A Deep Q Learning Experiment

On the mechanics of Landing a module on Moon

Manikanta Reddy Dornala
CS 7642 - Reinforcement Learning, Georgia Institute of Technology

In this experiment we implement a neural network based agent to optimize a future reward function for a sequential decision problem and explore the numerous intricacies involved in the problem. We present an agent that successfully learns to consistently steer towards the goal in a simulation of a module landing on the moon built upon OpenAI Gym[1]. We incorporate ideas from multiple previous works and adapt them to the specific problem.

1 INTRODUCTION

Reinforcement learning enables control agents to learn different strategies to interact with the environment through a sequence of observation, decisions, and reflections. The goal of any such agent is to observe the current state of the environment and take an action that maximizes the expected cumulative future reward. The mathematical representation of this statement is given by the Bellman equation for Q-value as

$$Q^*(s,a) = \sum_{s'} T(s,a,s')(R(s,a) + \gamma \max_{a'} Q^*(s',a')) \tag{1}$$

The optimal policy or the optimal decision function can then be obtained by maximizing the Q-value function.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a) \tag{2}$$

The Q-value can be approximated by the Q-Learning[6] algorithm that iteratively updates Q-values in a Q-table which converge to true optimal values over time.

Algorithm 1: Off Policy Q Learning

1 **Initialize:** Q(s,a), $\pi(s)$ and an initial state s
2 **while** *policy not converged* **do**
3 $a = \pi(s)$
4 Take action a and then observe new state s'
5 $y = R(s,a) + \gamma \max_{a'} Q(s',a')$
6 $Q(s,a) = Q(s,a) + \alpha(y - Q(s,a))$
7 $\pi(s) = \operatorname{argmax}_a Q(s,a)$, s = s'
8 **end**

Note that update for a time(t) depends on the observation from time(t+1) ($\max_{a'} Q(s',a')$). Δ at any time(t) doesn't depend on the the action to be taken at time($t+1$), which will be dependent on the policy(π) at $t+1$. Hence the term **Off** Policy. (appendix A)

The above algorithm has a high chance of being stuck with a sub-optimal policy when the world is non-deterministic (most, if not all problems). This is because we may never visit some of the states ever. It is in some sense a very safe agent that never takes risks and accepts its fate when the universe plays dice.

2 ϵ GREEDY *ACTION SELECTION* POLICY

It is imperative that the agent venture beyond its known experiences and *explore*. Every now and then it should take actions against the optimal policy learned based on the experiences so far. At the same time, place safe bets by *exploiting* its knowledge. The ϵ greedy policy suggests that the agent choose a random action from the action space,$\mathcal{A}$, with a probability ϵ and the optimal action with a probability $1-\epsilon$ during the *action-selection* phase. This has to be performed instead in Line 3 in Algorithm 1.

$$a = \begin{cases} random(\mathcal{A}ction) & \text{with probability } \epsilon \\ \max_{a'} Q(s',a') & \text{otherwise} \end{cases} \tag{3}$$

ϵ can be a constant or some variable function. Constant value can lead to arrogant behavior, hence in practice it is good to use a *time-decaying* ϵ function.

3 Q-LEARNING IN CONTINUOUS STATE SPACES

Defining state transitions based on actions taken and computing the transition probabilities works well in discrete worlds. Regular problems are too large for us to keep track of all state-action pairs. Instead, we approximate Q values with a function $Q(s,a,\Theta)$, where Θ is the set of parameters that defines the function approximation.

We consider the problem where the state space is continuous but the actions are discrete and finite (n in number). In such case the Q function (or the **action-value** function) takes as input an action(a) (from a discrete set) and a state(s) (comprising of continuous values). We can look at this in a different way as having n different functions each giving out a Q-value for the state(s), such that the function that produces the maximum Q-value is the optimal action at that state.

In such case, the Off policy ϵ greedy Q learning algorithm remains the same except for the update step at 6 in Algorithm 1 by a different update rule.

One of the updates, which will find *a minimum*, that can be iteratively performed on the parameters Θ is via a loss function $\mathcal{L}(\Theta)$ given by

$$\begin{aligned} Y_t^Q &= R_{t+1} + \gamma \max_a Q(s_{t+1}, a, \Theta_t) \\ \mathcal{L}(\Theta_t) &\propto \mathop{\mathbb{E}}_{s_t,a_t} (Y_t^Q - Q(s_t,a_t,\Theta_t))^2 \\ \nabla(\mathcal{L}(\Theta_t)) &= (Y_t^Q - Q(s_t,a_t,\Theta_t))\nabla_{\Theta_t} Q(s_t,a_t,\Theta_t) \\ \Theta_{t+1} &= \Theta_t + \alpha(\nabla(\mathcal{L}(\Theta_t)) \end{aligned} \tag{4}$$

Neural networks are a kind of function estimators that are driven by gradient descent in general.

4 DEEP Q LEARNING[3]

We construct a neural network with weights(Θ), input(s) and n output nodes, each output node corresponding to Q-value of taking an action i for the input state(s).

But instead of driving the update rule on the values from the previous iteration, we pick random (action-values, state) pairs from a *sample space* as there is a lot of correlation between continuous states, which could lead to over-fitting and divergence. The sample-space is a random sampling of experiences from the agent's memory, collected over many episodes. This technique is termed ***Experience Replay***.

The agent thus learns from past experiences in general instead of immediate experiences and is expected to reach a global minimum.

Algorithm 2: ϵ greedy Off Policy Deep Q Learning

Initialize: Neuralnet weights Θ , an initial state s and a fifo queue buffer $\mathcal{B}$ of finite size
for *episode = 1,E* **do**
  for *time = 1,T* **do**
    $a_t = \begin{cases} random(\mathcal{A}ction) & \text{with probability } \epsilon \\ \max_a Q(s_t, a, \Theta_t) & \text{otherwise} \end{cases}$
    Take action a_t and then observe new state s'_t and reward received r_t
    Enque (s_t, a_t, r_t, s'_t) in $\mathcal{B}$
    Sample a minibatch $\mathfrak{b}$ observations, $\{(ss_j, sa_j, sr_j, ss'_j) | j \text{ in } 1, \mathfrak{b}\}$ from $\mathcal{B}$
    for *j in 1,*$\mathfrak{b}$ **do**
      $\delta = \max_{sa'} Q(ss_{j+1}, sa', \Theta_t)$
      $y_j = \begin{cases} sr_j & \text{if } ss_{j+1} \text{ is terminal} \\ sr_j + \gamma\delta & \text{otherwise} \end{cases}$
    end
    Update Θ using any loss($\mathcal{L}(\Theta)$) optimization algorithm over $(s_j, y_j) \forall j \in 1, \mathfrak{b}$. (appendix B)
  end
end

We'll apply this learning agent on the Lunar Lander simulation

5 LUNAR LANDER

Lunar lander is an environment available in OpenAI gym. It simulates the landing of a module on the surface of moon. Fig. 1

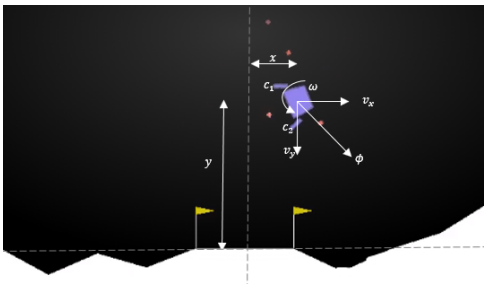


Figure 1: Lunar lander with state vectors

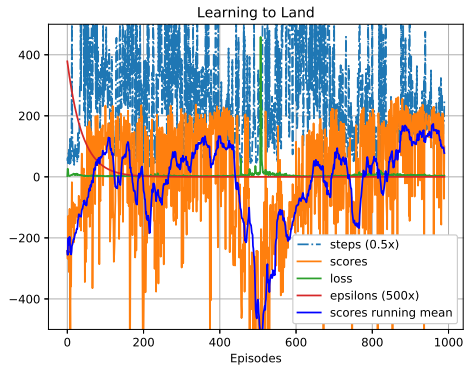
The state of the lander at any given moment of time is defined by 8 *continuous* variables. x, y, v_x, v_y, ϕ orientation , ω angular speed , $c_1$ground contact of leg1 , c_2 ground contact of leg2.

It can perform four *discrete* actions, *do nothing*, *fire right*, *fire left*, *fire main*. Every fire of main engine is rewarded −0.3. A Crash −100. Come to rest 100. Land on the pad between 100…140. The environment stops beyond 1000 time frames. So the worst possible scenario leads to a reward aleast −240.

This particular discrete action−continuous state problem can be solved by a simple PID controller but it involves a fair bit of physical understanding of action space and assumptions about the environment (appendix C).

On the other hand, Q-Learning needs no prior knowledge and no interpretation of the transition. $s \xrightarrow[a]{r} s'$.

The neural network implemented is a multilayer perceptron with 3 hidden layers. Multiple configurations of ϵ, $\mathfrak{b}$ and other params have been experimented and most of them produce similar results as below.



The agent continually learns to produce a reward slightly less than 100. Notice that most of the runs in training are governed by 1000 runs episodes. Which means the lander didn't manage to crash land but began to hover in mid-air. This is the case of converging to a local minimum, where the lander avoids huge punishment from crashing and spends on fuel for hovering.

6 DOUBLE Q LEARNING[5]

Vanilla Deep Q Learning is very optimistic, it tries to avoid great punishments and accepts anyone who offers a morsel of positive future reward and converge to a suboptimal policy.

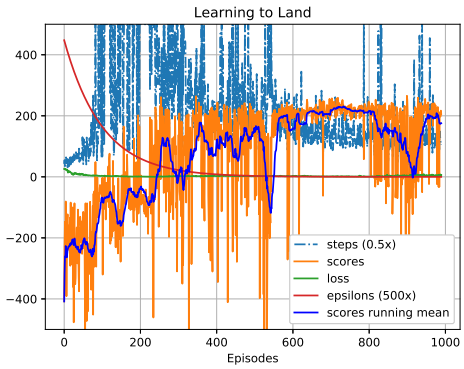
Due to the nature of max operator DQN also leads to overestimations in values which results in propagation of wrong information about the information of which states are more valuable directly affecting the quality of policy. The max operator ignores underestimations of Q values but honors a Q value that has been wrongly overestimated when selecting the optimal policy.

Both of these can be prevented by introducing a second neural network in the agent. While the first model learns the optimal policy the other model decides the current optimal policy. The second model is updated with the new parameters from the first one every now and then.

$$\begin{aligned} Y_t^Q &= R_{t+1} + \gamma \max_a Q(s_{t+1}, a, \Theta') \\ \Theta_{t+1} &= \Theta_t + \alpha(\nabla(\mathcal{L}(\Theta_t)) \\ \Theta' &= \Theta \text{ Every now and then} \end{aligned} \tag{5}$$

If we keep the hyper parameters of vanilla dqn 4 the same and introduce a second network that is updated every ($n = 1000$) observations we achieve a considerable improvement in performance averaging about a per episode score of 170. We are close!

But notice how there are still a lot of episodes that take the 1000 step runs. The lander still aims to receive a less negative reward by avoiding the ground.

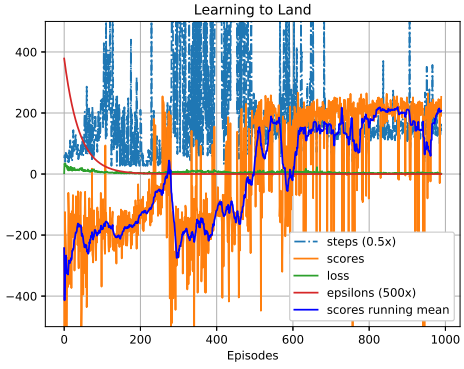


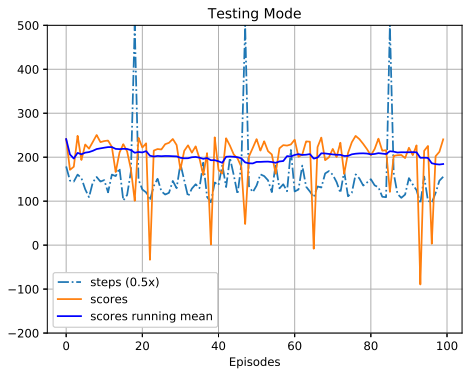
7 PREFERENTIAL MEMORY

Landing on the ground is a particular set of sequences that when happen to lead to a huge positive reward. By our exploration strategy, the landing event happens enough times that the lander knows there is a positive reward state but is not *confident* enough that it takes that course of action over and over.

To increase the confidence in events leading to landing, we increase the chance of such (state, action, reward, next state) tuples from being sampled during the model update step. Whenever an action leads to a positive reward, we re-save the same tuple multiple(a constant τ) times. This is slightly different from the method prescribed in *Prioritized Experience Replay*[4] where a probability weight is assigned to experiences in memory based on the TD error term.

The average test score is now about $\sim$ 199, with the same setting in other hyperparameters as in 6

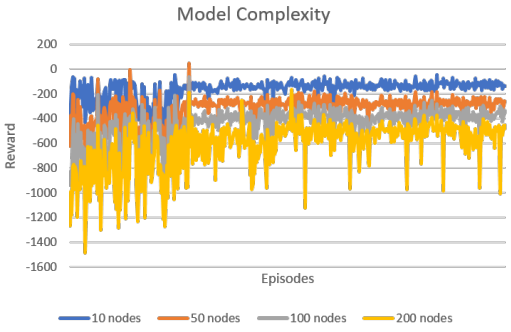




8 ON MODELS AND HYPER PARAMETERS

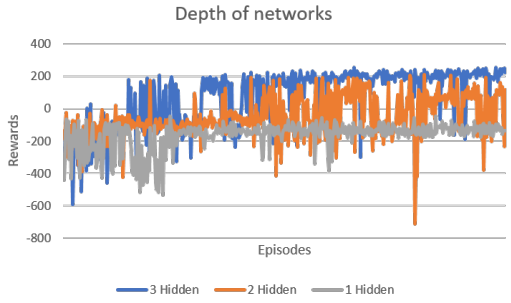
8.1 Model Size and Depth

The neural network comprises of nodes grouped into layers. In general, we measure the size of a model by the number of nodes it has and the depth of model by the number of layers.



In general, on increasing the size of a model decreases the rewards obtained. As this lead to more parameters and overgeneralization of the problem.

On the other hand, if we fix a number of nodes(= 100) and build a deep network, the reward increases!

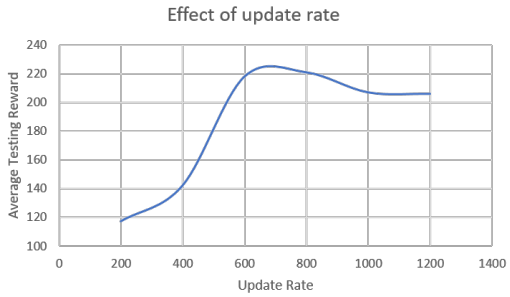


Theoretically, a neural network with only 1 hidden layer should be able to approximate any function with enough nodes. But a deep network is, capable of learning abstract concepts in the data with a few nodes. When a back-propagation update occurs, the weights of the shallow network simply adjust to memorize the input.

8.2 Updating the Second Model

As we employed Double Deep Q learning, we need to update the target model every n observations. This is expected to bring down

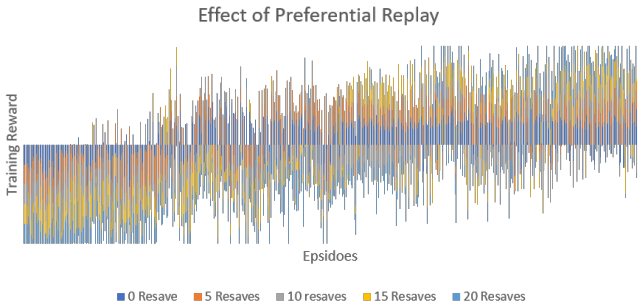
over-fitting. We define update rate as observation interval between successive updates of the target model.



As we can see small update rates tend to give smaller rewards as they can be overestimated by the same model. This was expected as small update rate would mean that target and model are temporally similar. After achieving a maximum, increasing the update rate will bring the reward down as in limit a large update rate would mean no update at all. For our model, an update rate around 800 is in the Goldilocks zone.

8.3 Preferential Memory Re-save

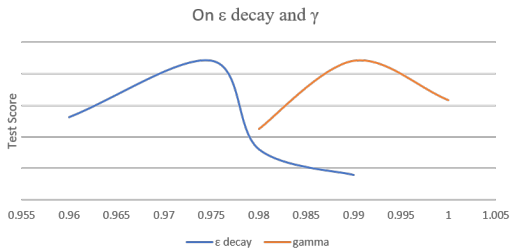
We mentioned how we save positively rewarding experiences multiple times. This approach(τ resave) increases preference by a direct increase in population.



As can be seen, increasing how frequently good experiences are sampled in general increases the reward. Notice that towards the end, the difference between multiple preferential re-save rates begins to blur. (Unarguably 0 re-saves still lags behind).

This is because, as more and more samples are drawn from a population dominated by good observations, and the epsilon for action selection goes down, the agent will be taking good actions anyway. That means either we re-save or not the agent's memory is going to be saturated with good experiences. The greatest effect of preferential re-saves is to speed up the convergence towards good actions. 0 re-saves will eventually get towards it through more iterations.

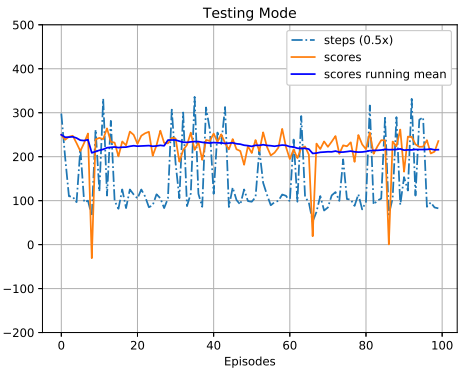
8.4 γ and ϵ decay



There are sweet-spots for both γ (reward decay) and ϵ decay to produce high rewards. They are essentially attributed to being the balance between relying too much on expected future rewards and exploring just enough to find the rewarding states.

9 CONCLUSION

The best model is three layers deep, with 100 nodes, 10 re-saves, 800 target-update rate, 0.975 ϵ decay and 0.99 γ. This agent consistently provided a score > 200 averaging around ~ 221. The behavior of the landing trajectory is strikingly similar to one produced by a PID controller(appendix C), which is based on the physics and understanding of the system. It is quite remarkable that the agent learns the ways of the world and its mathematics without any prior knowledge of the same.



10 APPENDIX

A OFF POLICY VS ON POLICY UPDATES

Off-policy learners compute one policy while following another. They can learn the optimal policy regardless of the behavior of the current policy. Whereas On Policy learners believe their current policy is good enough and try to improve upon it. Both stem from the temporal difference update given by

$$Q(s_{t+1}, a_{t+1}) = Q(s_t, a_t) + \alpha(y_t - Q(s_t, a_t)) \tag{6}$$

The difference between both learners is the update(y)

$$\begin{aligned} \text{SARSA (On Policy)} &: y_t = r_t + \gamma Q(s_t, a_t) \\ \text{Q-Learning (Off Policy)} &: y_t = r_t + \gamma \max_{a'} Q(s_t, a') \end{aligned} \tag{7}$$

The *max* operator in Q-Learning makes the update step, current action agnostic. However the same *max* can cause the neural network to diverge due to its non linear nature.

On policy, methods are stable and converge but might be trapped in locally optimal policies.

B OBJECTIVE OPTIMIZATION

Optimization algorithms *optimize* (*minimize* or *maximize*) a *Target Objective* function[8]. Target functions in our cases are also labeled as *Error*, *Loss* or *Cost* functions $\mathcal{L}(\Theta)$. The goal of the optimization algorithm is to find a function $y = \mathcal{F}(x, \Theta)$ such that the value of $\mathcal{L}(x, y, \Theta) \forall x$ is *optimized* in the space of x, y.

Optimization can be done by solving for $\mathcal{F}$ with constraints defined by $\mathcal{L}$ in a space $\mathcal{D}$. The solution is obtained by solving simultaneous equations generated by $\nabla\mathcal{L} = 0$ in $\mathcal{D}$. For example,

$$\begin{aligned} \text{Let} &: (x, y) \in \mathcal{D} = \{(1.01, 1.99), (2, 4), (3.99, 8.01)\} \\ \mathcal{F}(x, \Theta) &: y = \theta x \\ \mathcal{L}(x, y, \Theta) &: e = (y - \mathcal{F}(x, \Theta))^2 \\ \text{Goal} &: \text{Minimize } \mathcal{L} \\ \text{Solution} &: \theta = 2 \end{aligned} \tag{8}$$

It is hard to solve directly when the dimensionality of the parameter vector Θ and available space $\mathcal{D}$ is huge. Instead, we apply iterative algorithms. Equation 4 is one such algorithm called *Stochastic Gradient Descent*.

In the neural network built for solving Lunar Lander, we use the *Adam*[2] optimizer. It is known to perform well when the parameter space and data space are large.

C PID CONTROLLER[9] SOLUTION

Although reinforcement learning shows promise in learning to land the module on the moon. The caveat here is the tremendous amount of time and computing power it took to build and empower the agent to steer it correctly. With better knowledge and a precise implementation, we can beat the game. This method has nothing to do with Reinforcement Learning and we slide it in from learnings of a different course(AI for Robotics). Please note that this procedure relies heavily on the knowledge about the system and environment. That implies we know which action does what.

In control systems, the agent is expected to do something along a pre-estimated function. The action you take is to minimize a utility function ($u(t)$). (same old story).

$$u(t) = \tau_p e(t) + \tau_i \int_0^t e(\lambda) d\lambda + \tau_d \frac{de(t)}{dt} \tag{9}$$

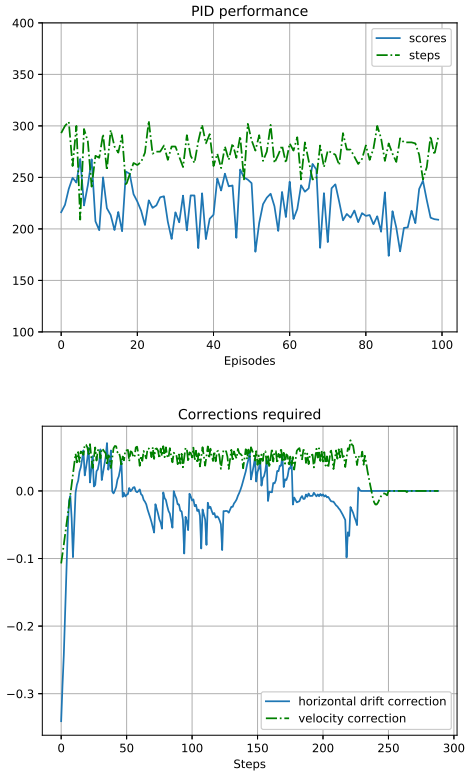
The trick in here lies in defining $e(t)$ for our problem. We define two functions instead of one. The first to correct horizontal drift and the second to correct the velocity on touchdown at unit time steps.

$$\begin{aligned} u_{v_y}(t) &= \tau_p v_y + \tau_i \int_0^t v_y dt + \tau_d \frac{dv_y}{dt} \\ &= \tau_p v_y + \tau_i y + (ignoring) \\ u_\phi(t) &= \tau_p(x + v_x - \phi) + \tau_i \int_0^t (x + v_x - \phi) dt + \tau_d \frac{d(x + v_x - \phi)}{dt} \\ &= \tau_p(x + v_x * 1 - \phi) + (ignoring) + \tau_d(v_x - \omega) \end{aligned} \tag{10}$$

For sake of brevity, we consider only terms that will be available. Integral of deviation and Derivative of velocity aren't directly available through state, so we ignore them.

Since we can take only one action at any given time, we do nothing when $|u_{v_y}|, |u_\phi| < \eta$, fire main engine if $|u_{v_y}| > |u_\phi|$, right engine when $u_\phi > \eta$ and left engine when $u_\phi < -\eta$.

The parameters are generally tuned using a *coordinate descent*[7] algorithm (*Twiddle*)(yet an other objective optimization algorithm), but for this case it was easy enough to handpick producing a 222 mean score!!!



As mentioned earlier, the PID controller generated trajectory is strikingly similar to the one produced by the best RL agent.

D REFERENCES

REFERENCES

[1] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[2] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

[4] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *CoRR*, abs/1511.05952.

[5] van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461.

[6] Watkins, C. J. C. H. and Dayan, P. Q-learning.

[7] Wikipedia. Coordinate descent.

[8] Wikipedia. Mathematical optimization.

[9] Wikipedia. PID controller.