On the effect of different function approximator network structures in Deep Deterministic Policy Gradient (DDPG) methods

Martijn Brummelhuis - 4442164
https://github.com/mbrummelhuis/bio-inspired
Keywords: *Reinforcement Learning, Deep RL, Deep Deterministic Policy Gradient (DDPG), Continuous action spaces.*

ABSTRACT

No definitive way of optimising deep neural network architectures exists yet, so for problems where such networks are used as function approximators, it is a matter of experimentation to find out the best network structure. This problem was investigated in the context of Reinforcement Learning in continuous action spaces by using a deep learning based approach called Deep Deterministic Policy Gradients. By training an agent's actor, critic and target networks on OpenAI Gym's 'LunarLanderContinuous-v2'-environment using varying neural network architectures, it was found that simple networks lack the necessary approximation power to model the ideal policy function, and complex networks require more training data than was allotted here (1000 episodes) in order to match performance with the models of medium complexity. Additionally, a hyperparameter study was conducted in which it was found that very small values for τ impair the actor and critic networks from learning and thus this has an adverse effect on performance, while larger values (tested for 0.1) and batch sizes (32, 64 and 128) do not meaningfully impact agent performance.

List of abbreviations

(D)NN	(Deep) Neural Network
AC	Actor-critic (methods)
DDPG	Deep Deterministic Policy Gradients
DQN	Deep Q-Network
i.i.d.	Independently and identically distributed
MDP	Markov Decision Process

List of symbols

α	Actor learning rate
β	Critic learning rate
γ	Discount rate
γ	Discount rate
μ	Actor network
π	Policy
τ	Target network update weight factor
θ	Network learnable parameters
a	Action
$Q^\pi(s, a)$	Expected discounted reward under policy π taking action a from state s
r	Reward
S	Action space
S	State space
s	State
$V^\pi(s)$	Expected discounted reward under policy π from state s

1 INTRODUCTION

In 2013, DeepMind published a paper [1] in which they demonstrated a software architecture that could play old Atari games with the same inputs a human gets when playing, namely the raw pixels of the screen. There was no prior about the game in the software but it somehow managed to achieve superhuman performance on some of the games. This same method was later used by DeepMind to beat human master level performance on the game of Go [2, 3]. The underlying technique that made this possible is called Deep Reinforcement Learning and is a confluence of Reinforcement Learning and Deep Neural Networks (DNNs).

Reinforcement Learning is the idea that an agent can perform actions in the environment, and this environment will then give a reward (feedback, either positive or negative) and a state (mathematical description of the current situation). The goal is to maximise the reward. The challenge is then to construct a policy function that takes in the current state and finds the right action that maximises the reward. As artificial neural networks (or deep neural networks) are function approximators, they can be used as policy function, and trained to approximate the optimal policy. This also yields the nice analogy that the neural network is the 'brain' of the agent, as it perceives the state and selects the next action.

With this analogy, it seems rather obvious to try and use neural networks as function approximators for the policy of a Reinforcement Learning agent. However, in practice, this implementation is not so straight-forward. Progress has successfully been made in this area, as demonstrated on the Atari games, but this proposal only works for discrete action spaces. In the real world, and especially control, continuous

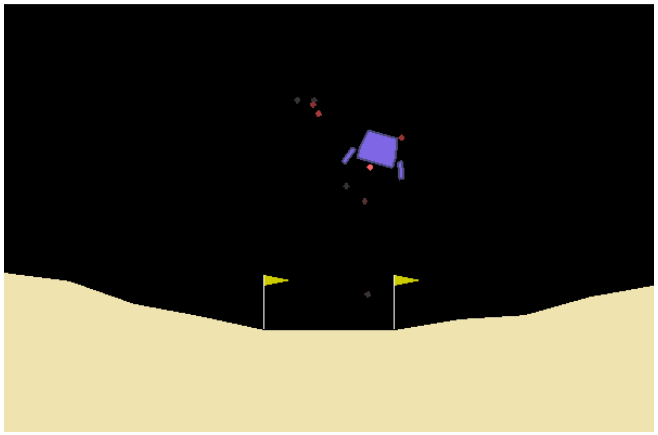

Figure 1: Frame from the rendered LunarLander environment

action spaces are the norm rather than an exception, and thus a method for dealing with these would be a great step forward. Fortunately, the researchers at DeepMind did not disappoint, and have devised a way to deal with these continuous action spaces. The method is called Deep Deterministic Policy Gradient (DDPG) [4] and can very bluntly be described as a combination of Deep Q-Learning and actor-critic (AC) methods.

To demonstrate this method, in this report a DDPG will be applied to one of the OpenAI Gym environments [5], called LunarLanderContinuous-v2. The goal is to land the lander between the flags, upright, by firing the main and side thrusters using as little fuel as possible. The OpenAI Gym provides different environments suited to learn reinforcement learning, and is an attempt to start a benchmark for general reinforcement learning, similar to the function of ImageNet [6] for image classification agents. Since no definitive method yet exists to determine optimal network architecture for a given problem, different network architectures will be trained and tested and a simple hyperparameter study will be conducted.

The reason for using the LunarLanderContinuous-v2 environment is twofold. Firstly, the environment translates really well to real-world problems, as landing a thruster-powered vehicle on a surface in a given location is in itself a real-world problem, often attempted to solve in bio-inspired ways [7, 8]. The continuous action space contributes to this real-world relevance. Secondly, the environment is very suited for implementation in Python, such that no unnecessary time and effort should be put towards getting it to work and figuring out how to get states and rewards out of the environment and getting the actions back in. A frame from the rendered version of the environment is shown in Figure 1

2 Reinforcement Learning background

Getting into the world of Reinforcement Learning [9], it becomes quite evident that there is a large amount of concepts and methods of which an understanding is required before it can be attempted to meaningfully read a state-of-the-art paper on the subject. In this section, a brief overview is given of the concept and basic knowledge that is assumed in most literature and that forms the core of the field.

2.1 Main concept

The main concept of Reinforcement Learning is inspired by the way (intelligent) organisms learn in nature. This comes down to a concept where the *agent* (analogous to the organism) can take *actions* which influence an *environment*. This environment, in response, changes its *state* (mathematical description of relevant parameters) and gives a *reward* (measure of how 'good' or 'bad' the action was) to the agent. The agent also observes the new state and uses this information to formulate a new action. This kind of discrete-time process, where the environment can take a state $s \in S$ and the agent can take action $a \in A$ which causes the environment to go to state s' with probability $P_a(s, s')$ to receive reward $r_a(s, s')$, is called a *Markov Decision Process* (MDP). The loop is shown in Figure 2. The objective of the agent is to acquire the highest possible cumulative reward.

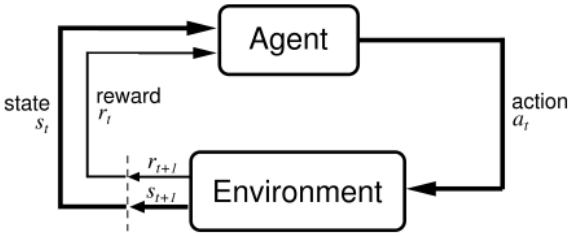

Figure 2: Basic Reinforcement Learning loop

Usually, an agent will not get a reward for every action it takes. When thinking about the Atari games, the agent only gets a reward once it scores a point or something similar, but not every frame (frames are the input states). This makes it very hard for the agent to determine which behaviours lead to rewards, as good behavours are not immediately rewarded and bad behaviours are not immediately punished. This dilemma is called the *sparse reward problem* and to get around it, a practice is called *reward shaping* is deployed, where intermediate rewards are introduced to nudge the agent in the right direction.

If the reward shaping is done incorrectly, the agent may find a way to *exploit* the environment's rewarding mechanism in ways that were not intended, and do not stimulate the desired behaviour. Additionally, finding a way to get high rewards may also mean that the agent has found a local optimum in the solution space, which is why the agent should not only exploit (i.e. get high rewards), but also occasionally *explore*. The question when to do which is usually denoted the *explore-exploit dilemma*.

2.2 Value function

As mentioned above, a Reinforcement Learning agent will attempt to maximise its rewards. The mathematical expression for the expected rewards is called the *value function* and is shown in Equation 1. This equations denotes the expected reward in state s following policy π (so all following actions are determined by π), where all rewards from the next state onward are discounted by γ, such that close rewards are weighed higher than rewards very far in the future.

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle|s_t = s\right\} \quad (1)$$

Another important function is Equation 2, which is the state-value-function or Q-function. This function is very similar to the value function but instead of following policy π from state s, action a is taken from state s and policy π is followed thereafter. Action a as such does not have to be the next action as described by π. This function allows for exploration as it can give a value to the question: What action would be best given the state and policy?

$$Q^{\pi}(s,a) = E_{\pi}\{R_t|s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle|s_t = s, a_t = a\right\} \quad (2)$$

Both functions can evidently not be calculated for continuous action spaces, so instead function approximation methods are used to get relevant values for V and Q. The performance of value function approximations can be evaluated using the Bellman optimality equation presented in Equation 3.

$$V^{\pi}(s) = E_{\pi}\{r_{t+1} + \gamma V^{\pi}(s_{t+1})|s_t = s\} \quad (3)$$

2.3 Model-based and model-free

A distinction in the Reinforcement Learning field is the separation between model-based and model-free approaches. The difference between these is that in the model-based approach, the agent learns or is embedded with a mathematical description of the environment, which allows it to predict future states and rewards.

This approach is generally more data-efficient for simpler problems [10], but recently, when no model of the environment is available (and the environment is potentially complex), preference is given to model-free approaches[1]. Other reasons for choosing a model-free approach are that we don't want to assume there is a model, or that we don't want to wait until a model is learnt when the model can be complex but the control law can be quite simple [11]. DDPG is a model-free method.

2.4 On- and off policy

Another distinction often made in the field is that between on- and off-policy learning. In on-policy learning, the agent learns the value function when following a specific policy. Contrary, in off-policy learning the agent uses one policy to select actions, and another to learn the value function following an optimal policy. DDPG is an off-policy method.

3 Related Work

In this section, an overview is given of the relevant research leading up to the state-of-the-art in Reinforcement Learning. A number of influential techniques and methods are touched upon to give an overview of the field and give context regarding the method implemented in this report.

[1] https://ai.stackexchange.com/questions/4456/whats-the-difference-between-model-free-and-model-based-reinforcement-learning

3.1 Q-learning

One of the most well-known algorithms for reinforcement learning is the Q-Learning algorithm [12]. This algorithm is usually explained using finite, small and discrete state and action spaces. It is a powerful algorithm for simple MDPs, but does not do well when the action space grows too big, such as with many-degrees-of-freedom systems or discretised continuous action spaces. Q-learning is also a model-free, off-policy method.

The Q-Learning algorithm works by 'discovering' the Q-value (as in Equation 2) for each state-action pair and storing these in the Q-table. Through exploration of the space, the Q-value for each state-action pair is determined, and as the agent becomes more experienced, the Q-table will start to approximate the true distribution of Q-values. When the agent has sufficiently explored the environment, it will know the expected reward for each state-action pair, and can simply select the action with the highest expected reward. This discourages exploration, so Q-learning is usually combined with an ϵ-greedy strategy, where the agent will take the best action with probability 1-ϵ and a random action with probability ϵ.

The Q-learning algorithm is very influential in the Reinforcement Learning field, and thus multiple variations have been devised to increase its performance, such as Double Q-Learning [13], which addresses the positive bias problem by using separate Q-functions to determine the maximising action and another to determine the Q-value associated with that action, and then alternating them.

3.2 Deep Q-Networks

The work mentioned in the introduction is a combination of the theory behind DNNs as function approximators and the Q-learning algorithm mentioned above. Apart from the advantages of using DNNs as function approximators for the value function, this method is easily extensible to take advantage of the advances in deep learning for processing of high-dimensional data [14]. The Deep Q-Network algorithm [1] utilises this to create a end-to-end solution for playing Atari games. In this method, the Q-table is replaced by a neural network, and the input states are changed from the low-dimensional state representation to a high-dimensional raw pixel data.

Another important innovation from this work is the introduction of *experience replay*, or the replay buffer, which stores 'memories' (state-action-reward-next state combinations) that the agent has seen. The agent can then sample from the replay buffer when learning. This improves generalisation because it brings the samples closer to an i.i.d. compared to when the samples were obtained sequentially from the environment.

DQN also introduces the concept of *target networks*, which is a copied network used in calculating the loss. This improves stability as the loss is of a parameterised network is dependent on its own parameters, creating a sort of 'chasing your own tail'-situation, which introduces instability.

This approach also exhibits the positive bias problem, like Q-learning and was improved in a similar way to the

Q-learning algorithm by generalising the double-Q function approach [15] to the DQN method.

3.3 Policy gradient and Actor-Critic methods

Policy gradient methods maximise the cumulative future reward and thus use gradient ascent of a (usually stochastic) parametric policy function with respect to the expected reward. The most widely used policy gradient method is the REINFORCE algorithm [16]. Policy gradient methods are already able to deal with continuous or highly discretised action spaces. However, these methods still suffer from a number of problems, including instability and slow convergence[2].

This is where the Actor-Critic methods and the connection to the Q-function come in: When decomposing the expectation function of the policy gradient method, an expression for the Q-value can be isolated. The action-value-function is learned by the critic (taking in the action and state and outputting the value), while the policy (state to action) is learned by the actor. Based on this theory, a number of actor-critic methods was developed [17].

4 METHOD

The method followed in this implementation closely follows the method of [4], called Deep Deterministic Policy Gradient (DDPG). This method is appealing as it is one of the most recent, state-of-the-art methods for solving model-free reinforcement learning problems with continuous state and action spaces. A short explanation of the algorithm will be given here, but for a more detailed explanation please refer to the paper. The alterations made for this project will be emphasised at the end of this section.

4.1 DDPG

Building on the theory of Deterministic Policy Gradients [18] and the AC methods of subsection 3.3 DDPG [4] expands this by using an actor that deterministically maps states to actions and a critic that learns the Q-function. In addition to these two NNs (actor and critic), two other NNs (called target networks) are used (see subsection 3.2. The difference with DQN is that the target networks are related to the actor and critic networks by the relation shown in Equation 4 and 5, instead of a hard copy. The influence of τ on the efficacy of this method is investigated in this project.

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \tag{4}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \tag{5}$$

To further increase stability, batch normalisation [19] is applied to the mini-batches that are learned from. This is done because the states the agent receives may contain any continuous value describing the environment, and it is not necessarily evident on which scales they are. By normalising them, the scale effect is negated.

Exploration with deterministic policies is not always straightforward, as given a certain state the agent will always choose the same action. For this reason, noise is added to the action as determined by the actor, and for DDPG the authors used the Ornstein-Uhlenbeck [20] process as a noise generator.

4.2 Implementation and alterations

The DDPG implementation for the LunarLanderContinuous-v2 environment by Dr. Phil Tabor[3] was used and modified to fit the scope of this project. The repository accompanying this report contains a folder with the relevant unaltered, original code. This subsection will detail the modifications and additions made for this project. From here on, the code presented by Dr. Tabor will be referred to as 'original code' and the final code used for this project will be referred to as 'modified code'.

The main additions were done with the goal of conducting the experiments documented in this report: Testing the algorithm with different network architectures and conducting a hyperparameter study. To this end, the original code was expanded so it gets all important information from the configuration file (`config.json`), such that this file could easily be used to control the code and be archived for future reference. The configuration file is changed throughout the experiment by the main file (`main.py`), which holds the actual numerical settings of the experiment.

In the utility file (`utils.py`), only `plotLearning` has been largely repurposed from Dr. Tabor's repository, with only small changes to improve the graph styling. The other utility functions have been added to aid the main experiment. The files `hyperparam_check.py`, `validation.py` and `visual_inspection.py` were added to automatically execute the hyperparameter check/sensitivity analysis, code validation experiments and the rendered visual inspection of the best solution.

Probably the most invasive and key modification of the original code is the change from hard-coded, preset network architecture to incorporating a dynamic network builder that constructs the network according to the architecture specified in the configuration file. This modification is done in the DDPG file `ddpg.py`, which contains the main objects and methods used in this implementation of DDPG. The network architecture can be specified using a list where the index is the layer and each index contains an integer specifying the number of nodes in that layer. The input and output layers are determined by the environment.

4.3 Experiment

The actual experiment consists of a number of training runs of the DDPG algorithm on the "LunarLanderContinuous-v2"-environment of OpenAI gym [5]. To investigate the influence of differing network architectures on the performance of the actor; or in other words: The influence of the network architecture on how well the network can approximate an ideal policy function.

[2] https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f

[3] https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/DDPG/pytorch/lunar-lander

The steps below were taken:

1. Reproduction of Dr. Tabor's results with original code for validation of the code.
2. Reproduction of Dr. Tabor's results with the modified code for validation of modifications.
3. Triple learning runs with different network architectures and standard hyperparameters.
4. Visual inspection of the best trained architecture to check for interesting solutions.
5. Learning runs with varied hyperparameters for standard network architecture ([400, 300]).

The first two steps are mainly for validation purposes, to ensure that both the original code and results are accurate, and to ensure that the modifications made to the code did not cause any unintentional side-effects.

The third step is the main experiment, where the network architecture is varied according to Table 1 and trained. It should be noted that there exists an infinite amount of network architectures and no definitive method for determining a priori a feasible architecture has yet been discovered, so the tested architectures are chosen somewhat arbitrarily around the architecture used in the original paper. The results are aggregated and saved for analysis. The main experiment is carried out three times to obtain ensemble results, necessary because of the stochastic nature of the experiment due to the action signal noise and environment interaction.

In step four, the best trained agent's networks are loaded and rendered environments are visually inspected to check for interesting or unexpected solutions.

No.	Layers	Number of nodes	Parameters
1	1	[25]	275
2	1	[250]	2 750
3	1	[1000]	11 000
4	2	[40, 30]	2 050
5	2	[400, 300]*	124 500
6	2	[1000, 800]	811 400
7	3	[40, 40, 25]	3 075
8	3	[500, 400, 250]	305 650

Table 1: Specification of the tested network architectures
*Architecture from original paper

The fifth and final step is a sensitivity analysis or hyperparameter check, to get an insight into the effect of different hyperparameter values on the final result. In this experiment, the batch size sampled from the replay memory and the τ-value (the weight with the target networks are updated) are varied according to Table 2. The sensitivity analysis uses the standard τ-values multiplied by and divided by a factor 100, as a variation within the same order of magnitude may not be significant. For the batch size, the standard value is halved and doubled as due to the nature of the batch size parameter, these are quite large differences. The default architecture is then trained for every combination of these varied hyperparameters.

For reference, relevant settings that were not varied over the course of the experiments are documented in Table 3

τ-value	Batch sizes
0.1	32
0.001	64
0.00001	128

Table 2: Hyperparameter values for check, middle row are default values.

General characteristics	
Seed	0
Number of episodes	1000
Activation function	ReLU
Maximum memories	$1 \cdot 10^6$
Actor learning rate (α)	0.000025
Critic learning rate(β)	0.00025
Discount rate (γ)	0.99
Ornstein-Uhlenbeck noise characteristics	
σ	0.15
θ	0.2
dt	0.01

Table 3: Unvaried settings

5 Results

In this section, the results from the experiment will be presented. The structure of this section follows the experiment steps detailed in subsection 4.3. In Table 4, the available computational resources are listed to give insight into the computational load of the learning process. Additionally, the numerical score values that are plotted can be found in the accompanying Github repository.

Device	Type	Clock speed	Cores
CPU	Intel Core i5 4690K	3.9 GHz	4
GPU	Nvidia GTX 980	1,317 MHz	2048

Table 4: Specifications of the computational devices

5.1 Validation

In steps 1 and 2, the original code was validated and the modifications made for this project were validated as well by a reproduction of earlier presented results and a comparison of these newly generated results. The runtime required on the earlier listed specs is presented in Table 5 for comparison (it is expected that times are similar) and to give insight into the required compute.

	Original code	Modified code
Run 1	02:14:58	03:01:28
Run 2	02:08:38	02:39:22
Run 3	01:56:16	03:16:26

Table 5: Total runtime of learning experiment (HH:MM:SS)

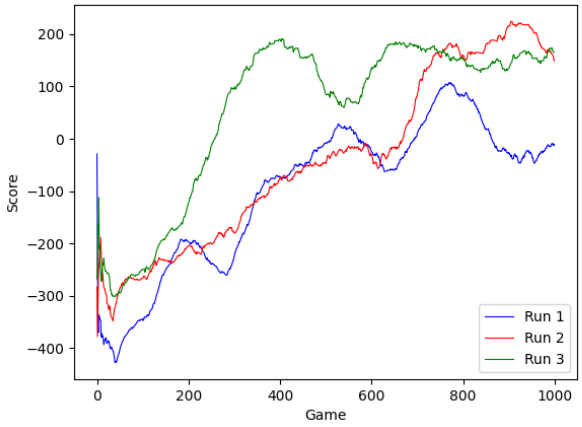

Figure 3: Three validation runs of original code (100-game trailing average score)

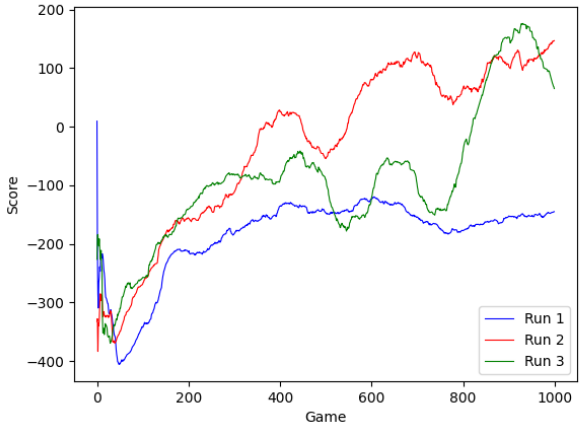

Figure 4: Three validation runs of modified code (100-game trailing average score)

From the compute times, it can be concluded that both instances used a similar amount of compute, which supports the notion that the operations did not have unforeseen errors. Actually, in one of the earlier runs a significant discrepancy in runtime (order of magnitude difference) was observed, which lead to the discovery of a programming error in the modified code. It has since been fixed. The differences in compute time is attributed to a number of factors: How the agent acts determines how fast a single episode is over, i.e. when the agent immediately crashes, an episode is very short. The added noise to the action signal also introduces differences in the outcomes of the environment, which in turn creates differences in training the networks. The difference in computational resources used are attributed to this randomness.

From Figure 3 and 4 it can be seen that the behaviour over time follows a roughly upward trend, although the agent is not continuously improving. This is true for both versions of the code (original and modified), which in turn supports the notion that the operations did not have unforeseen errors. Additionally, this is the expected behaviour of the agent (general but not necessarily continuous improvement) under the DDPG algorithm, which validates the implementation.

5.2 *Experiment results*

As stated in step three of subsection 4.3, the main experiment consists of trying out different neural network architectures for default hyperparameter settings to investigate the influence of NN architecture on approximation power of the ideal policy under default hyperparameters. The results can be found in Figure 5, 6 and 7. In Appendix A, the same results are plotted by architecture instead of run. The times spent for each training run are presented in Table 6.

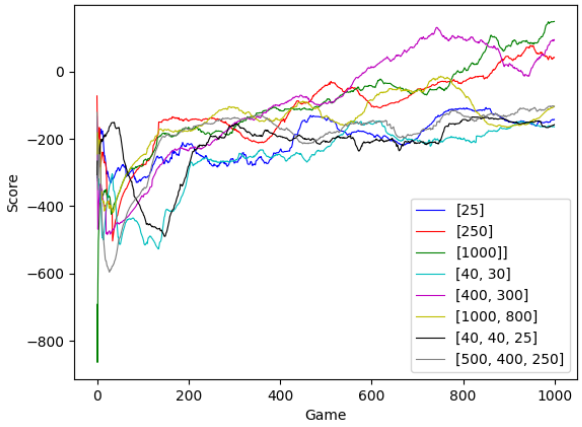

Figure 5: Score with different NN architectures, first run (100-episode trailing average score)

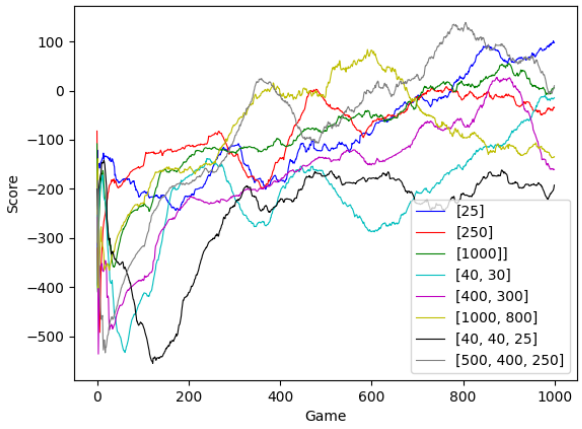

Figure 6: Score with different NN architectures, second run (100-episode trailing average score)

At first, it is obvious that there is quite a lot of difference between the runs. These differences are attributed to the randomness introduced by the action signal noise and the subsequent training steps that are dependent on what happens due to the action signal. Additionally, all the figures share the commonality that the beginning of the plot (episode 0 - 100), a sharp drop in performance appears to show. This is not actually a drop in performance, but rather the effect of plotting the 100-episode trailing average, which still needs

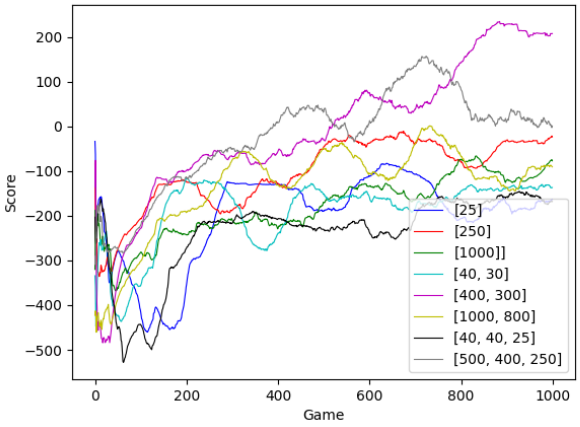

Figure 7: Score with different NN architectures, third run (100-episode trailing average score)

to 'establish' in the first 100 episodes, and as it appears, the agent receives a rather high score in the beginning (due to chance), which pulls up the performance, but the average performance in the beginning of training is quite poor, pulling down performance over the first 100 episodes. After this, training generally improves average performance.

Analysing for the different architectures, it can be seen that architectures with a low number of nodes generally perform worse than architectures with a higher number of nodes. A notable exception to this is in Figure 6, where the smallest network actually performed the best at episode 1000 (but it can also be seen that all networks performed worse in the second run than in run one and three). The [40, 40, 25] network performed badly in all three runs and the [40, 30]-network performed in the lower tier in run one and three compared to the rest, and performed badly in all runs when looking at scores.

The original architecture ([400, 300]) performed quite well in all runs, which shows that the authors of the DDPG paper did a good job at optimising their architecture. Additionally, the [1000] network performs quite well, but strangely the performance of the [1000, 800] network lags a little behind the [1000]- and [400, 300]-network's performances. This can be attributed to the number of learnable parameters of this network (see Table 1), which is significantly higher than that of the well-performing networks. This points to the [1000, 800]-network being a little too complex for the task at hand, and needing more training data to properly train to a good policy function approximation. The networks with multiple tens of thousands parameters seem to be in the sweet spot regarding complexity and training data required/given. This can also be seen with the [500, 400, 250]-network, which has 305 650 parameters and performs quite well in runs two and three, and averagely in run one.

From the time spent shown in Table 6, it is found that the small networks take significantly less time to complete 1000 episodes than their larger counterparts. This can be attributed to a number of factors: Firstly, small networks take less compute to forward- and backward-pass, and as such need less time. Secondly, the smaller networks may not be able to learn the required control policy to properly land the Lunar Lander, causing it to crash instead, which is obviously a much quicker way to get to the ground. An episode is over when the lander lands/crashes, so this could contribute significantly. Lastly, there is a degree of randomness to all these experiments which shows in the time spent as well, as for example the smallest network in run two takes quite a lot of time, where the other small networks ([25], [40,30] and [40, 40, 30]) are spending significantly less time.

No.	Architecture	Run 1	Run 2	Run 3
1.	[25]	01:08:20	01:31:51	00:26:47
2.	[250]	02:01:16	01:53:49	01:31:46
3.	[1000]	02:14:22	01:55:56	00:58:39
4.	[40, 30]	00:38:58	00:45:24	00:32:33
5.	[400, 300]	02:34:57	02:09:12	02:32:30
6.	[1000, 800]	02:52:34	02:40:00	02:37:55
7.	[40, 40, 25]	00:42:32	00:35:45	00:36:23
8.	[500, 400, 250]	02:25:58	02:22:40	01:55:48

Table 6: Time spent playing 1000 episodes for each architecture per run (HH:MM:SS)

5.3 Visual inspection

The visual inspection of the most well-trained network at episode 1000 (which was the [400, 300]-network of the third run) shows that the lander does not take any unexpected solutions (an example of such a solution would be if the lander would flip 360 degrees before landing).

In most games, the agent seems to have mastered the main thruster control, to let the lander drop in the beginning and then sharply decelerate shortly before hitting the ground, like a suicide burn. This indeed maximises the score as score is subtracted for firing the thruster. The agent seems to have adequate, but less fine control over the lateral movements of the lander, and it has the tendency to overshoot the side-thrusters and land close to one of the flags, instead of in the middle. This sometimes yields a problem when the landing surface has a convex-like shape, and the lander glides off it when landing too close to the edge of the landing zone. An example of this is shown in Figure 8. On the other hand, a concave-like surface geometry sometimes has the lander glide into the landing zone. The agent also seems to favour landing close to or on the left flag rather than the right.

5.4 Sensitivity analysis

In this subsection, the results of the (limited) sensitivity analysis or hyperparameter check are presented. The hyperparameters were varied as shown in Table 2 for the default NN architecture. These results correspond to step 5 of the experiment and are shown in Figure 9 (legend shows separately in Figure 10, because of readibility issues when displaying inside the plot).

From the plot, it is immediately obvious that the net-

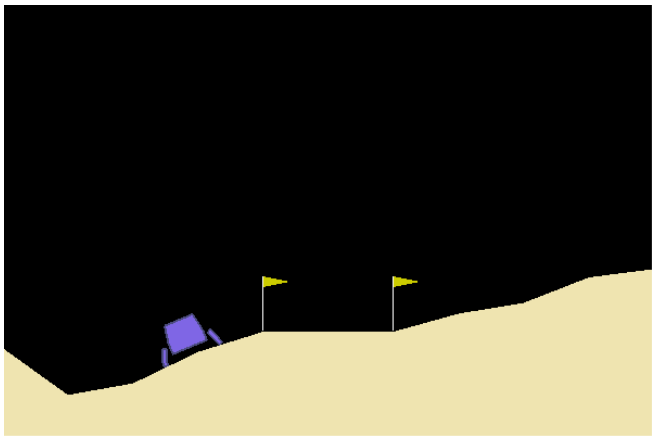

Figure 8: Convex-like landing surface where lander is gliding away from the landing zone.

works with $\tau = 0.00001$ perform quite bad, they don't really seem to learn. This can be attributed to the low τ-value, which means that the update of the target networks happens extremely slowly, so much so that over 1000 episodes, the target networks do not update enough to actually help the actor and critic network to learn from the training data over the 1000 episodes. More episodes may help when using low τ-values. The τ-value of 0.1 performs better for larger batch sizes, but worse for the small batch size.

The effect of batch size on performance is inconclusive, as from the results of the sensitivity analysis no clear relationship between batch size and performance is found. More test runs may yield a clear relationship, or varying over a larger range of batch sizes may clarify a relationship. It is also possible that the batch size setting is not very consequential, but given its role in the algorithm, this seems unlikely.

The default hyperparameter values of τ=0.001 and batch size of 64 performs quite well, suggesting the authors of the DDPG paper did a proper hyperparameter optimisation.

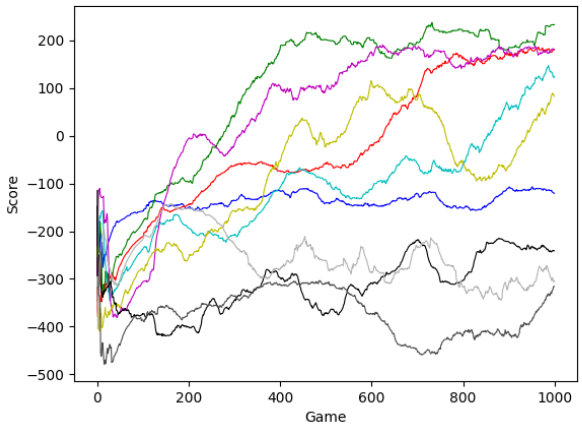

Figure 9: Scores with different hyperparameter settings (100 game trailing average score)

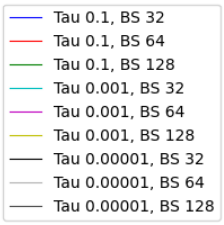

Figure 10: Hyperparameter settings for the different training runs, BS is batch size (legend for results plot)

6 Discussion

In the context of computationally heavy tasks, it is always interesting to think about what could have been done additionally if more compute were available. In this project, an endeavour was made to obtain ensemble results for all experiments (running the experiment three times in the case of the validation and main experiment) due to the partially random nature of the learning process. However, due to limitations in computational power (total runtime was already around 90 hours cumulatively), for the sensitivity analysis it was not possible to obtain ensemble results. Rerunning the sensitivity analysis two times to obtain the triple results would add approximately 30 hours, which was deemed to be infeasible.

Of course, the same could be said for all other experiments, which, in case of unlimited compute, should have been done much more often to increase confidence in the validity of the conclusions drawn from said results.

With the goal of producing the best algorithm for solving the environment, a good iteration of the actor and critic could be used and trained further, possibly improving performance even more. This would be an interesting investigation for follow-up studies as it was deemed outside the scope of this project.

This project demonstrates that an agent can learn, given an environment, to land a lander, but it is yet unclear how well this translates to the real world. An interesting study would be if this architecture is also sufficient in terms of approximating power to perform in a more complex environment (more states, more realistic dynamics).

7 Conclusion

Neural networks remain a topic of heavy research, and when it comes to architectures there is no ubiquitous way of optimising for a specific task. Therefore, when employing one of these networks, one should always investigate the effects of architecture and hyperparameter choices, and optimise (iteratively). The question central to this project is that of the choice of architecture for the DDPG paper, which seems to be rather well-optimised as the original architecture and settings create well-performing agents. It seems networks that are much smaller cannot achieve an adequate approximation power to approximate the underlying ideal policy function, and networks that are too large (complex) require more training data to match performance after 1000 episodes. In the ideal case, more architectures would be

tested with more reruns, providing ensemble results and building more confidence in the findings of the experiments, but available compute limits this.

There have also not been significant increases in performance when varying the target network update parameter, however the sensitivity analysis suggests that it may be beneficial to use a larger value for this parameter, along with larger batch sizes, but this is not conclusive due to the single run combined with the inherent randomness of the experiments. Collecting ensemble results for the sensitivity analysis may substantiate this claim.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[7] B. J. Pijnacker Hordijk, K. Y. Scheper, and G. C. De Croon, "Vertical landing for micro air vehicles using event-based optical flow," *Journal of Field Robotics*, vol. 35, no. 1, pp. 69–90, 2018.

[8] B. Herissé, T. Hamel, R. Mahony, and F.-X. Russotto, "Landing a vtol unmanned aerial vehicle on a moving platform using optical flow," *IEEE Transactions on robotics*, vol. 28, no. 1, pp. 77–89, 2011.

[9] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[10] C. G. Atkeson and J. C. Santamaria, "A comparison of direct and model-based reinforcement learning," in *Proceedings of international conference on robotics and automation*, vol. 4. IEEE, 1997, pp. 3557–3564.

[11] H. Van Hasselt and M. A. Wiering, "Reinforcement learning in continuous action spaces," in *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE, 2007, pp. 272–279.

[12] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[13] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, pp. 2613–2621, 2010.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[15] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[16] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.

[17] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[18] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*. PMLR, 2014, pp. 387–395.

[19] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[20] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Physical review*, vol. 36, no. 5, p. 823, 1930.

APPENDIX A ADDITIONAL PLOTS

These are the performances sorted by architecture.

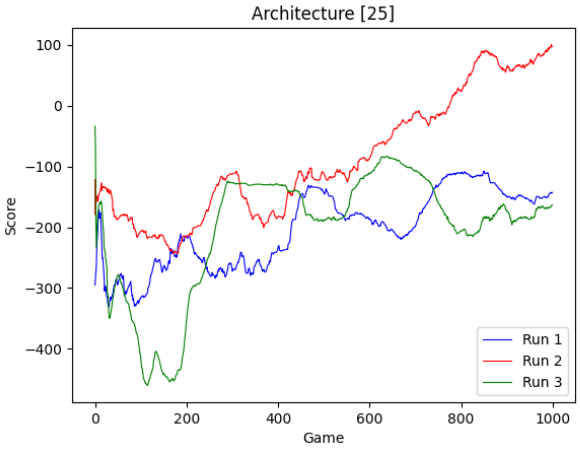

Figure 11: Performance of the three runs for [40]-network

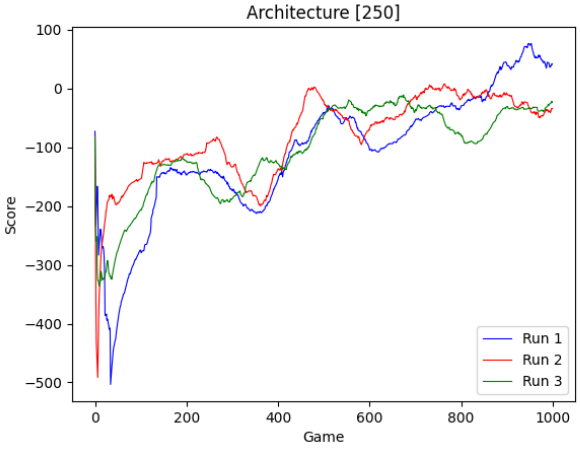

Figure 12: Performance of the three runs for [250]-network

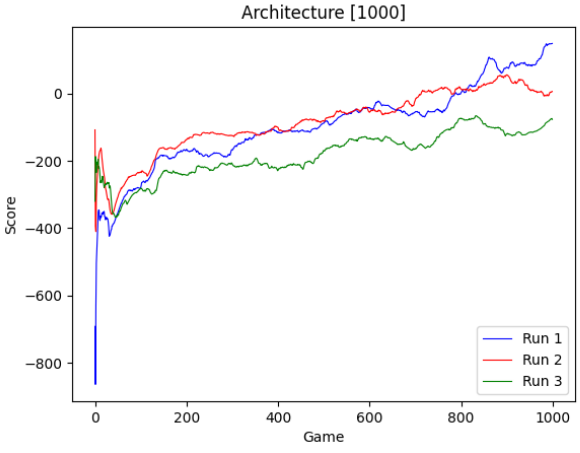

Figure 13: Performance of the three runs for [1000]-network

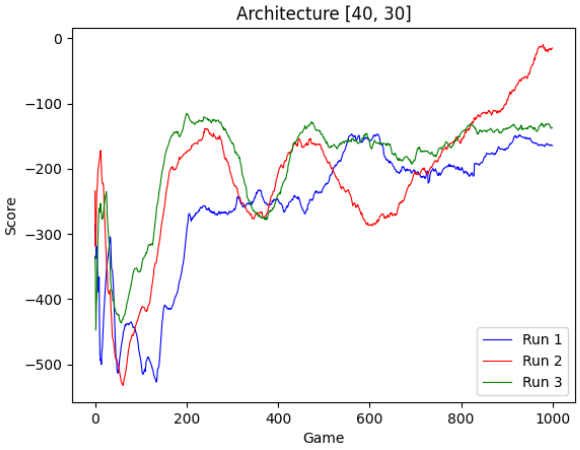

Figure 14: Performance of the three runs for [40, 30]-network

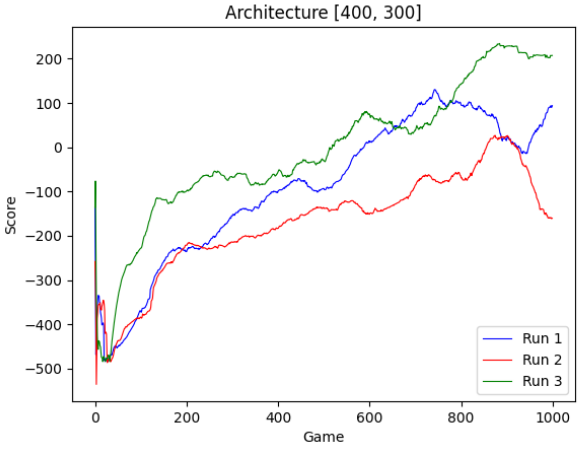

Figure 15: Performance of the three runs for [400, 300]-network

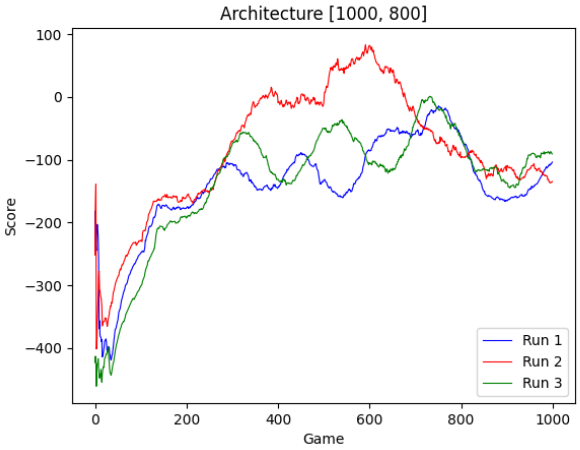

Figure 16: Performance of the three runs for [1000, 800]-network

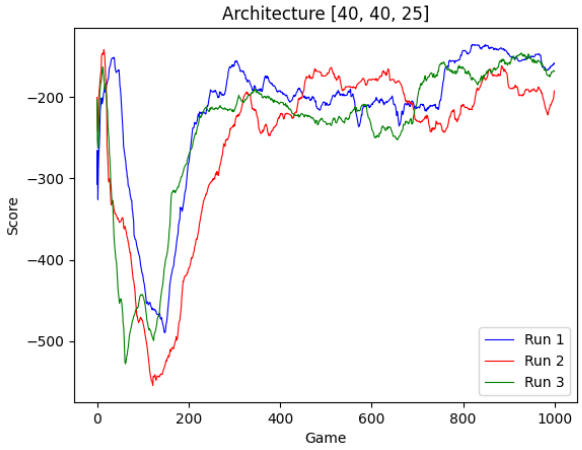

Figure 17: Performance of the three runs for [40, 40, 25]-network

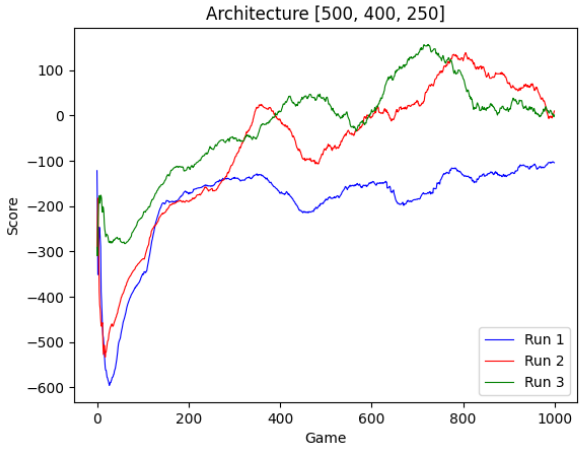

Figure 18: Performance of the three runs for [500, 400, 250]-network