Machine Learning Engineer Nanodegree Capstone Project

Mateusz Bednarski

May 22, 2017

1 Introduction

Domain for my capstone project is control problems or robot motion. This requires either a physical robot or simulation engine. For purposes of this project I decided to make use of OpenAI Gym[BCP+16]. Gym is set of environments simulating various tasks. It provides ready to use simulators and frameworks for comparing algorithms. From available environments, I have selected "LunarLander-v2"[Kli17]. For a few reasons:

- For first, reinforcement learning interested me the most, so I want to get deeper into this.

- OpenAI provides leaderborads for each environement, making result comparison easy.

- OpenAI provides problem implementations, thus I can focus only on RL part.

- I love space and landing on a planet/moon is interesting problem for me.

- It has (approximately – more details in dataset section) continuous state space, so basic tabular Q-learning will not work – I need to examine more sophisticated techniques.

- Lunar lander is quite challenging environment.
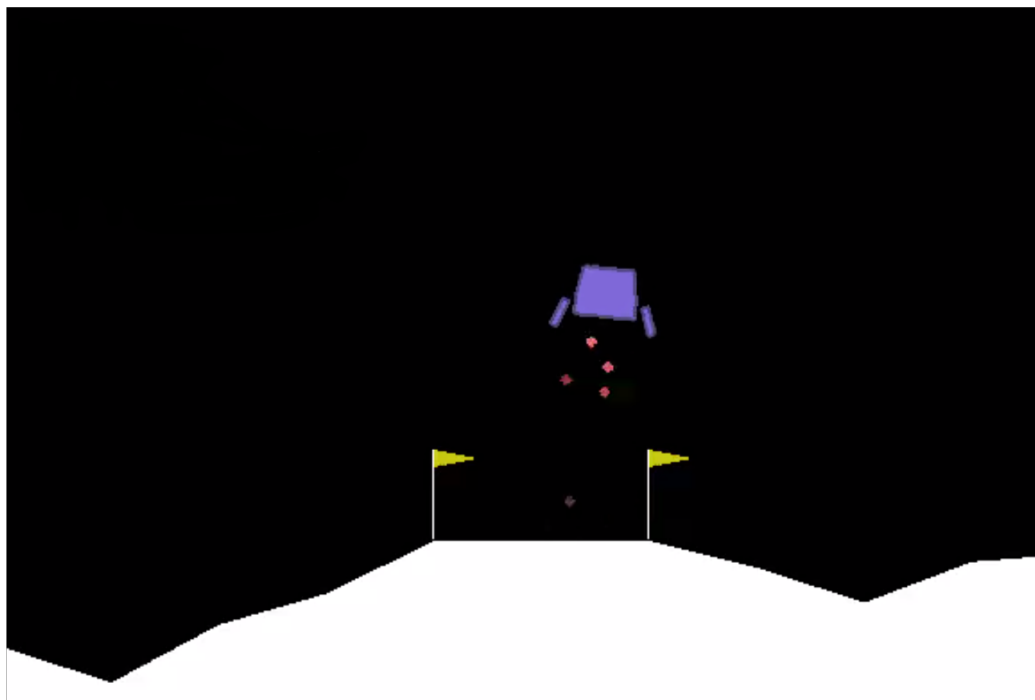
2 Problem Statement

Figure 1: Screenshot from the environment

There are a landing pad and a lander. Lander has two legs, and three engines – thrust directed down, left and right. Possible actions are one of do nothing/fire main engine/fire left engine/fire right engine. Task is to land on the landing pad. Lander starts above the pad and is affected by gravity. Simulation is finished either lander lands or crashes. Landing outside pad is also possible. Fuel is unlimited and there is no time penalty. Rewards and penalties are already defined in the environment:

- Moving towards pad with zero speed: from +100 to +140

- Crash: −100

- Successful landing: +100

- Leg with ground contact: +10

- Firing main engine (not side engine): -0.3

- When landing outside pad, an additional penalty is given.

Environment is considered as solved, when average episode reward for 100 consecutive episodes is at least 200.

2.1   Solution statement

Solution for this problem can be expressed as a policy π such that, following it will allow lander to safely touch the ground and receive as high reward as possible. In order to find such policy I will search over policy space using policy gradient approximation. Fast convergence to optimal policy is nice to have but it is not a must.

More details and definitions about policy is provided in Analysis section.

2.2   Evaluation metric

For measuring performance I will use number of episodes until solving the environment and time to reach it. This is because I suppose, that some approaches may perform more, faster episodes when others conversely - less, longer episodes.

3   Analysis

3.1   Data Exploration

As it is a reinforcement learning problem, there is no dataset understood in classical way.

However, I performed an analysis of state distribution. In selected approach this does not give much value, but if other method would be chosen (for instance space discretization or state encoding) it could be valuable.
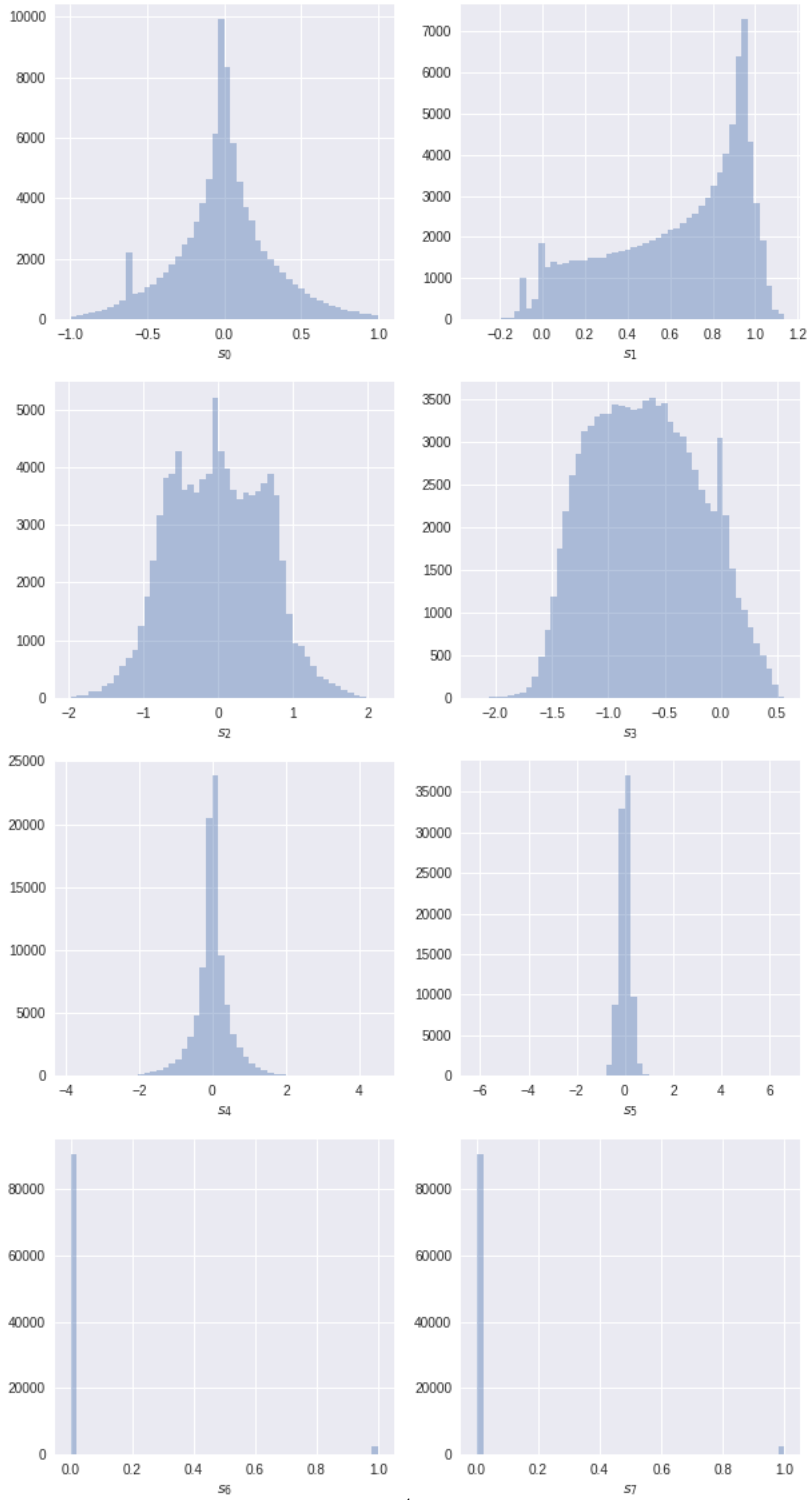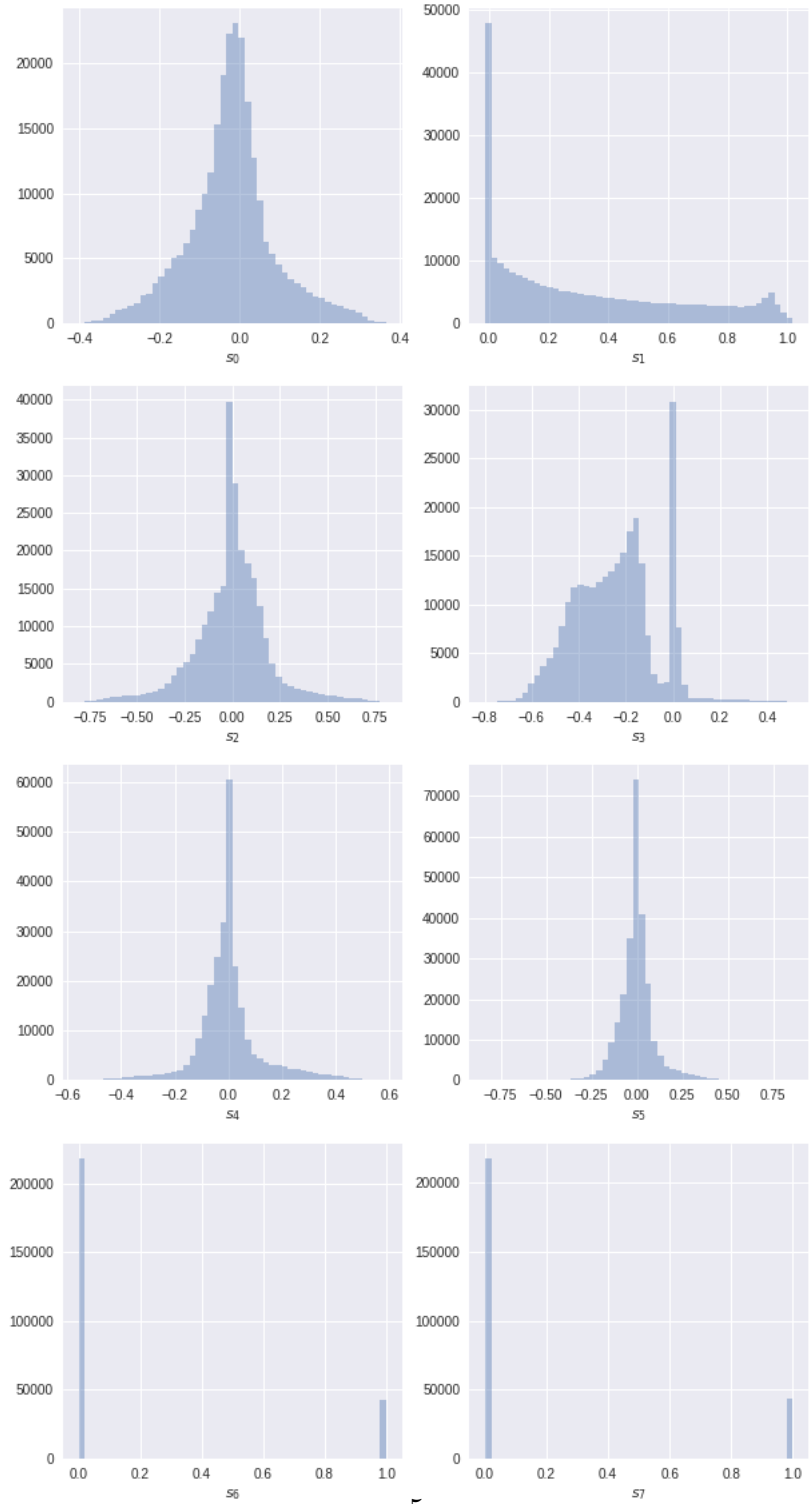
Figure 2: Random agent - 1000 episodes

5

Figure 3: Trained agent - 1000 episodes

As can be seen distribution is pretty similar. s_0, s_4, s_5 are basically the same. Trained agent produced much less variance in s_1, s_2 and s_3. Peak at 0 in s_3 is probably because initial state. s_6, s_7 corresponds to leg touching the ground. Given that, is not suprising that trained agent have value 1.0 more often.

3.1.1 Action and state space

Action space is a discrete, finite set:

$$A = 0, 1, 2, 3$$
$$|A| = 4 \tag{1}$$

State space is a vector of 8 real numbers describing lander position, velocity and orientation. Last two components says if left and right lander leg touch ground. If so, they are set to 1, instead - 0.

$$S \in \mathbb{R}^8$$
$$s_{0..5} \in (-\infty, +\infty) \tag{2}$$
$$s_{6,7} \in \{0, 1\}$$

3.1.2 Space size

Before, I said that space consists of real numbers. Given that, space state size would be uncountably infinite. However, during to machine representation of real numbers it is not exactly true. In typical implementation, *float* can handle one of 2^{52} values [1]. Size of Q-table for LunarLander is:

$$|Q| = |S \times A| = |S| \times |A| = \underbrace{\left(2^{52}\right)^6 \cdot 2^2}_{|S|} \cdot \underbrace{4}_{|A|} \approx 2.24 \cdot 10^{102} \tag{3}$$

Which requires $9 \cdot 10^{89}$ PB of memory. Also, each state should be visited enough number of times. It definetely makes this unsolvable by tabular Q-learning.

---

[1] http://stackoverflow.com/a/8875223

3.2 Algorithms

Because Q-space $(S \times A)$ is infinite in size, it is impossible to use Q lookup table. There are many approaches to deal with this problem. Basic two approaches to deal with this problem is approximation and discretization. Approximation is to replace Q table with a function $h(s, a) \to \mathbb{R}$, that will approximate $Q(s, a)$ values. Therefore, the goal is to find a function $h(s, a)$, that will return a Q-value for a given pair (s, a). Discretization combines ranges of continuous space state into discrete buckets. However, I decided to use completely different approach.

3.3 Parameter-exploring Policy Gradient

Instead of using techniques that are focused on searching function approximating q-values, Parameter-exploring Policy Gradients[SOR+10] uses *policy space*. As Ultimate goal is to find a policy that will solve the environment this algorithm search the policy space in order to find such one.

PEPG is a policy gradient method, it does mean that it estimates policy gradient and use it to maximize rewards given by following a policy.

For purposes of this project I have implemented this algorithm with slight differences.

3.3.1 Policy

For first, we define a parametrized policy π_θ

$$\pi_\theta : s \to a \tag{4}$$

For this PGPE, the policy can be basically free-form. I used two layered feed-forward neural network. (one hidden layer, and one output layer). Hidden layers contains 32 neurons. Output layer has 4 neurons - one per possible action. All layers has tanh activation function.

If we denote x as state, $\theta = (W, b, W_2, b_2)$ as policy parameters, the whole policy is following:

$$z = xW + b$$
$$\varphi = \tanh(z)$$
$$z_2 = \varphi W_2 + b_2 \qquad (5)$$
$$\varphi_2 = \tanh(z_2)$$
$$\text{selected action} = \arg\max(\varphi_2)$$

There is no need to backpropagate this network.

3.3.2 Definitions

- θ is policy parametrization (one-dimensional vector)

- P in size of parametrization θ. $\theta \in \mathbb{R}^P$

- $\mathcal{N}(\mu, \sigma^2)$ is normal distribution with mean μ and variance σ^2

- μ is current mean parametrization

- T and S are matrices with shape $P \times N$

3.3.3 Parameters

- N is a size of the population

- $\alpha_\mu, \alpha_\sigma$ are learning rates

- H is history size

- σ is initial standard deviation for policy parameters

3.3.4 Algorithm

Algorithm works in two phases. During first (training phase) in every iteration it evaluates population of N policies and updates μ and σ. For each iteration, policy parametrized by μ is evaluated separately. It does not affect learning, instead measures current performance. If for 20 consecutive trials, validation reward is greater than 200 (it is threshold for solving) we assume that parametrization μ is sufficient and we can go to test phase.

In test phase, we use learned parametrization μ in order to get 100 solved episodes. This is done for optimalization - as correct policy was found there is no need to evaluate N policies every time anymore.

Algorithm 1 PGPE for LunarLander-v2

1: initialize μ with zeros
2: test phase $\leftarrow$ false
3: **while** not solved **do**
4:     **if** test phase **then**
5:         validation score $\leftarrow$ evaluate policy(μ)
6:         continue
7:     **for** $n = 0$ to $N - 1$ **do**
8:         sample $\theta^{(n)}$ from $\mathcal{N}(\mu, \sigma^2)$
9:         $R^{(n)} \leftarrow$ evaluate policy($\theta^{(n)}$)
10:     $T_{ij} \leftarrow \theta_i^j - \mu_i$
11:     $S_{ij} \leftarrow \frac{T_{ij}^2 - \sigma_i^2}{\sigma_i}$
12:     $R^{(n)} \leftarrow R^{(n)} - b$
13:     validation score $\leftarrow$ evaluate policy(μ)
14:     b $\leftarrow$ mean reward of last H iterations
15:     **if** mean validation score for last 20 is greater than 200 **then**
16:         test phase $\leftarrow$ true
17:     $\mu \leftarrow \alpha_\mu \mu T R$
18:     $\sigma \leftarrow \alpha_\sigma \sigma S R$

3.4 Benchmark

For benchmark I have selected following evaluation from gym - `https://gym.openai.com/evaluations/eval_QOfhB9uhQ1W2hKqF8H365A` by user @pranz24. This algorithm (Covariance Matrix Adaptation Evolution Strategies) solved environment after 60199 episodes and 3 hours. I will use it to compare with outcome of my work.

4 Methodology

4.1 Data Preprocessing

No form of preprocessing was done in this project as it was not required. However, in general RL can use data preprocessing in order to represent observations in more suitable way (e.g.: tile coding, RBF coding, continuous values discretization).

4.2 Implementation

4.2.1 Parallelization

Some part of the algorithm can be easily parallelized - loop at lines 7-9. This is because each iteration in population is independent from another. Created implementation utilizes this possibility. For parallelizsm, library `joblib` was used. As a result, execution time is reduced 2-3 times.

Algorithm was implemented in two classes - `NeuralNetPolicy` and `PGPEAgent` `NeuralNetPolicy` responsibility is to select an action basing on given state and policy parametrization. Interface is following:

- Constructor takes number of features, actions and sizes for hidden layers

- `unpack` gets 1D vector and extract weights for neurons

- `get_number_of_parameters` computes total number of parameters (this is size for θ)

- `forward_pass` performs forward pass for given parametrization and state. Returns action to perform.

`PGPEAgent` implements the algorithm. Constructor takes all required parameters (environemnt to solve, α_σ, α_μ, σ, size of history and population, number of test iterations. It contains two methods.

- `run` takes maximum number of episodes to perform. it implements PGPE and basic visualization plotted in real time.
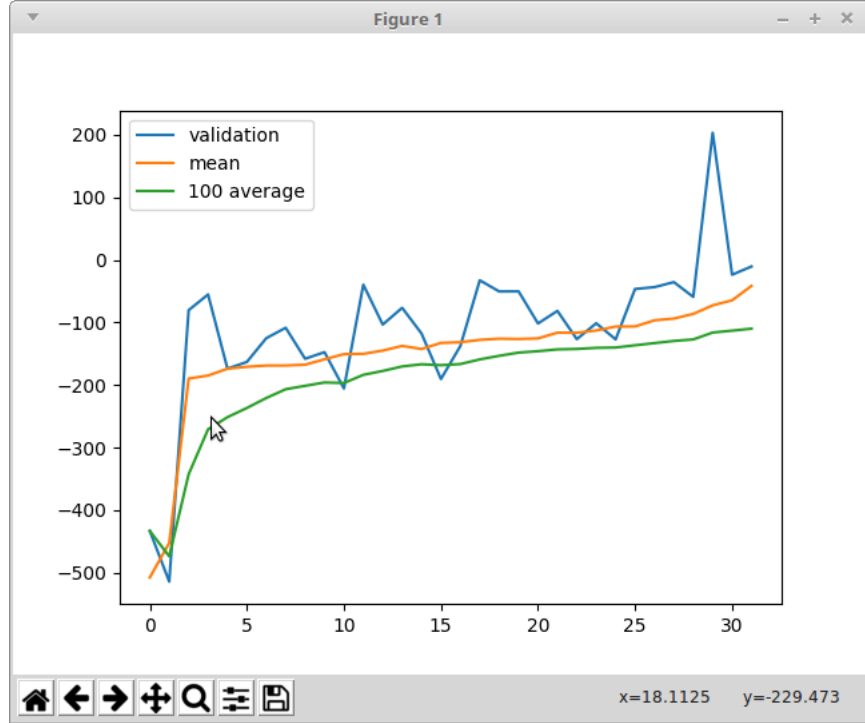

Figure 4: Visualization during training

`evaluate_policy` performs full episode using given parametrization θ. This function is not member of any class because of it will be runned in parallel - `joblib` requires such code not to be a class method.

Apart of them `Monitor` class was implemented. This class is responsible for saving duration and reward for episodes and store them into files. Next, they can be used to generate statistics and plots (`analysis.py`).

4.3 Complications and Refinement

Solution improvements came along with solving issues, so I am presenting them in a single list.

- **Policy** Initially, two layer neural network policy was used (both layers with size of 32 neurons). This approach caused slower divergence in aspect of iterations to solve (more parameters was needed to tune) and iteration duration (more computations for forward pass). Later one it turned out, that simpler policy is also capable for this problem and reduces training effort. However, with removing hidden layer completely I was not able to get solution. Learning: policy should be just as complicated as required, making it more sophisticated does not provide any benefit.

- **GPU utilization** To make process even faster I have tried to perform some computations on GPU. For this I have implemented policy forward pass using TensorFlow[AAB+15] using various scenarios (creating computation graph at once and use `tf.placeholder` to set weights, creating graphs multiple times with `tf.constant`). However, it resulted with worse execution times. I did not perform specific profiling, but I believe there were too little computations to equilibrium cost of CPU/GPU communication.

- **Another Python distribution** Another attempt to optimize the code, was to use Intel Python Distribution[2] but results were unnoticeable.

- **Floating point errors** Initially I have used much higher learning rates. Learning was faster, but I have experiencing floating point arithmetic errors - policy parameters quickly became too high causing overflow errors. This forced me to use `np.seterr('raise')` in order to be notified about those errors.

- **Random number generator** A lot of issues occurred when I tried to generate random numbers in threads. Given that, final implementation computes random numbers in main thread and send them to worker threads.

---

[2]https://software.intel.com/en-us/intel-distribution-for-python

5 Model evaluation

5.1 Measured qualities

For each test, two qualities was measured: number of iterations and total execution time (computed as sum of all iteration times). They are the same as in Evaluation metric section.

5.2 Parameters

Parameters are also same as shown before. Quick recap:

- N is population size

- $\alpha_\mu, \alpha_\sigma$ are learning rates

- H is history size

- σ is initial standard deviation for policy parameters

Most comprehensive way to perform analysis would be to run grid search over whole parameter space with few repetitions - in order to reduce fluctuations influence. However, with regard to long execution time it was impossible. Instead, I established a baseline set of parameters and measured influence of each parameter separately. The baseline parameters are following:

Parameter	baseline value
N	500
α_μ	$1 \cdot 10^{-4}$
α_σ	$1 \cdot 10^{-5}$
H	50
σ	0.5

Table 1: Baseline parameters

As there are 5 parameters, 10 configurations were created - with bigger and smaller value of each one. Parameters values for those settings are described in Table 2.

#	N	α_μ	α_σ	H	σ	time to finish [min]	iterations to finish	it/min
0	500	0.0001	0.00001	50	0.5	23.15	55	2.3
1	400	0.0001	0.00001	50	0.5	15.78	91	5.7
2	600	0.0001	0.00001	50	0.5	25.29	49	1.9
3	500	0.0002	0.00001	50	0.5	30.96	113	3.65
4	500	0.00005	0.00001	50	0.5	25.86	253	9.78
5	500	0.0001	0.00002	50	0.5	23.22	47	2.02
6	500	0.0001	0.000005	50	0.5	27.78	78	2.80
7	500	0.0001	0.00001	75	0.5	29.14	88	3.02
8	500	0.0001	0.00001	25	0.5	29.54	75	2.53
9	500	0.0001	0.00001	50	0.6	24.87	91	3.66
10	500	0.0001	0.00001	50	0.4	27.5	52	1.89

Table 2: Parameters exploration

Fist of all results show that indeed higher number of iterations does not imply longer execution time - e.g. comparing #0 with #1. After this, selected baseline looks like being near to the optimum (at least local one).

Reducing size of population (#0,1,2) causes almost twice as much iterations to perform, but they are a lot of faster. This is because in later stage of learning when models performs better also execution time increases - lander stops crashing fast. As expected increasing N makes fewer, longer iterations.

Both learning rates (#0,3,4,5,6) seems to be near optimum, because changing them makes process longer. Especially reducing α_μ.

Similar effect can be observed for H (#7,8).

Increasing initial σ(#9,10) causes big incrementation in terms of iterations and reducing - in time.

In project repository, there are plots for all of those configurations (parameter exploration directory).

In order to provide more complete and reliable analysis for each case multiple runs should be executed. Also, relations between parameters may be explored. This could be done either by hand or using GridSearch or RandomSearch from scikit-learn[PVG+11] over parameter space. However it was done because of lack of available computation time.

In fact, for all parameters model was able to solve the task.

5.3 Results

In results analysis I will use baseline parameter values. Implemented model finished learning phase after 55 iterations and approximately 23 minutes. As population size was set to 500, this gives actual number of simulation episodes $55 * 500 = 27500$. After learning phase there were 200 test runs (without learning). To sum up, agent achieved "average for 100 consecutive trials above 200" at iteration **21** after total number of episodes equal to $21 * 500 = 10500$. So here is the point when environment can be considered as solved (even if agent was learning longer).
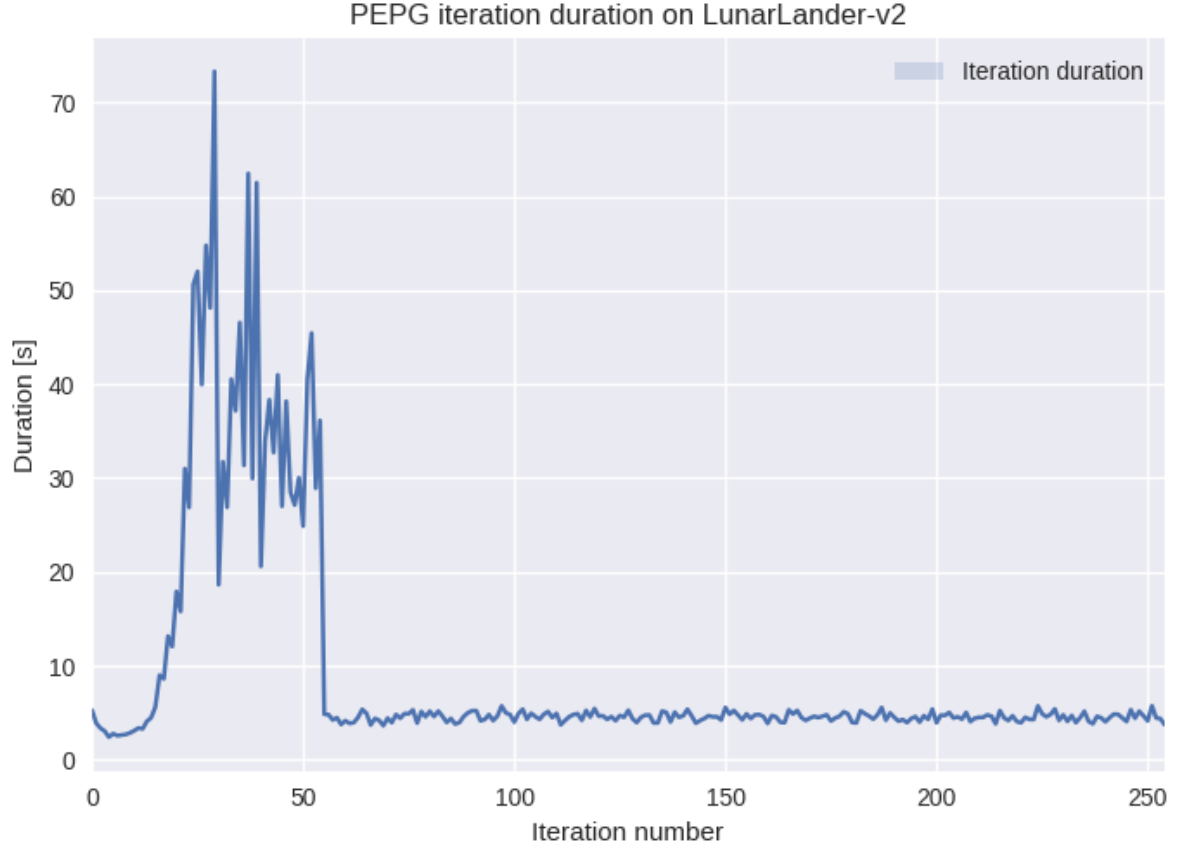

Figure 5: Agent rewards per iteration

Figure 6: Iteration durations

When it comes to iteration duration time it can be seen that during learning phase this time is a lot of longer due to fact that on testing phase there is only one episode instead of few hundreds.

Repeating model executions give similar results.

5.4 Justification

Presented model performed better than one selected in benchmark section. To recap benchmark model solved problem after 60199 episodes in about 3 hours, when presented here required only 10500 and 23 minutes.

5.5 Hardware

Hardware environment used in this project is Intel i5 3,1GHz CPU with 16GB RAM memory and Linux Mint 18.1 x64.

6 Conclusion

6.1 Free form visualization

Some visual analysis was already performed in results section. Here I will describe one interesting aspect that I found during development.

Mostly all of trials I have monitored created similar pattern. (I selected plot where it is easy to see). Just after the start, there is a sudden drop (A). Afterwards there is long phase with steady scores (B). At some point it starts growing (C) and converges near optimum (D).

For visualization I have also created a video. It is included in the repository in file **solving.mp4** WARNING: this video contains loud sound.

6.2 Reflection

To put it briefly: I have implemented the PEGP algorithm in multi-threaded version in order to solve LunarLander-v2 reinforcement learning problem. Implemented solution uses feed forward neural network with one hidden layer and does not require backpropagation. I am satisfied with achieved results. I found aspect of optimizing code very interesting even if I was not able to utilize GPU. Another interesting thing was to actually research algorithms that could help me - at some point I wanted to give up because I had feeling, that I will not be able to complete this project.

6.3 Improvement

PEPG algorithm can benefit from technique called *symmetrical sampling*. Briefly it generates random perturbations (ϵ) on current policy(μ and apply as addition and a subtraction, resulting in two populations - $\mu - \epsilon$ and $\mu + epsilon$). This makes gradient estimation more precise and reduces variance. However this makes implementation more complicated and due the fact that I am satisfied with current results I did not decide to implement this. Another improvement could be deeper parameter exploration (as mentioned earlier).
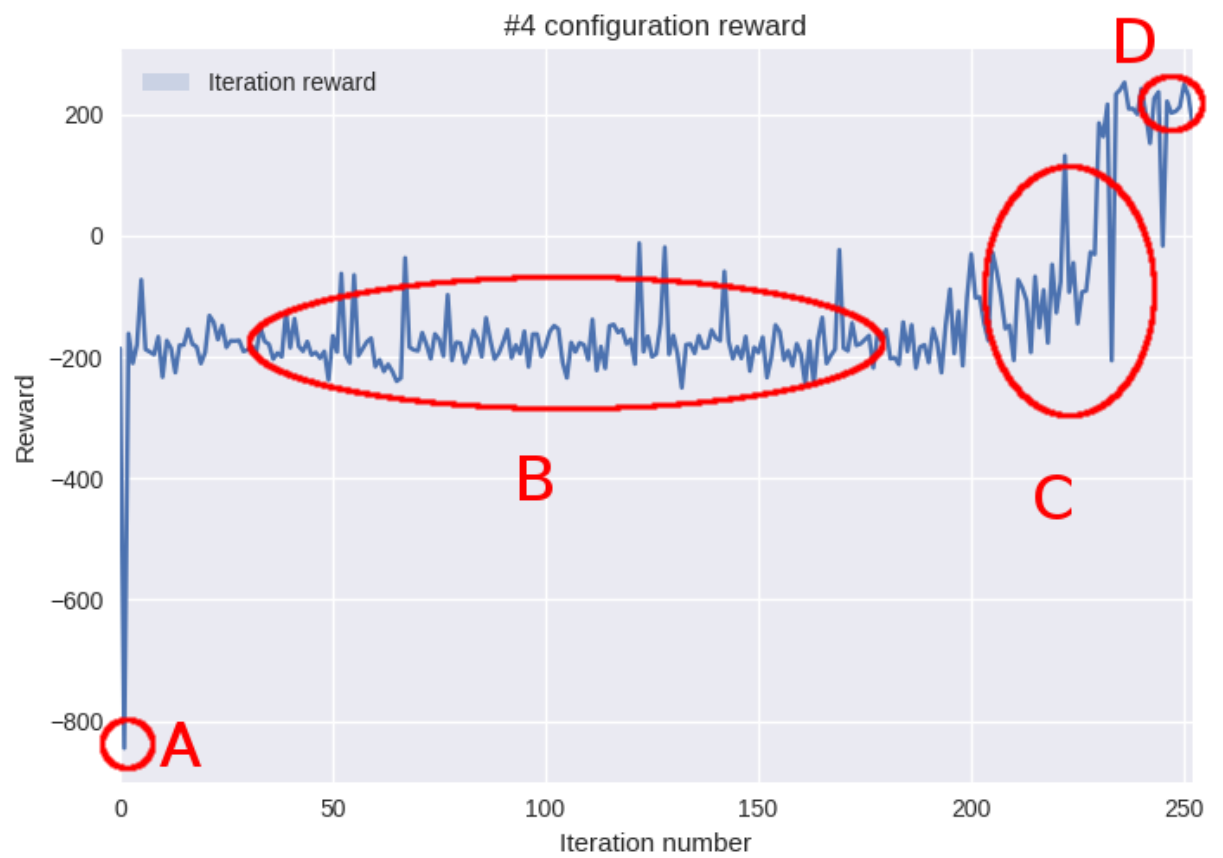
Figure 7: Reward history for configuration #4

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: http://tensorflow.org/.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. arXiv:arXiv:1606.01540.

[Kli17] Oleg Klimov. Lunarlander-v2, apr 2017. URL: https://gym.openai.com/envs/LunarLander-v2.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[SOR+10] F. Sehnke, C. Osendorfer, T. Rueckstiess, A. Graves, J. Peters, and J. Schmidhuber. Parameter-exploring policy gradients. 2010. URL: http://robot-learning.de/uploads/Publications/Neural-Networks-2010-Sehnke_[0].pdf.