Reinforcement Learning Assignment

Paul Couzy[1] (s1174347) couzy@strw.leidenuniv.nl
Eva van Weenen[1] (s1376969) vanweenen@strw.leidenuniv.nl

Leiden Observatory, The Netherlands

Abstract. We apply Reinforcement Learning to the Lunar Lander Open-AI Gym environment. Using Monte-Carlo Policy Gradient and Parameter-exploring Policy Gradient we optimize the policy that maximizes the total reward. We find that Monte-Carlo Policy Gradient takes 500-800 episodes to solve the environment. For this algorithm we vary over different γ and find that γ should be in the range of 0.98-0.99 to solve the environment. As Monte-Carlo Policy Gradient is not the most efficient method due to a high variance after a large number of episode, we apply Parameter-exploring Policy Gradient. We find that it solves the environment after 30-50 episodes and that a population size of $N = 600$ is preferred above $N = 500$ for finding a quicker solution.

Keywords: Reinforcement Learning · Machine Learning · Policy Gradient · Monte-Carlo Policy Gradient· Parameter-exploring Policy Gradient · Neural Networks

1 Introduction

In the past century, humans have tried to program computers to imitate their behaviour through the process of "learning" – gaining or acquiring knowledge of or skill in (something) by study, experience, or being taught – and this field was rapidly called *Machine Learning*. Machine learning has now been applied in many areas, such as classification, regression, clustering and dimensionality reduction. *Reinforcement Learning* is one of the oldest fields of Machine Learning and has applications in self-driving cars, robots and playing board games. It is based on behaviourist psychology. In 2013 a revolution in Reinforcement Learning took place when English researchers outperformed humans in common Atari games Mnih et al. (2013a). Not many years later, in 2016 their setup AlphaGo was able to defeat the world champion of the game Go, Lee Sedol Mnih et al. (2015).

In Reinforcement Learning, a software **agent** interacts with an **environment** in discrete time steps. The agent makes **observations** o_t and takes **actions** a_t in order to maximize the cumulative **reward**. After taking the action, the environment moves to a new **state** s_{t+1} and the associated reward r_t is calculated. Almost every problem in Reinforcement Learning can be described as a Markov Decision Process (MDP), which helps making decisions on a stochastic environment. A Markov state contains all useful information from the history, thus once the current state is known, all history can be erased. In finite Markov

Decision Processes, actions do not only influence immediate rewards, but also future rewards. A trade-off then needs to be made between immediate and future rewards.

The **value function** $v(s)$ indicates the expected total reward and in this way indicates how good it is to be in a given state. The algorithm that an agent uses to decide which action to take is called the **policy** $\pi(a|s)$. The policy is the probability of an action $A_t = a$ given a state $S_t = s$. The **action-value function** q_π is then the value of an action a given a state s under policy π.

There are different ways to evaluate and find the optimal policy. One method is the **Policy Gradient** method. Policy gradient methods search policy space by using gradient descent and are specifically good at solving high-dimensional environments with continuous states and actions (Benbrahim and Franklin, 1997). In this project we will use two different Policy Gradient methods: Monte Carlo Policy Gradient (section 3) and Parameter-exploring Policy Gradient (section 4). We apply these algorithms to the Lunar Lander Open-AI Gym environment. The environment will be further explained in section 2.

Notation: In order to prevent ambiguities, we use the following notation in the rest of the report: *italics* for scalars (P, μ), ***bold italics*** for vectors ($\boldsymbol{s}$, $\boldsymbol{z_1}$), and ***bold capitalized italics*** for matrices ($\boldsymbol{W}$).

2 Lunar Lander

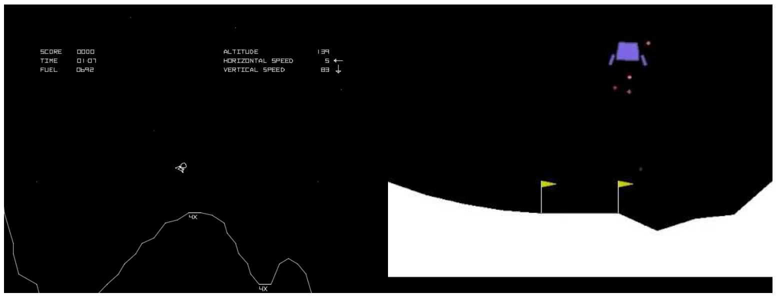


Fig. 1: On the left picture one can see the original Lunar Lander game. The space ship is tilted to the right and the two landing spots are marked by the 4x symbol. On the right we can see the rendered LunarLanderv2 environment. The landing path is between the two yellow flags.

Lunar Lander is a classic singe player arcade game released by Atari in August 1979. It is an established benchmark used to test artificial intelligence and machine learning. The goal of the original game was to land the space ship on the surface of the moon, hence the name Lunar Lander. Players had engines to rotate the module and one main engine to counter gravity. The game ended when a person successfully landed on the surface of the moon, crashed the space ship or ran out of fuel for the engines. In this assignment we are going to solve the Lunar Lander environment with the help of Reinforcement Learning methods.

We will use the environment provided by the Open-AI Gym (Brockman et al., 2016), this gives us the environment LunarLanderv2. This environment differs from the original Lunar Lander in some ways. The goal is again to land on the moon, but now we need to land between two flags at the center, with coordinates $(x, y) = (0, 0)$. The terrain at the center is always flat, the other terrain is random. Our input from this environment consists of a 8-state vector called $\boldsymbol{s}$,

$$\boldsymbol{s} = \left(x, y, \frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, \theta, \frac{20\partial\theta}{\partial t}, \mathbb{1}_{leg1}, \mathbb{1}_{leg2}\right). \tag{1}$$

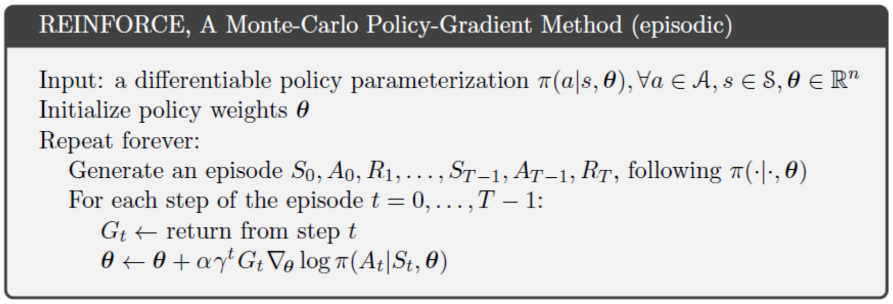
For our state-vector x and y describe the coordinates of the lander, $\frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}$ the velocity in the x and y directions, θ is the angle made by the space ship and $\frac{\partial\theta}{\partial t}$ is the angular velocity of the space ship, it is enhanced in this environment by a factor of 20, so it rotates faster than normal. The last two $\mathbb{1}$'s represents two Boolean functions, which are true if the legs touch the ground and false otherwise. The reward for successfully landing anywhere is 100 points, no matter if lands on the landing pad or outside. The reward for successfully landing on the landing pad is between 100 and 140 points. The reward for firing the main engine is -0.3 points per time step, the two side engines can be fired at no cost, unlike the original game Lunar Lander, the fuel we have is infinite. Each leg that makes successfully contact with the ground yields a reward of 10 points. Crashing the ship anywhere yields a reward of -100 points. This environment is solved we achieve the reward reaches a minimum of 200 points for ten episodes. For our space ship there are 4 discrete actions in each time step: Fire the main engine, fire the left engine, fire the right engine, do noting. Our goal is to know for each state the optimal action, so the space ship can safely land ten times to solve the environment.

3 Monte-Carlo Policy Gradient

One of the algorithms we will use to solve the Lunar Lander environment is based on the REINFORCE algorithm (Williams, 1992). The algorithm is described in pseudo code in Fig 2, at each time t the policy parameter $\boldsymbol{\theta}$ is updated proportional to the product of return G_t and a vector, the gradient of the probability of the action taken normalized by the probability of taking the action. This way the vector on which $\boldsymbol{\theta}$ is updated is towards the direction of repeating action A_t on the future visits of S_t. The size of the step is proportional to return G_t, so if the return is high, it is intuitive that one can take a big step in that direction, the step is disproportional to the action probability, this ensures that the actions with low probabilities are also taken into account. Otherwise the actions with high probability could win, despite that they are not optimal. As the method depends on chance and is updated in the direction of the Gradient it is a stochastic gradient ascent method. The return G_t is calculated in the following way:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \gamma^{T-t+1} R_T. \tag{2}$$

Our return G_t is the weighed sum of all the future rewards R_t up till R_T, where T is the end time of an episode. The γ factor is a numerical parameter, which is the discount rate of the rewards. A reward one time step away is more important than one the reward in the far future. The γ parameter is a choice of how big the trade-off is between the rewards at different times. In this short experiment we are going to change the parameter γ and see how it effects the speed and performance of the algorithm.



```
REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta}), \forall a \in \mathcal{A}, s \in \mathcal{S}, \boldsymbol{\theta} \in \mathbb{R}^n$
Initialize policy weights $\boldsymbol{\theta}$
Repeat forever:
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    For each step of the episode $t = 0, \ldots, T-1$:
        $G_t \leftarrow$ return from step $t$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta})$
```

Fig. 2: The REINFORCE algorithm described by Sutton in pseudo code.

3.1 Algorithm

Our version of the algorithm is a Monte-Carlo policy gradient. For this we use a neural network build by Tensorflow inspired by (Mao, 2017), this small network consists of three layers, the input layer gets the state from the Lunar Lander as input. The hidden layer has 32 nodes, we use the tanh function as activation function for the first two layers and those layers each get a constant bias of 0.1; the output layer has 4 nodes and gives 4 probabilities back, where each probability corresponds to one of the actions we can take for the Lunar Lander, the last layer has no activation function. This network is very similar to the Parameter-exploring policy gradient seen in section 4, for a graphical representation of the network look at Fig 4. The goal of this network to calculate the gradient. Our loss function, is given by:

$$-\log \pi(A_t|\ S_t, \theta) \times G_t. \tag{3}$$

This network uses this loss function as an simple way to calculate the gradient. The downside of Monte-Carlo is the high variance it provides, so having a robust learner can be difficult if we implement this algorithm. The upside is that the Monte-Carlo method is unbiased and does not depend on the initial values, which our network has to take.

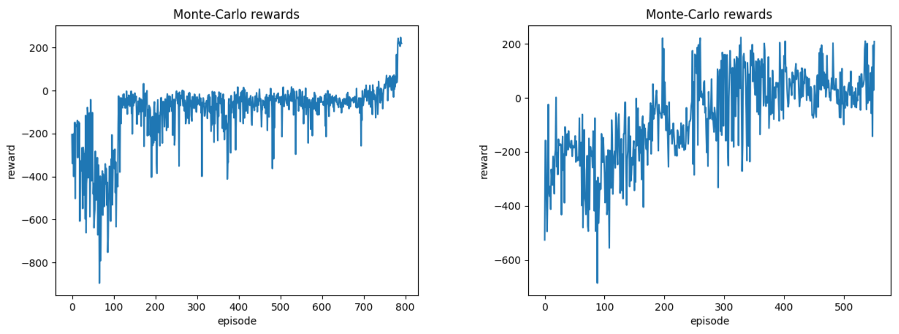


Fig. 3: On the left the reward plotted for each episode for the Monte-Carlo gradient policy with $\gamma = 0.99$ for run 2 which stopped after 792 episodes. On the right the reward is plotted with $\gamma = 0.98$, this run took 554 episodes.

3.2 Results

The γ parameter was tested with several values, it was initially set at 0.99. The values $0.90 - 0.96$ could not solve the lunar lander environment in 5000 episodes for 10 runs, it did not even solve it for one of the 10 runs. Watching the rendered game, it seems the lander only learns to hover or fly and does not learn to land properly. Because it does not land quickly or crashes. It achieves negative rewards, because of the cost due to firing the main engine. For the value of 0.97 a peculiar thing happened. It managed to solve it in 5 out of 10 runs, with 5000 episodes each. For values of 0.98 and 0.99 the algorithm seems to work smoothly.

For each run it solves the environment with an average of nine minutes per run as can be seen at Table 1. The reward for each episode is shown in figure 3. Their is quite some variance in both plots; the right figure shows that the Monte-Carlo method suffers from high variance, which was expected from the algorithm.

In hindsight it seems logical and intuitive that we need a high value of γ, the only positive reward we can accumulate is in the final time step T, that is when we safely land on the surface, no matter if it is on the landing pad or not. If takes long and γ isn't that high, the return G_t will not be high, and it will not learn to land safely between the two flags.

γ	Runtime min (s)	Runtime mean(s)	Runtime max (s)
0.95	8048	8329	9179
0.97	217	4968	8997
0.98	77	520	1587
0.99	64	522	1163

Table 1: Runtimes for different values of γ over 10 runs.

4 Parameter-exploring Policy Gradient

Our objective is to find the optimal policy to let the Lunar Lander land in between the flags and with both legs at the ground, while maximizing the reward. We use a policy gradient method, *Parameter-exploring Policy Gradient* (PEPG) to search policy-space for an optimal solution. This method has shown to outperform well-known policy gradient algorithms (Sehnke et al., 2010). Most policy gradient algorithms (also REINFORCE (Williams, 1992)) converge slowly to an optimal policy because of a high variance in their gradient. This is also observed in our analysis of Monte-Carlo Policy Gradient in section 3. As most policy gradient algorithms sample from a probability distribution, they add noise to the estimate of the gradient resulting in a high variance (Munos, 2006; Sehnke et al., 2010). The algorithm that we will use, Parameter-exploring Policy Gradient, uses a different method to approach the optimal policy. The parameters of this policy are sampled from a Gaussian distribution, making the controller deterministic (Sehnke et al., 2010).

The algorithm for Parameter-exploring Policy-Gradient we use is described in algorithm 1 and mainly based on the algorithm of Bednarski (2017). Our policy is defined by a forward pass neural network, that predicts an action from a given state and a set of optimizable weights. We optimize our policy, and thus our weights, by calculating the rewards of N random samples of weights and consecutively calculating the gradient. We first describe the layout of the network and then explain how it is applied in our algorithm.

4.1 Neural Network

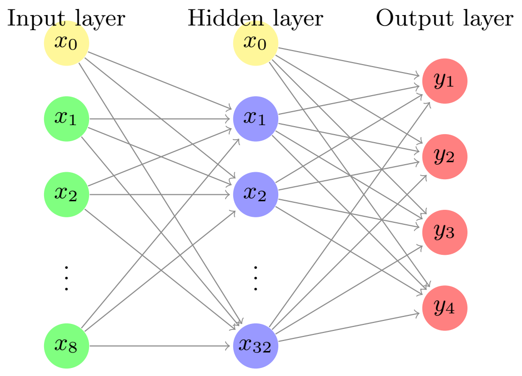


Fig. 4: Neural network for Parameter-exploring Policy Gradient with 8 input neurons (the state), 32 neurons of the hidden layer and 4 output neurons (the action). To the input and hidden layer we add a bias.

The policy we use to predict an action from a given state is a neural network. The design of the neural network can be observed in figure 4. The neural network uses as input the state-vector, has 32 hidden nodes and calculates the action as output using fully connected nodes. For the hidden and output layer we use the *tanh* activation function. The network returns the best action with the ARGMAX function, and using this selected action we calculate the reward. The calculations of the network are shown in equations 4 to 10.

$$\boldsymbol{z^{(1)}} = \boldsymbol{s} \cdot \boldsymbol{W^{(1)}} + \boldsymbol{b^{(1)}} \tag{4}$$

$$= \left(s_0 \dots s_7\right) \cdot \begin{pmatrix} W^{(1)}_{0,0} & \dots & W^{(1)}_{0,31} \\ \vdots & \ddots & \vdots \\ W^{(1)}_{7,0} & \dots & W^{(1)}_{7,31} \end{pmatrix} + \left(b^{(1)}_0 \dots b^{(1)}_{31}\right) = \left(z^{(1)}_0 \dots z^{(1)}_{31}\right) \tag{5}$$

$$\boldsymbol{\varphi^{(1)}} = \tanh \boldsymbol{z^{(1)}} \tag{6}$$

where $\boldsymbol{s}$ is the state vector, $\boldsymbol{W^{(1)}}$ is a 8×32 weights matrix between the input and hidden layer and $\boldsymbol{b^{(1)}}$ are the first bias weights. The activation function tanh is used in equation 6. We then perform the following equations

$$\boldsymbol{z^{(2)}} = \boldsymbol{\varphi^{(1)}} \cdot \boldsymbol{W^{(2)}} + \boldsymbol{b^{(2)}} \tag{7}$$

$$= \left(\varphi^{(1)}_0 \dots \varphi^{(1)}_{31}\right) \cdot \begin{pmatrix} W^{(2)}_{0,0} & \dots & W^{(2)}_{0,3} \\ \vdots & \ddots & \vdots \\ W^{(2)}_{31,0} & \dots & W^{(2)}_{31,3} \end{pmatrix} + \left(b^{(2)}_0 \dots b^{(2)}_{3}\right) = \left(z^{(2)}_0 \dots z^{(2)}_{3}\right) \tag{8}$$

$$\boldsymbol{\varphi^{(2)}} = \tanh \boldsymbol{z^{(2)}} \tag{9}$$

$$\boldsymbol{a} = \textsc{argmax}\ \boldsymbol{\varphi^{(2)}} = \left(a_0 \dots a_3\right) \tag{10}$$

where $\boldsymbol{\varphi^{(1)}}$ contains the values of the nodes in the hidden layer, $\boldsymbol{W^{(2)}}$ is a 32×4 weights vector between the hidden layer and the output layer and $\boldsymbol{b^{(2)}}$ are the second bias weights. The activation function tanh is used in equation 9. In equation 10 we apply the ARGMAX function to get binary action values.

4.2 Algorithm

We apply our neural network in the algorithm as follows. We start our algorithm by initializing scalar b, the baseline reward which is the mean reward of the last H episodes, and by initializing vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$:

$$\mu_i = 0 \qquad \text{for } i \in 1, 2, ..., P \tag{11}$$

$$\sigma_i = 0.5 \qquad \text{for } i \in 1, 2, ..., P \tag{12}$$

where P is the total number of weights in the neural network defined by:

$$P = (n_x + 1) \cdot n_h + (n_h + 1) \cdot n_y = (8 + 1) \cdot 32 + (32 + 1) \cdot 4 = 320 \tag{13}$$

Algorithm 1 PGPE for LunarLander-v2 (Bednarski, 2017)

```
 1: initialise $\mu$ with zeros ; initialise $\sigma$ with values 0.5 ; initialise $b = 0$
 2: initialise neural network
 3: function EVALUATE POLICY(policy, $\theta$)
 4:     while lunar lander has not landed or max. episodes is not reached do
 5:         predict action with forward pass through neural network
 6:         calculate reward of selected action
 7:         total reward += reward
 8:     return total reward
 9: while not solved do
10:     for $n = 0$ to N-1 do
11:         sample $\theta^{(n)}$ from $\mathcal{N}(\mu, \sigma^2)$
12:         $r^{(n)} \leftarrow$ EVALUATE POLICY(theta) with parallel programming
13:     current reward $\leftarrow$ EVALUATE POLICY($\mu$)
14:     $T_{ij} \leftarrow \theta_i^j - \mu_i$
15:     $S_{ij} \leftarrow \frac{T_{ij}^2 - \sigma_i^2}{\sigma_i}$
16:     $r^{(n)} \leftarrow r^{(n)} - b$
17:     $b \leftarrow$ mean reward of last H iterations
18:     $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \alpha_\mu \boldsymbol{T} \cdot \boldsymbol{r}$
19:     $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma} + \alpha_\sigma \boldsymbol{S} \cdot \boldsymbol{r}$
```

with $n_x = 8$ the number of input nodes, $n_h = 32$ the number of hidden nodes, $n_y = 4$ the number of output nodes and the +1 is caused by the bias. The weights of the network are flattened and stacked into a one-dimensional array $\boldsymbol{\theta}^j = (\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)})$.

We evaluate the rewards of N randomly sampled sets of weights as follows. Each different set of weights is a different policy for selecting an action, therefore we will further identify these as different policies. For each different policy $j \in N$, each weight θ_i^j $(i \in 1, 2, ..., P)$ is randomly sampled from a Gaussian distribution around mean μ_i and standard deviation σ_i, forming an $N \times P$-matrix $\boldsymbol{\theta}$. We forward pass the state $\boldsymbol{s}$ and weights $\boldsymbol{\theta}^j$ for each policy j to select an action. We perform this action and for the combination of action and state, we calculate the reward and go to the next step, until the environment is solved. For each policy $j \in N$ we return the final reward of all steps r^j. Because we calculate N total rewards independently, we use parallel programming to speed up the process. Besides evaluating the reward for each policy, we calculate the reward of the current policy using current weights $\boldsymbol{\mu}$.

We then start updating the weights as follows. We first calculate $P \times N$-matrices T and S.

$$\boldsymbol{T_j} = \boldsymbol{\theta^j}^\top - \boldsymbol{\mu}^\top = \begin{pmatrix} \theta_{0,j} \\ \vdots \\ \theta_{P,j} \end{pmatrix} - \begin{pmatrix} \mu_0 \\ \vdots \\ \mu_P \end{pmatrix} \tag{14}$$

$$\boldsymbol{S_j} = \frac{\boldsymbol{T_j}^2 - \boldsymbol{\sigma}^2}{\boldsymbol{\sigma}} = \frac{\begin{pmatrix} T_{0,j} \\ \vdots \\ T_{P,j} \end{pmatrix}^2 - \begin{pmatrix} \sigma_0 \\ \vdots \\ \sigma_P \end{pmatrix}^2}{\begin{pmatrix} \sigma_0 \\ \vdots \\ \sigma_P \end{pmatrix}} \tag{15}$$

We then subtract the mean reward of the last H episodes b from the total reward r^j for each policy j.

$$r^j = r^j - b \tag{16}$$

If the evaluated reward is higher than the mean of the last H rewards, the $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are updated with a positive factor, towards more probable actions. Lastly, we update the weights using the following equations:

$$\boldsymbol{\mu} = \boldsymbol{\mu} + \alpha_\mu \boldsymbol{T} \cdot \boldsymbol{r} \tag{17}$$

$$\boldsymbol{\sigma} = \boldsymbol{\sigma} + \alpha_\sigma \boldsymbol{S} \cdot \boldsymbol{r} \tag{18}$$

with α_μ and α_σ the learning rates for $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ respectively. Once we have updated $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ we can draw new weights $\boldsymbol{\theta}$ from Gaussian distributions around $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ and repeat the previous steps until the environment is solved.

4.3 Initializations

For our algorithm we used the initializations shown in table 2. We set the learning

Parameter	**Main initialization**	**Test initialization 1**	**Test initialization 2**
μ_i	0	0	0
σ_i	0.5	0.5	0.5
α_μ	0.0001	0.0001	0.0001
α_σ	0.00001	**0.00002**	0.00001
N	500	500	**600**
H	50	50	50

Table 2: Initializations for the PEPG algorithm

rates α_μ and α_σ to the specific values of the "Main initialization" as Bednarski (2017) showed that making both learning rates twice as high or twice as low increases computation time. This is also the case for the populations size N, the standard deviations $\boldsymbol{\sigma}$ and the history size N. In all cases Bednarski (2017) showed that increasing or decreasing these parameters increased computation time or the number of episodes considerably. The only cases in which the number of episodes was decreased was for $\alpha_\sigma = 0.00002$ and for $N = 600$. In these cases the computation time did not increase significantly. Furthermore we must note that Bednarski (2017) only ran each set of initializations once, therefore we can not rely entirely on his results. We will therefore test our algorithms for initializations that seemed promising to see whether the performance significantly increases.

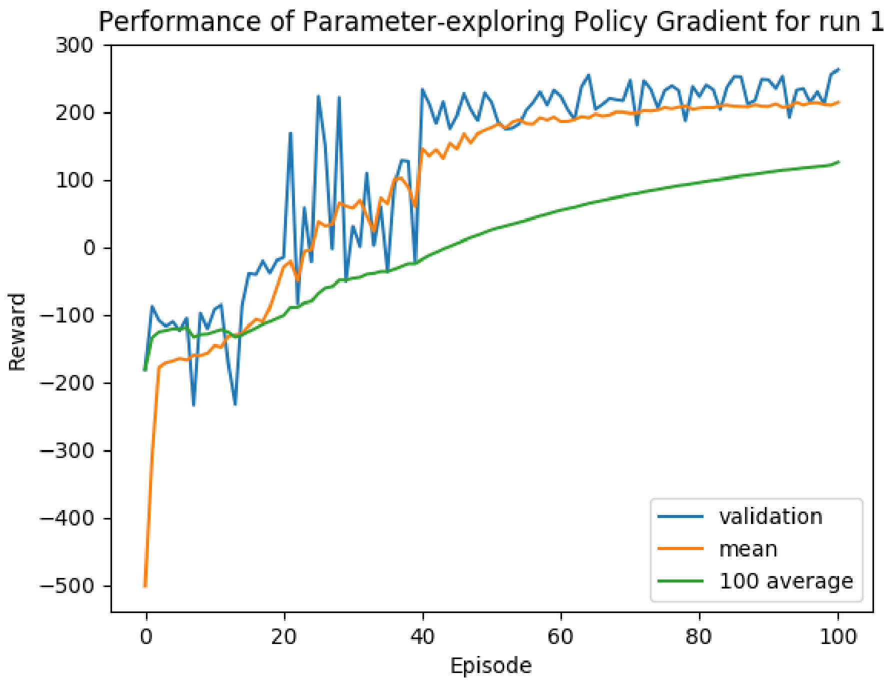
4.4 Results

Main initialization From running our main initialization ten times with ten different random seeds, we received three different types of results (see figure 5). The first type (figure 5a) occurs most often, eight out of ten times, with a fast increase towards reward 200. The reward of 200 is often first reached after 40 to 50 episodes. The other two types (figures 5b and 5c) are outliers and only occurred once. In both types we observe a rather high variance between the reward of the current weights for each episode. The reward increases gradually towards 200, and is often reached after a large number of episodes. The reason why there are sometimes in which it takes a long time to solve the environment is not entirely clear. It could be possible that crashing is observed as a very negative set of actions, therefore preventing the lander to fly towards the ground.

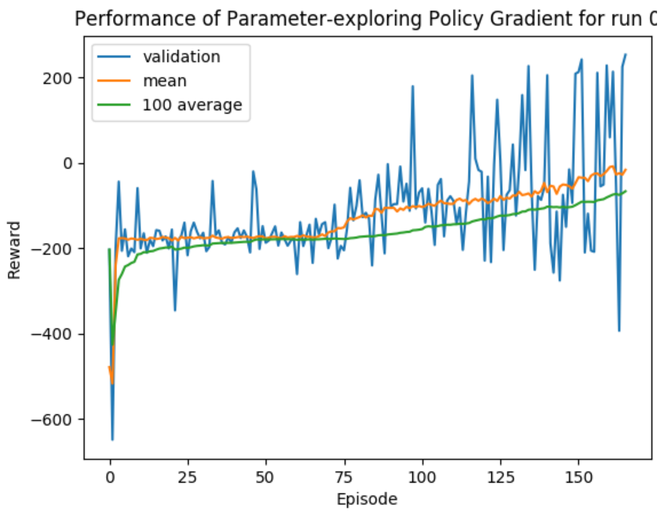
When we compare the performance of the Monte-Carlo Policy Gradient with Parameter-exploring Policy Gradient, we find that in most cases the environment is solved after much fewer episodes for PEPG than for Monte-Carlo Policy Gradient. This observation is in line with our expectations as we expected Monte-Carlo Policy Gradient to have an increasing variance for an increasing number of episodes.

Test initialization 1: $\boldsymbol{\alpha_\sigma = 0.00002}$ As the initialization of $\alpha_\sigma = 0.00002$ seemed promising in Bednarski (2017) we wanted to implement our Parameter-exploring Policy Gradient algorithm with this initialization. In contrast to Bednarski (2017) we found that $\boldsymbol{\sigma}$ was updated with such a high rate that we obtained negative standard deviations. Even setting the standard deviations to zero when they were lower than zero did not help, and only resulted in rewards of -800. We therefore conclude that we should not double the learning rate and continue with the previous initialization.

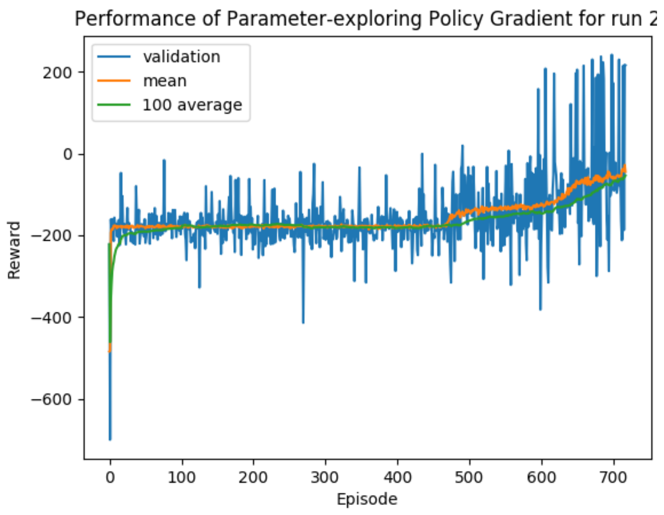
Test initialization 2: $\boldsymbol{N = 600}$ As the initialization of $N = 600$ proved more fruitful in Bednarski (2017), we evaluate this setting. We find that this setting generally converges much faster towards the reward 200. The first time this happens is generally after episode 30, see figure 6a.



(a) The most standard achieved result of the main initialization. We observe a fast increase towards reward 200. In most cases, a reward of 200 is achieved after 40 to 50 episodes.

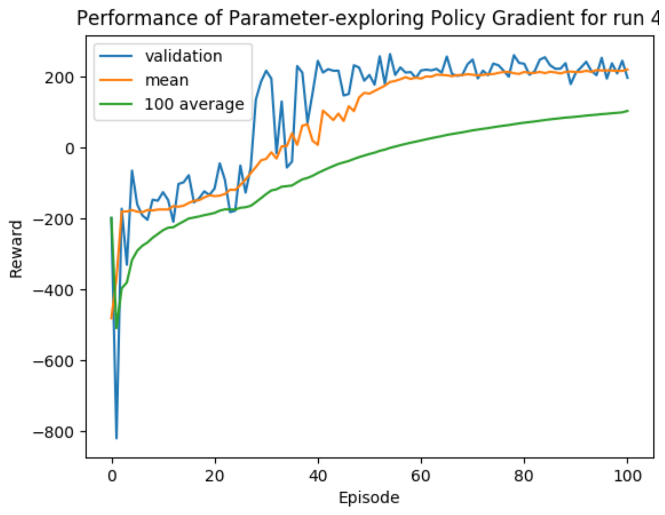


(b) We observe an increasing variance for an increasing number of episodes. The reward of 200 is achieved after a reasonable amount of episodes (100-200).

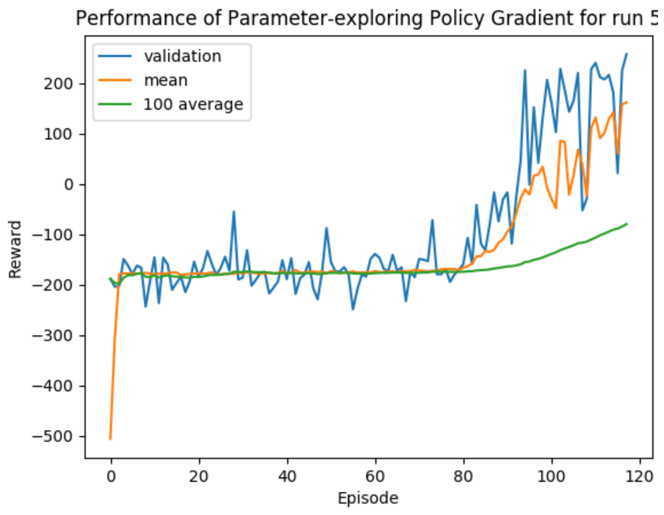


(c) We observe an increasing variance of an increasing number of episodes. The reward of 200 is reached after a very large number of episodes (700-800).

Fig. 5: Results of different runs of the main initialization. We plot the different rewards versus the episode. The orange line 'mean' shows the mean of total rewards of N different evaluated policies $\boldsymbol{r}$ for each episode, the blue line 'validation' shows the reward of the current weights $\boldsymbol{\mu}$ and the green line '100 average' shows the average of the last 100 rewards of the current weights. We observe three different types of results, shown in figures a, b and c.



(a) The most standard achieved result of test initialization 2. We observe a fast increase towards reward 200. In most cases, a reward of 200 is achieved after 30 to 40 episodes



(b) We observe an increasing variance of an increasing number of episodes. The reward of 200 is reached after a reasonable amount of episodes (100)

Fig. 6: Results of different runs of test initialization 2. We plot the different rewards versus the episode. The orange line 'mean' shows the mean of total rewards of N different evaluated policies $\boldsymbol{r}$ for each episode, the blue line 'validation' shows the reward of the current weights $\boldsymbol{\mu}$ and the green line '100 average' shows the average of the last 100 rewards of the current weights. We observe two different types of results, shown in figures a and b.

It makes sense that increasing the sample size increases the performance of the algorithm. In this way, there are more policies (weight samples) evaluated and we can obtain a solved environment much faster. The downside to increasing the population size is an increase in computation size. For future research, we advise to investigate the population size further.

As we mentioned, the Parameter-exploring Policy Gradient method is not perfect however. There are ways in which we can improve the method. The performance of the Parameter-exploring Policy Gradient could be significantly improved using symmetric sampling, as described in Sehnke et al. (2010). We have not implemented it in our analysis as this was too difficult, but it could be used in future research. Besides using symmetric sampling, the initial values could be tested more extensively, in order to achieve better results.

5 Conclusion and Discussion

The Monte-Carlo Policy Gradient performed well with a γ factor of at least 0.98, this parameter could be improved by narrowing the search area around this number with small steps to get to the optimal value for this problem. The learning-rate was not changed in the experiments, for the future this could also be changed and check if this effects the speed and performance, it could be that a low γ parameter would work with a high or low learning rate. As for now this question remains open.

As we observed a high variance with Monte-Carlo Policy Gradient, we implemented the Parameter-exploring Policy Gradient algorithm. With the initialization of $N = 600$ this algorithm was able to achieve the best result and reach a reward of 200 after 30 episodes. The initializations could be researched much more extensively however to achieve better results. Furthermore, implementing symmetric sampling could improve our results.

Deep Q-learning (Mnih et al., 2013b) was tried to solve this problem, however we did not achieved to make it work to solve the lunar lander, as it was not able to land correctly. As Mao (2017) had the same problem with getting Deep Q-learning to solve the Lunar Lander, the decision was made to abandon this method and look for other algorithms to solve the problem. In the future one could try to use Deep Q learning with a bigger network, other loss functions, other activation functions or tuning the parameters so the environment can be solved.

Bibliography

M. Bednarski. Machine learning engineer nanodegree capstone project. https://github.com/mbednarski/machine-learning-nanodegree, May 2017.

Hamid Benbrahim and Judy A. Franklin. Biped dynamic walking using reinforcement learning. *Robotics and Autonomous Systems*, 22(3):283 – 302, 1997. ISSN 0921-8890. https://doi.org/https://doi.org/10.1016/S0921-8890(97)00043-2. URL http://www.sciencedirect.com/science/article/pii/S0921889097000432. Robot Learning: The New Wave.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Lei Mao. Monte-carlo policy gradient, 2017. URL https://leimao.github.io/article/REINFORCE-Policy-Gradient/.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, 2013a.

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. *Nature*, 529:533–541, 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013b.

Rémi Munos. Policy Gradient in Continuous Time. *Journal of Machine Learning Research*, 7:771–791, 2006. URL https://hal.inria.fr/inria-00117152.

Frank Sehnke, Christian Osendorfer, Thomas Rckstie, Alex Graves, Jan Peters, and Jrgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551 – 559, 2010. ISSN 0893-6080. https://doi.org/https://doi.org/10.1016/j.neunet.2009.12.004. URL http://www.sciencedirect.com/science/article/pii/S0893608009003220. The 18th International Conference on Artificial Neural Networks, ICANN 2008.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.