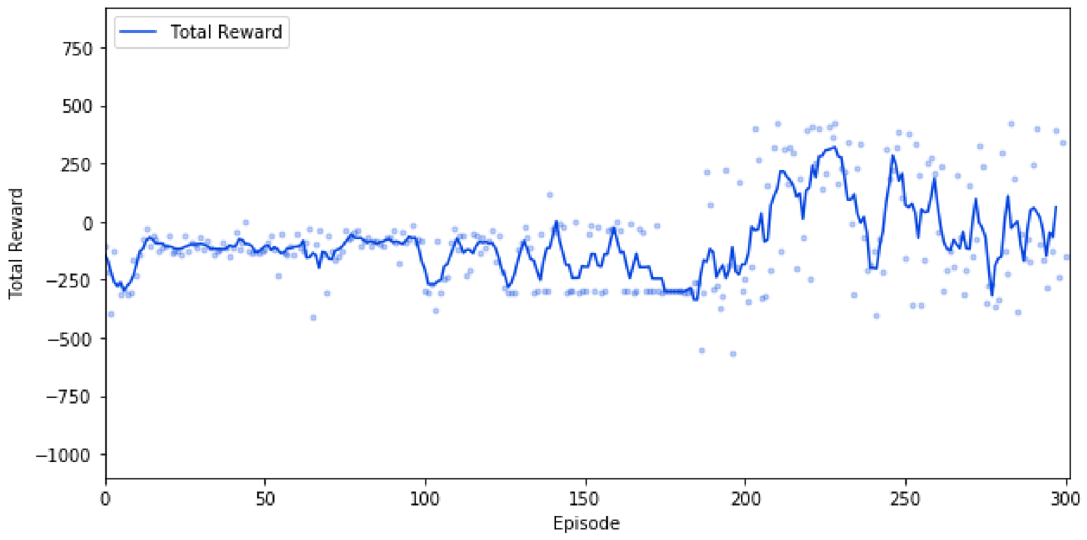
The proposed model can learn from the environment in most of the repetitions. In some trials, it can’t learn due the level of randomization that is being applied to the algorithm. This indicates that the noise has to be controlled a little bit better in order to make the learning process more stable.

The average rewards is greater than 200, which means that the environment is being consistent being solved. Even though, there are many points making making wrong decisions and failing to land.

V. Conclusion

Free-Form Visualization

By controlling and slowing tuning the DDPG hyperparameters it was able to start getting a stable learning around 200-250 episodes. For this solution the actor and critic networks were modified to have the first layer with 600 nodes and the second (and third for the critic) with 400 nodes.



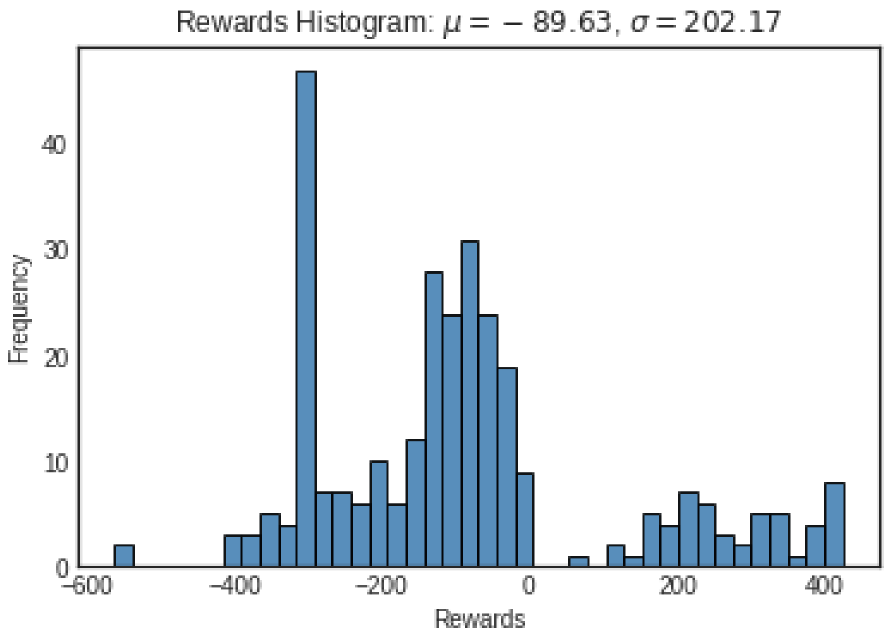


Figure 13 - Solved environment with 600x400 networks and action specific layers

This is still a not stable solution but indicates that the agent can be tuned to achieve a faster performance only by modifying the dimensions of the networks. The problem with this solution is the time for each learning iteration, as the networks grow, a lot of computing is required and the training and the final implementation start becoming slow and inefficient.

Reflection

Accordingly to papers and demonstrations of DDPG in several continuous action environments, it can have an average performance and solve the environment in +1500 episodes. DDPG can also easily overestimate the Q values and then it struggles to learn as it leads to a policy-breaking situation.

The DDPG algorithm was evaluated with several sets of hyperparameters but it still not completely robust to random seeds for this environment.

The concept of replay buffer also seems that doesn’t help much the training to converge, finding the right balance between the amount of good experiences and bad experiences that the agent has to use usually leads the agent to learn bad behaviors.

A few attempts were done in this sense, like splitting the buffer into 3 and organize the experiences in Positive, Neutral and Negative. The idea behind this is to reduce the amount of negative experiences that the agent will have during the beginning of the training and balance out the examples. This didn’t give good results as it’s hard to define what’s a good example and a bad example during the training as the rewards are only incremental, and sometimes a small reward in one step might lead to a big reward on the next step.

Controlling the action and critic networks dimension seems to have the same level of importance than buffer size, soft update factor or the discount gamma. This makes the algorithm extremely hard to tune and hard to manage changes during a real world implementation. If a new environment factor must be added, DDPG might have to be restructured and retuned, which is time consuming.

Improvement

A few modifications to the vanilla DDPG were evaluated but without great success as presented in the papers. As mentioned before, the algorithm is sensitive to hyperparameter variation and won't converge if the parameters are not precisely tuned.

A better technique for the replay buffer should be considered. A couple examples are the Prioritized Experience Replay and the Hindsight Experience Replay, which seems to speed up the learning convergence when compared to the vanilla DDPG.

References

Achiam, Joshua. (2019, Mar 14). Spinning Up Documentation - DDPG. Retrieved from: https://spinningup.openai.com/en/latest/index.html.

Açıkmeşe, Behçet & Casoliva, Jordi & Carson, John & Blackmore, Lars. (2012). G-FOLD: A Real-Time Implementable Fuel Optimal Large Divert Guidance Algorithm for Planetary Pinpoint Landing.

Doshi, Neerja. (2018, Mar 26). Deep Learning Best Practices (1)—Weight Initialization. Retrieved from: https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94.

Emami, Patrick. (2016, Aug 16). Deep Deterministic Policy Gradients in TensorFlow. Retrieved from: https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html.

Glorot, X., et al. (2011). Deep sparse rectifier networks. In Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume, volume 15, pp. 315–323.

Huang, Steeve. (2018, Jan 12). Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG). Retrieved from: https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287.

Islam, R., Henderson, P., Gomrokchi, M., Precup, D.. (2017). Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. Reproducibility in Machine Learning Workshop, ICML'17. arXiv:1708.04133.

Lillicrap, T. P., et. al. (2015) Continuous control with deep reinforcement learning. arXiv:1509.02971.

Maksutov, Rinat (2018, May 1). Deep study of a not very deep neural network. Part 2: Activation functions. Retrieved from: https://towardsdatascience.com/deep-study-of-a-not-very-deep-neural-network-part-2-activation-functions-fd9bd8d406fc.

OpenAi Gym Leaderboard. (April 2019). Retrieved from: https://github.com/openai/gym/wiki/Leaderboard#lunarlandercontinuous-v2

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized Experience Replay. In International Conference on Learning Representations (ICLR). arXiv:1511.05952.