CITY
UNIVERSITY OF LONDON
EST 1894

COURSEWORK COVERSHEET – GROUP SUBMISSION

Complete all details below, attach securely to your work, and submit to the relevant box outside the SMCSE Course Office (A302).

First Name:	Surname:	Student No:	Student Signature (see declaration below):
Vaida	Gulbinskaite	160006251	
Gediminas	Sadaunykas	160046802	

Programme: MSc Data Science

Module Title: Software Agents

Module Code: INM426

Assignment Name: Optimizing Deep Q learning, in OpenAI gym environment

Lecturer: Eduardo Alonso

Submission date: 12/04/2017

DECLARATION:

I confirm that the work is my own, that I have not copied the work of others, nor allowed others to copy my work and that I have referenced the work of other authors used in an appropriate way. I have also read and complied with the guidelines on plagiarism as set out in the student handbook. I understand that the university may make use of plagiarism detection software and that my work may therefore be stored on a database which is accessible to other users of the same software. I certify that the word count is correct.
Students should be aware that where plagiarism is suspected, a formal investigation will be carried out, and action may be taken under the university's rules on Academic Misconduct. This might result in penalties ranging from mark deduction to withdrawal from the university.

Lecturer/Marker to complete
Feedback to Student: **Feedback within:** ☐ **Feedback on Moodle:** ☐

If penalties were applied:

Original mark: _____

Marks deducted: _____

Reason for deduction: _____

Provisional Mark:

Optimizing Deep Q learning, in OpenAI gym environment

Gediminas Sadaunykas and Vaida Gulbinskaite – City, University of London

Table of Contents

Domain and method

Definition of domain and task

OpenAI [1] gym provides the main public environment to train reinforcement learning (RL) agents. Content of the gym, involves Atari2600 game environments, along with more recent basic human movement, and robotic hand control environments. A big breakthrough in reinforcement learning occurred with the publishing of 2013 paper, by DeepMind technologies [2] in which they used an Artificial Neural Network as a Q matrix approximator. This technique is the most beneficial when the problem is high dimensional. Their learner was able to surpass any previous approach on 6 Atari2600 environments, and achieved superhuman level on 3 of them.
In this paper, our goal is to solve the **'LunarLander-v2'** environment, using deep-Q learning. The goal of this mini game is to land the Lunar Lander on its two feet, between the two flags, after it is being dropped into the environment. There are 4 actions available: fire left, right or main engines or do nothing. The game episode ends when Lander crashes, lands or flies away from the environment coordinate system. (figure 1)

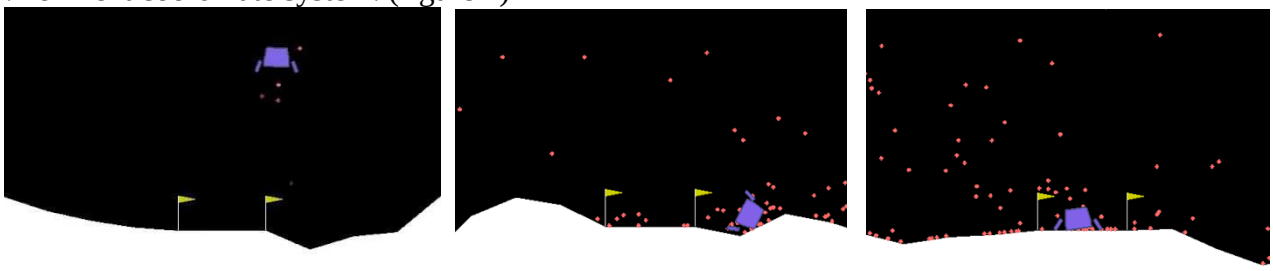


***Figure 1, 2,3**: **Left (1) -** Lander dropped into environment, firing right and main engines, to steer towards middle and reduce the falling speed. **Middle (2) -** the lander crashed outside the designated landing pad between the flags. **Right (3)** - Lander lands in the designated landing pad between two flags, touching the ground with both of its feet.*

The state space is discrete, which is viable adjustment, due to Pontryiagin's maximum principle in optimal control theory.[1] Lander earns progressively higher reward the closer it gets to the designated landing pad. It also gains points if it lands on its feet. The state matrix is defined by an array with 8 dimensions, specifying landers' position as well as coordinates in the environment.

Definition of state transition function

The Agent operates in a discrete space; therefore, the state transition function, depending on number of frames, is defined as follows:

$$s_{t+1} = f(s, a_t)$$

Where t is the number of frames, s is agent's state and a is agent's action

Definition of the reward structure

'LunarLander-v2' has a built in reward structure, which is high dimensional, and therefore could not be easily represented in a simple, tabular reward matrix. Lander incurs the cost of firing engines, 0.3 for the main, and 0.03 for the side engines per frame, simulating the fuel expense. It can crash, which is equivalent to landing in an angle different from 0, it results in a cost of 100 points. It can land away from the landing pad, which results in a reward of 100 points, with an extra 20 points for landing on both feet. Solving the game, results in 200 reward points, with an

[1] Engines are either turned on to the maximum power, or turned off completely. More on Pontryiagin's maximum principle https://en.wikipedia.org/wiki/Pontryagin%27s_maximum_principle.

extra 20 for landing on both feet. On average, reward for moving from the top of the environment to the designated landing pad and zero speed is between 100 and 140 points.
Assuming that the position of a lander is defined by $F_{i,j}$, where i is x coordinate, j is y coordinate. This could be understood, as a centre of a lander, as it is composed of more than one point. The centre of the landing pad is at coordinates $(0,0)$. Left and right landing flags, are located at the same distance z from a centre (0,0).Lander's angle is $l \in [0^{\circ}, 360^{\circ}]$; landers' velocity v_t. $\beta_t = 0.03$ is the cost of firing side engine, at frame t.[2] $\delta_t = 0.3$ is the cost of firing main engine. $q_t \in (-100, 100, 200)$ is the final reward, from crashing, landing outside the landing pad and landing inside the pad. $b_t = 10$ reward for feet ground contact. (20 for both feet landing). $A_t \in (a_{t1}, a_{t2}, a_{t3}, a_{t4})$, action space (left engine, right engine, main engine, do nothing) at frame t. Assume, ground contact is everywhere at coordinates (x,0), $x \epsilon [-inf, inf]$. Actions are mutually exclusive. Reward structure at frame t could be defined as:

$$R_t = q_t + b_t + \alpha_t + \beta_t, \text{ given } F_{i,j}$$

$$q_t = \begin{cases} 200 + b, & if\ l_t = 0^{\circ}, v_t = 0, i \in (-z \dots z), j = 0 \\ 100 + b, if\ l_t = 0^{\circ}, v_t = 0, i \in (-\infty, -z)\ to\ (z, \infty), j = 0 \\ -100 + b, if\ l_t \neq 0^{\circ}, v_t = 0, i \in (-\infty, \infty), j = 0 \end{cases}$$

$$b_t = \begin{cases} 20, if\ \ l_t = 0^{\circ}, v_t = 0, i \in (-\infty, \infty), j = 0 \\ 10, if\ l_t \neq 0^{\circ}, v_t = 0, i \in (-\infty, \infty), j = 0 \\ 0, else \end{cases}$$

$$\alpha_t = \begin{cases} -0.03, if\ A_t \in [a_{t1}, a_{t2}], i, j \in (-\infty, \infty) \\ 0, else \end{cases}$$

$$\beta_t = \begin{cases} -0.3, if\ A_t \in [a_{t3}], i, j \in (-\infty, \infty) \\ 0, else \end{cases}$$

Policy

The policy (π) could be expressed as the following: $\pi(s) = s \rightarrow a$. It determines the way agent choses what actions to perform. With each episode, our agent executes an action, which, in turn, leads to the following: change in the environment state, which either penalises the agent or rewards it. The main goal of our agent is to find the optimal policy that allows the agent to collect the highest possible reward; in our case, this would mean landing in the designated area between two flags with both of lander's feet touching the ground.

Our agent will act in accordance to ε - greedy policy. Since the lander has no prior knowledge to base its decisions on, in the very beginning it would act erratically, whilst it is trying to explore the environment and attempt to achieve the highest reward. It is important to note that it is entirely possible for our lander to achieve the highest reward at the beginning of its training, which is not an indication that the lander has learned to operate in the environment and therefore served its purpose, but rather a product of a complete coincidence. We have implemented Exploration rate (ε), which regulates how much the agent needs to explore the environment. With more episodes, agent learns more about the environment and its reward structure, in which case, continuing to explore the environment at the same rate could be counter-productive. In order to avoid this, we have implemented exploration rate decay, which, with every episode, limits agent's need for

[2] $\beta_t = sideEngineForce * 0.03$ in continuous space, but in discretized version, engines are fired either full power or shut off, therefore $sideEngineForce \in (0,1)$, analogous is main engine.

exploration and pushes the agent further to exploitation. We have also used a Minimum Exploration Rate (ε_{min}) in order to avoid agent becoming too greedy.

In short, our agent starts off with an exploration rate x, which, with the help of exploration rate decay, will decrease with every episode, allowing agent to exploit the environment more over time, until the minimum exploration rate is reached. The agent will continue to operate in the environment with ε_{min} until the end of the pre-defined number of episodes.

Initial network structure

Action-value $Q(s, a)$ function is one of the most important parts of the reinforcement learning. Majority of the real life problems are very high dimensional. Values could not be represented in a single table, therefore, a normal, tabular q-learning is often not sophisticated enough. Novel method for action-value approximation was introduced in [2]. Convolutional neural network was used to estimate the action-value function from raw pixel data. Convolutional neural network, proved to be especially powerful in computer vision. In theory, it is largely inspired by the workings of mammalian visual cortex, exploiting local spatial correlations and building in robustness to natural transformations, such as change of angle or scale. However, all types of neural networks are powerful function approximators and are well suited for feature abstraction. In this paper, we used a feedforward neural network with two hidden layers. Each Hidden layer had 30 neurons. Input is 8 dimensional (as shown in figure 4). Both hidden layers use rectified linear units (ReLU) as activation functions. ReLU slowly replaced sigmoid, and tangent functions and as of 2018 is the most popular activation function.[3] The main advantage of ReLU is the significantly reduced likelihood of vanishing gradient, which, in turn, leads to much faster learning.

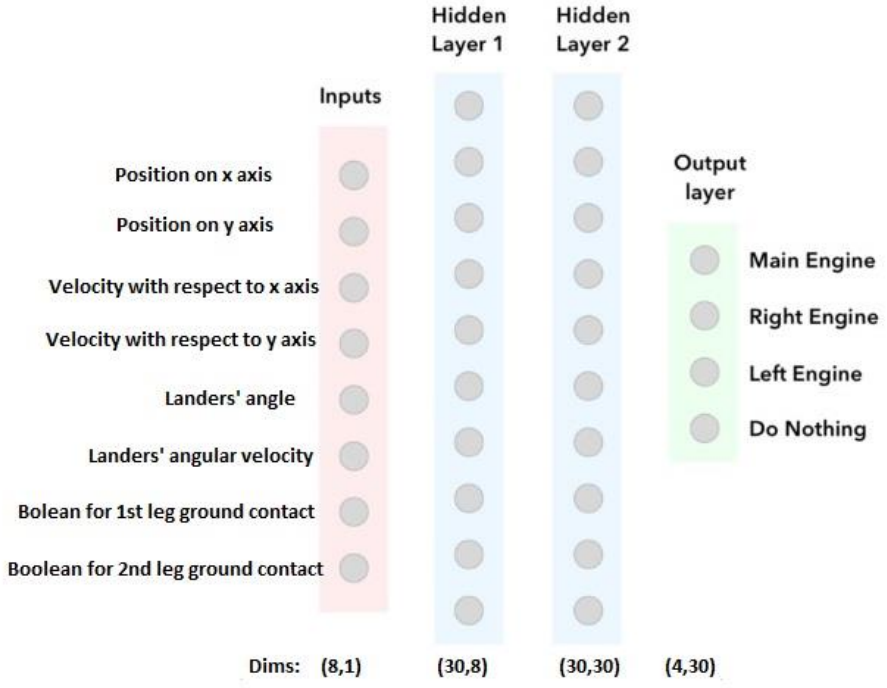


Figure 4. *Initial neural network structure for action-value approximation in LunarLander-v2 environment.*

[3] More on rectified linear units https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

Action-value function in this setting becomes $Q(s, a, \theta)$, where θ represents neural network weights. Neural network is trained using the gradient descent, optimizing loss function $L = \frac{1}{2}[r + \gamma max_a Q(s', a', \theta^-) - Q(s, a, \theta)]^2$. Here, learning rate $\boldsymbol{\alpha}$ defines the speed at which neural network weights are updated after each iteration. Steps to estimate action-value function.

1. Feedforward pass in neural network for the current state's to get predicted Q-values $Q(s, a, \theta)$ for all actions. (prediction outputs)
2. Feedforward pass in neural network for the next state s' and calculate maximum overall network outputs $max_a Q(s', a', \theta^-)$. (target outputs)
3. Set Q-value target for action a to $r + \gamma max_a Q(s', a', \theta^-)$. (from step 2) For other actions, set Q-value target same as returned from step 1, making error 0 for these actions.
4. Update weights θ using backpropagation, $\theta \coloneqq \theta + \alpha(r + \gamma max_a Q(s', a', \theta^-) - -Q(s, a, \theta))\nabla_\theta Q(s, a, \theta)$.

Definition of memory and replay

The neural network, used in the algorithm, overwrites past experiences with the new ones. In order to overcome this, we have created an array, that stores past experiences of our agent. In particular, our array stores the following information: current state, action, reward, next state, final state (boolean expression that indicates whether the state was final or not). A random sample of this array (with a defined batch size of 128 of the number of episodes) is then used to train the neural network. In order to improve agents' long time performance, we have also taken into account the future rewards. By incorporating a discount rate (γ), we allow the agent to maximise the discounted future reward, based on the current state.

Learning algorithm

Agent is trained, optimized and evaluated using python programming language. Pseudocode for training the agent using Deep-Q learning.

PSEUDOCODE FOR TRAINING DEEP-Q AGENT
Initialize agents' parameters
Initialize action-value function $Q(s, a, \theta)$ with random weights θ
Initialize target action-value function $Q(s', a', \theta^-)$ with weights $\theta^- = \theta$
Initialize replay memory D to capacity N
For episode e=1:E **do**
Initialize total_reward=0
Initialize starting environment s
    For frame t=1:T **do**
        With probability ε select random action a_t
            Otherwise select $a_t = max_a Q(s_t, a, \theta)$
        Execute action a_t and observe reward r_t and s_{t+1}
        Accumulate total_reward=+r_t
        Store experience $[s_t, a_t, r_t, s_{t+1}]$ in D
        Sample random minibatch of experiences $[s_t, a_t, r_t, s_{t+1}]$ from D
        Set target action-value function $y_t = r_t$ if episode terminates at $t + 1$
            Otherwise set $y_t = r_t + \gamma max_a Q(s_{t+1}, a_t, \theta^-)$
        Perform gradient descent on $L = \frac{1}{2}[y_t - Q(s, a, \theta)]^2$ with respect to θ and update the weights c $\theta \coloneqq \theta + \alpha(r + \gamma max_a Q(s', a', \theta^-) - Q(s, a, \theta))\nabla_\theta Q(s, a, \theta)$.
    End for
End for

Optimization and Analysis

Grid search

In order to find the best possible learning model for our software agent, we have employed a grid search for the following parameters:
Discount rate (γ)z When γ = 0, focus on immediate reward (myopic). On the contrary, when γ = 1, all rewards are equally weighted (far sighted). We have decided to look at three possible values for this parameter: 1. A small value of 0.1; 2. A medium value of 0.5; 3. A large value of 0.9

Exploration rate (ε) and its decay. When ε = 1, agent's main goal becomes exploring the environment as it disregards the sum of rewards it can collect. With ε = 0, agent becomes fixated on the reward and will not aim to explore the environment. Both approaches are valid, however at different stages of training, as at the beginning of agent's journey it needs to explore more to understand the environment and at the end of the training it needs to aim to collect as much reward as it can. We have selected the following values: 0.1, 0.5, 0.9.

We have also implemented a grid search for exploration rate decay so we can set the exploration rate high at the beginning of agent's training and decrease it over time, as agent learns about the environment more - it can pay more attention to collecting high rewards. We have selected two values: 1. a large value of 0.99 so the exploration rate decays slowly over time; 2. 0.1 a small value, so exploration rate decays fast over time; 3. a medium value 0.5.

Learning rate (α): Learning rate was used in the neural network. When $\alpha = 1$ the network learns really fast, however speed does not equal accuracy, as fast learning rate leaves the network vulnerable to overshooting and missing the global minimum, making the loss worse. When α is close to 0, the training becomes more reliable, however the learning process becomes very slow. We have selected the following: 1. 0.0001; 2. 0.01.

Neural net optimization

Neural network acts as action-value function approximator. One hidden layer is enough, to approximate virtually any function. However, scientists from Deep Mind Technologies, [2] used two hidden layers, in their convolutional neural network, to achieve human level in most of Atari2600 games. Two hidden layers were used in other successful OpenAI gym applications too. [3; 4] In our application two hidden layers are chosen as well, for a closer comparison to other OpenAI solutions. We improve our agent, using best parameters from grid-search, and optimizing two neural net parameters – activation function and number of nodes in hidden layers.[4]

Activation function: Sigmoid activation function had long been the most used activation due to several reasons: it is differentiable; non-linear; bounded (0,1) therefore well applicable to probability estimation. However, it suffers from vanishing gradient problem, where learning becomes very slow at the extremes of sigmoid activation function. A well-established standard nowadays is ReLU activation function. ReLU has several advantages over sigmoid activation function: no vanishing or exploding gradients; fast convergence; sparse neuron activation (as

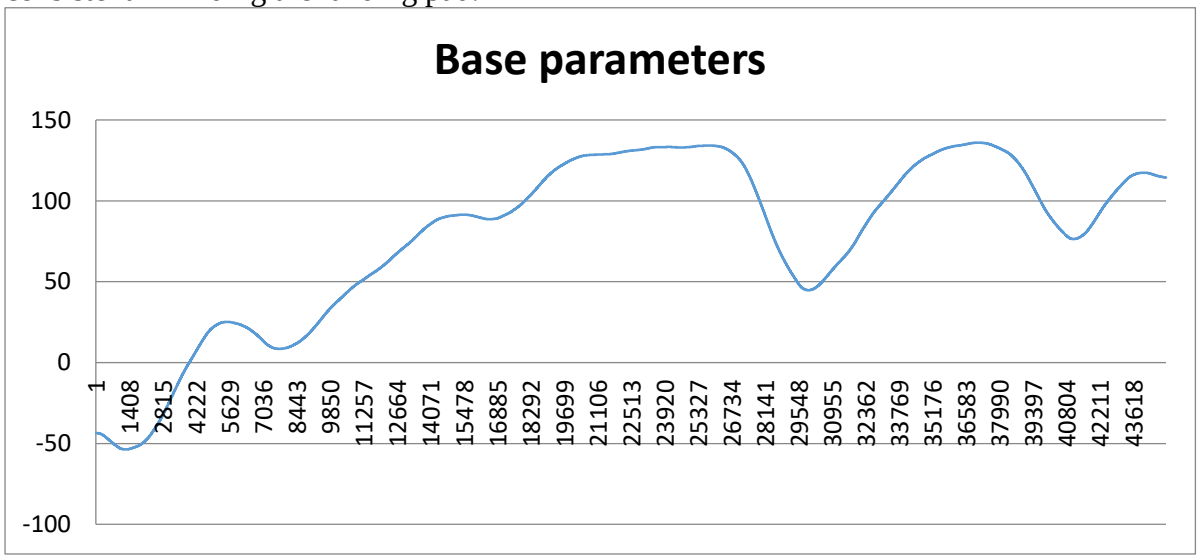
[4] Due to computational demands, and limited improvement potential, we use same number of neurons and same activation functions in both hidden layers.

opposed to dense using sigmoid). There is no free lunch, so ReLU has its own problems too. Dead neurons, which are caused by 0 gradient on the negative side of the function can negatively impact networks' capacity to learn. Leaky ReLU is an engineered solution, to dead neurons problem. At the expense of some convergence speed, it adds negative gradient to negative side of a ReLU function. As a result, neurons never completely die. All three functions are tested.

Neurons in Hidden Layers: Number of neurons in a hidden layer is responsible for networks' capacity to approximate functions. Usually, the more neurons the better network is able to fit the data, however generalizability is not necessarily improved, and speed of training is reduced. A well-established heuristic is to choose number of nodes being between number of input nodes and output. In our case it would be between 4 and 8. We chose 6, along 25 and 50 in neural network grid-search. Last two values, has been successfully applied in other OpenAi solutions: Cart Pole and Lunar Lander respectively. [3;5]

Analysis of results

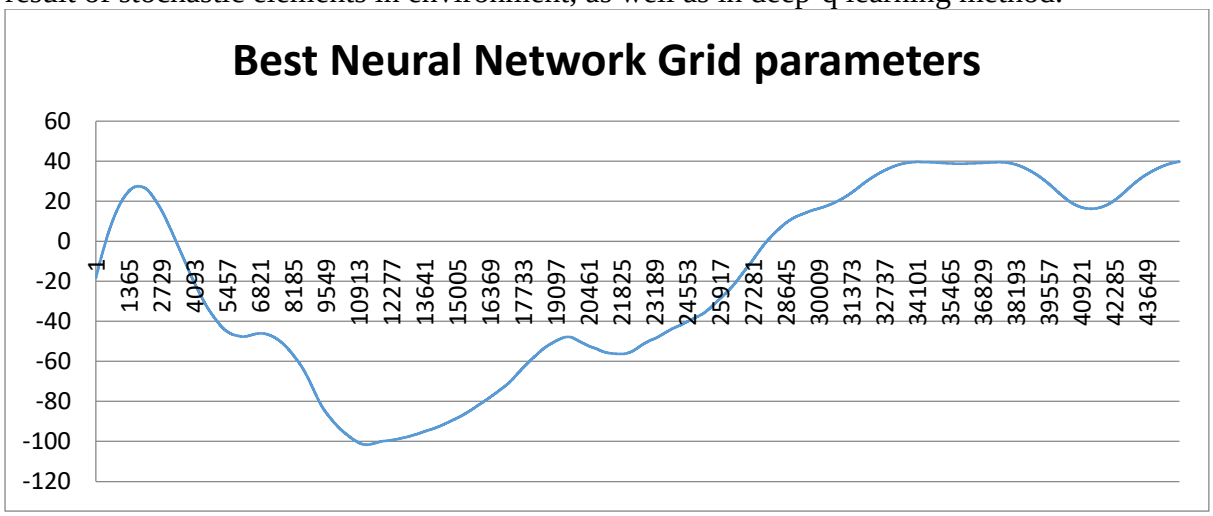
Initially we manually experimented with few parameter combinations. In Atari2600 environments, high exploration rate is usually coupled with high decay rate parameter.[3;4] It is a result of complex reward structure, which requires substantial exploration effort, before committing to actions that maximize the action-value function. Therefore, exploration rate of 0.8, and exploration decay of 0.999 was chosen. Discount and learning rates were more uncertain. However, in [6], N.Sprague showed, that high discount values (>0.9) in combination with low learning rate (<0.00025) achieved convergence in all three tested environments, while majority other combinations failed to converge, despite high number of learning episodes. Therefore, we chose discount rate of 0.99, and learning rate of 0.00025. Performance could be seen, in graph 1**.** We see a stable learning in the first 30k episodes, and then we observe a dip. It is uncertain why this dip occurred. It is known that deep-q learning does not guarantee convergence and can get stuck in local minima. In 'LunarLander-v2', it is quite common for an agent, to learn to float somewhere close to landing pad and accumulate rewards, for small distance from goal, before crashing or landing somewhere close to landing pad. In our case, some bad luck in sampling non-relevant experiences, along with still relatively high exploration rate might be the causes. However, it seems, that despite few dips agent continues to learn and towards the end of 50k episodes it is quite consistent in finding the landing pad.



Graph 1: Initial parameters *Change of scores (y axis) over the number of episodes (y axis). Smoothed using moving average with 5000 window*

Next, we performed a grid-search over the parameter space specified in section – 'Grid Search'. Results could be found in Appendix 1. We decided to take a broader view on the parameter space, in case we find some unexpected results. Grid search was performed over 20k episodes, due to computational constrains. Also, enough convergence to discriminate between combinations should be observed by then, as indicated by the base case. Overall, no convergence had been observed in most combinations. Third combination in table 1 (Appendix) produced some promising results, with unexpectedly low discount value of 0.5. In this model, our agent does not look into the future as far as in the base case, and if exploration is successful and experiences are relevant, it could result in faster convergence. However, despite being promising it still underperforms our base case, as marked in appendix 1. Grid search, therefore failed to find superior combination to our base case. No combination achieved positive average reward, while our base case produced an average of 17 reward points, indicating early convergence.

We continue optimizing our agent by taking the best parameters from grid-search, which are just our base case parameters. Next, we optimize the neural network structure, via grid-search on activation functions and number of hidden neurons as specified in section 'Neural Net optimization'. We observed a monotonic increase in performance with increase in number of nodes in hidden layers. It confirms the complex nature of action-value function which demands capacity beyond heuristic settings (6 nodes) and even that of used in CartPole solution (25 nodes). [3] Which is not surprising, as CartPole is evidently much simpler problem. Surprising result is the superiority of sigmoid activation function, when combined with 50 nodes. (table 2, appendix 1) It suggests, backpropagation does not suffer from vanishing gradient in our problem. Also, it could be that ReLU blows up activations of irrelevant nodes, as a result of prolonged phase of sub-optimal solutions. After only 10k of episodes, our agent with sigmoid activation function, 50 hidden nodes, and other parameters from base case, achieves 96.9 average reward points in the last 100 episodes**.** We run best parameters for 50k episodes. **(Graph 2)** In this scenario the learning differs from the base case. There is large negative dip in the first 25k episodes, where agent consistently accumulates negative reward. However, learning curve has stable positive slope. Interestingly, average accumulated reward value is still bellow base case. It is surprising, as this agent is engineered to be improvement upon the base case. We hypothesize, that this could be the result of stochastic elements in environment, as well as in deep-q learning method.



Graph 2: Best neural network grid parameters *Change of scores (y axis) over the number of episodes (y axis). Smoothed using moving average with 5000 window*

Conclusions and future work

In this paper we have used a Deep Q learning algorithm to train an agent in a discrete space. We have determined that it preformed best with base parameters, high exploration and exploration decay rates (0.8, 0.99); low learning rate (0.00025); high discount rate (0.9); 30 nodes in each of two hidden layers in action-value approximator, and ReLU activation functions. Understanding what agent does to land the Lunar Lander in the correct position, could be useful for modelling a real life drone or an experimental space rocket.

It would be interesting to see if the same algorithm could solve more than one discrete environment with a similar accuracy. The algorithm could be modified to apply for a continuous OpenAI Box2D environments, such as BypedalWalker-V2, CarRacing-V0 or even LunarLanderContinuos – V2. It could also be adjusted to perform more complex tasks; such as competing with another agent in a short game of basketball.

It could also be interesting to see how the algorithm performs in real life problems, e.g. simulating moose migration patterns by creating an environment that mirrors a forest or human commuting patterns by creating an environment that mirrors a city.

References

1. Chuchro, R. and Gupta, D. (2016) 'Game Playing with Deep Q-Learning using OpenAI Gym'. Available at: http://cs231n.stanford.edu/reports/2017/pdfs/616.pdf.
2. Li, Y. (2017) 'Deep Reinforcement Learning: An Overview', *arXiv preprint arXiv:1701.07274v5*, pp. 1–70. doi: 10.1007/978-3-319-56991-8_32.
3. K.Kim (2017) *Deep-Q learning with Keras and Gym*.
4. A.L.Ecoffet (2017) *Beat Atari with Deep Reinforcement Learning!*
5. Eicher, C., Saumer, H. and Chotrani, A. (no date) 'Using Deep Q-Learning with Lunar Lander'.
6. Sprague, N. (2015) 'Parameter Selection for the Deep Q-Learning Algorithm', *Rldm 2015*.

Tools

Chollet, F., *Keras,* Retrieved 05/03/2018 from https://github.com/fchollet/keras.

Code adapted from

Kim, Keon (2017). *Deep Q-Learning with Keras and Gym · Keon's Blog*. [online] Available at: https://keon.io/deep-q-learning/ [Accessed 05/03/2018].

Appendix

Table 1.

Parameter Grid Search Results							
Parameters					Statistics		
Gamma	Epsilon	Epsilon Decay	Learning Rate	Episodes	Average	Minimum	Maximum
0.1	0.1	0.99	0.01	20,000	-292.308	-1898.332114	258.1108479
0.1	0.1	0.99	0.0001	20,000	-100.469	-852.5405724	261.7529372
0.5	0.1	0.99	0.0001	20,000	-56.667	-874.0181938	261.2938695
0.9	0.1	0.99	0.0001	9,326*	-140.826	-911.8781155	245.1396837
0.9	0.9	0.5	0.01	20,000	-362.712866	-6025.060691	217.5010473
0.99	0.8	0.999	0.00025	20,000**	17.537	-1170.302725	265.0203475
0.9	0.9	0.99	0.0001	8,010*	-158.8367839	-852.2704174	207.1891758

*Kernel got restarted half way through the process

**Custom benchmark search; average taken for the first 20,000 episodes

Table 2.

Neural Network Grid Search Results					
Parameters			**Statistics**		
Number of neurons	**Function**	**Episodes**	**Average (last 100)**	**Minimum**	**Maximum**
6	Leaky Relu	10,000	-36.8140	-629.8552	276.3808
6	Relu	10,000	41.4515	-620.9514	258.4521
6	Sigmoid	10,000	-77.6900	-538.1112	242.3449
25	Leaky Relu	10,000	-59.3128	-874.3156	263.7785
25	Relu	10,000	60.5751	-657.2833	274.3964
25	Sigmoid	10,000	-79.0265	-527.2400	256.2395
50	Leaky Relu	10,000	-77.1950	-661.0669	232.5783
50	Relu	10,000	-113.1930	-567.7881	249.0507
50	Sigmoid	10,000	96.8556	-523.4790	276.3227