TESTING RECENT REINFORCEMENT LEARNING TECHNIQUES IN CLASSIC ENVIRONMENTS

Final Assignment
Neural Networks
Leiden University

ABSTRACT

The past couple of years have brought about many Reinforcement Learning advances. Their usefulness has traditionally been measured on the Atari suite. This project describes the Q-Learning and Deep Q-Network algorithms and their recent extensions, along with accompanying techniques. Afterwards, they are applied in two classic environments, and their behavior is analyzed. After hyper-parameter tuning, the best-performing algorithms solve the Cart Pole and Lunar Lander environments in 10 seconds and 25 minutes, respectively.

1 INTRODUCTION

Reinforcement Learning (RL), and in particular Deep Q-Networks (DQN) have seen a lot of success lately. Many extensions have been developed, modifying some aspect of the algorithm to increase performance. Their combined behavior has not been studied extensively, and even then, only on a limited number of settings.

In this project, we set out to measure the performance of both simple and complex RL algorithms, together with their extensions in two classic environments. Although the settings are not the most difficult possible, successful application and understanding of the learning techniques is what guides us.

The report is structured as follows: Section 2 provides a very short description of general RL concepts as well as the context which sparked the idea of this project. Section 3 presents the problems in which RL algorithms will have their performance measured. Concrete learning algorithms and recent extensions pertaining to them are presented in Section 4. Other techniques complementary to the learning process are described in Section 5. The application, observed behavior, rationale, and results are presented in Section 6. Finally, section 7 gives an overall summary of the results and an overarching conclusion.

2 BACKGROUND

This section presents Reinforcement Learning concepts and context on which next sections built upon.

Reinforcement Learning Along with Supervised and Unsupervised Learning, RL is the third main paradigm in Machine Learning. It concerns itself with finding the best strategy for maximizing cumulative reward in a sequential decision-making problem. The most distinctive trait is that a RL algorithm is in charge not only of learning from observations of the environment, but also steering the way new experience flows in. Figure 1 shows the the RL interaction flow. It is composed of an *agent* that selects an action a (from the action space $\mathcal{A}$) based on the observed state s (from state space $\mathcal{S}$) of the *environment*. It then adjusts itself based on the received reward $r \in \mathbb{R}$. This process is repeated until the environment reaches a *terminal* state s and the *episode* ends.

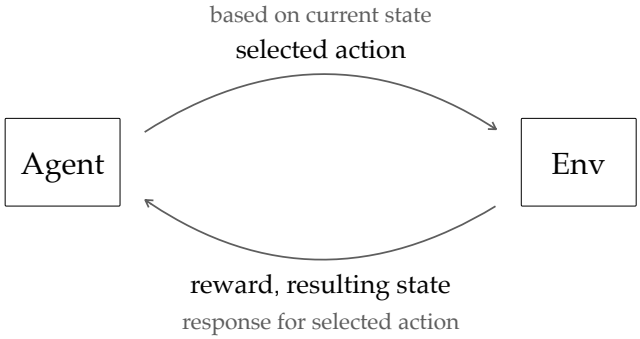


Figure 1: Reinforcement Learning process

Testbeds One common benchmark for RL algorithms efficiency is the Atari environment (described in detail by Mnih et al. (2013)), in which the agent learns to play arcade video games. This environment is particularly difficult because the agent is fed nothing more than pixel values, and has to adapt in order to obtain a good score in lieu of the common knowledge humans possess.

Measuring Performance In the Atari environment, Hessel et al. (2017) showed that combining recent techniques leads to better performance than applying any individual one. Human performance is surpassed on most games, and algorithm extensions help obtain more than double the performance of the vanilla algorithm. The same team, Horgan et al. (2018), has since shown that through engineering effort and tremendous computational resources, another significant jump in performance can be achieved.

In contrast, in this project, we measure the performance some RL techniques (described in Section 4), in a simpler environment (described in Section 3).

3 ENVIRONMENTS

This section presents the two environments used as a benchmark in the rest of the report. They are both hosted in OpenAI's Gym [Brockman et al. (2016)] framework.

For each environment rules, mechanics and objective are described. Afterward, the formal RL components are detailed: the *state* – information the agent has about the world, characterizing the world at any given step; *actions* – what the algorithm can pick from, based on the current state; and *reward* – how the environment responds to the taken action. Finally, we describe what causes an episode to end and when the environment is considered solved.

3.1 CART POLE

This is one of the simplest environments available. The same way one checks matrix dimensions as a simple requirement for correctness so does a RL practitioner sanity-check its implementation on the Cart Pole environment. This environment was first described by Sutton et al. (2016).

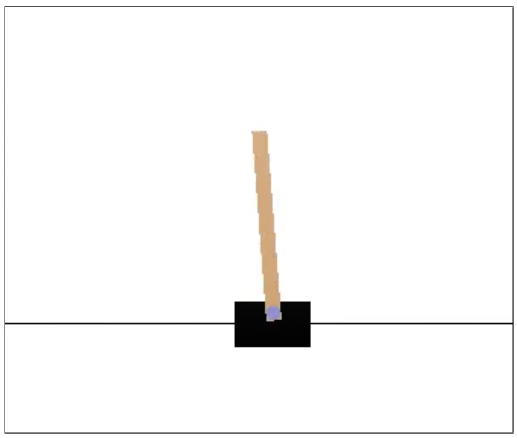


Figure 2: Example frame of the Cart Pole environment. The pole is currently leaning to the left. Moving the base to the left is required in order to bring it to a vertical position.

Dynamics The pole starts in a vertical position. The cart (base) can move left or right, swaying the pole in the process.

Objective Maintain the pole upright, for as long as possible.

State features

- lateral cart position
- lateral cart velocity
- pole angle
- pole angular velocity

Actions

- nudge cart to the left
- nudge cart to the right

Reward $+1$ for each timestep in which the pole is kept upright. Besides this original reward formulation, we impose an additional of -20 when the pole falls or the cart moves off-screen, in order to help the agent understand that it erred.

Episode end The pole falls, the cart is moved off the screen or 200 time-steps have passed.

Solve criteria Obtain a reward of over 200, i.e.: keep the pole upright for 200 time-steps.

The Cart Pole environment is fast to simulate and easily solved. Its simplicity cannot allow more sophisticated techniques to show their usefulness. It serves as a good initial test for algorithms because of its fast simulation time but cannot be used to showcase RL algorithms because of its low-performance ceiling.

3.2 LUNAR LANDER

This is a more complicated environment. Whereas the previous happens in 1-dimensional space, this takes place in 2 dimensions and features a more complex world.

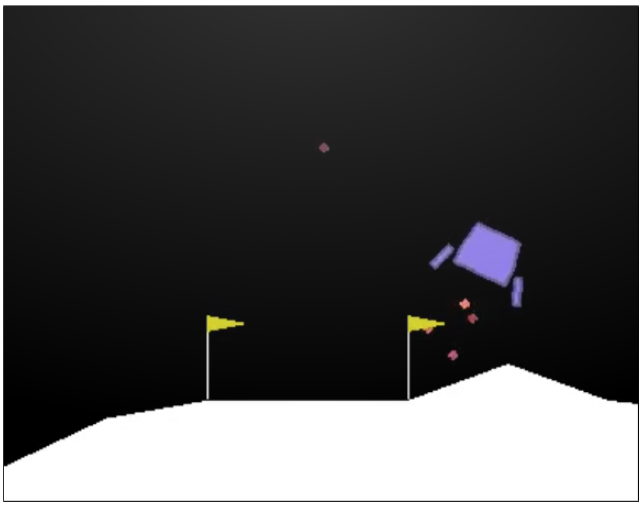


Figure 3: Example frame of the Lunar Lander environment. The spaceship (purple) is currently falling and rotating to the right. A small upward thrust and actuation of the left engine is required to land the ship on the soil (white) between the two flags (yellow).

Dynamics The ship is spawned at the top of the screen. It spaceship naturally falls through space and stops when reaching the soil. The main engine can be fired to push it upwards, while side-engines can rotate the ship clock-wise or counter-clockwise. The main engine consumes fuel when fired.

Objective Land the spaceship between the two flags, always centered at coordinates $(0, 0)$, while using as little fuel as possible.

State features

- ship x position
- ship y position
- ship x velocity
- ship y velocity
- ship angle
- ship angular velocity
- left ship leg touching soil (yes/no)
- right ship leg touching soil (yes/no)

Actions

- fire main (below) engine
- fire left engine
- fire right engine
- do nothing

Reward Positive for proximity to landing point and negative for fuel consumption.

Episode end Lands the ship between the flags, get off-screen or 1000 time-steps pass.

Solve criteria Obtain a reward 195 or more in 100 consecutive episodes.

The increased complexity of the environment is a double-edged sword. One one hand, it is a good test for RL algorithms. Weaker ones fail, while stronger ones can prove themselves. On the other hand, it is much slower to simulate. Running a random agent (no learning) for 100,000 episodes of Cart Pole takes under 1 second, while the Lunar Lander needs about 16 seconds. The increased state-space size already means agents need to observe substantially more situations. Coupled with the slowness of simulation, it means the number of experiments that can be made is greatly reduced.

4 AGENT ALGORITHMS

This section introduces the agents whose performance is measured. These algorithms are a part of a family of *value iteration* [Sutton et al. (2016)] methods. The learned strategy (i.e.: what action to take in each situation), known as *policy* is derived greedily from the value estimation, meaning that the action with the highest expected value is chosen, in each situation. This way, the problem of learning an efficient strategy boils down to accurately approximating state-action values. If one knows which action is best in each situation, following the induced course of action inevitably leads to the maximum reward.

Each subsection presents increasingly more complex techniques. From a basic tabular approach (4.1), to basic algorithm extensions (4.2) and finally to an approximate method (4.3) and its extensions (4.4).

4.1 Q-LEARNING

One of the simplest RL methods, Q-Learning (QL), builds a state-action *value* table $Q(s, a)$. It holds the utility estimates of taking action a in state s, for all actions tried in every encountered state. The pseudocode is presented in Algorithm 1.

Algorithm 1: Q-Learning with ϵ-greedy policy

Initialize Q-table arbitrarily, $Q(s, a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
foreach *episode* **do**
  $s \leftarrow$ environment's initial state
  repeat
    if *exploring (with probability ϵ)* **then**
      $a \leftarrow$ sampled randomly from $\mathcal{A}$
    else if *exploiting* **then**
      $a \leftarrow argmax_i\, Q(s, i)$
    $r, s' \leftarrow$ environment's reaction to taking action a
    $f \leftarrow$ future value $\begin{cases} 0 & \textbf{if } s' \text{ is terminal} \\ max_i\, Q(s', i) & \textbf{otherwise} \end{cases}$
    update $Q(s, a)$ towards computed action value $r + \gamma f$, with step size α
    $s \leftarrow$ next state s'
  until *s is terminal*;
end

The algorithm's parameters are the discount factor $0 < \gamma < 1$, usually close to one and the learning rate (or step size) $0 < \alpha < 1$, usually very small and the exploration probability $0 < \epsilon < 1$. The discount factor controls how much immediate rewards are preferred over delayed ones. Low values produce more hedonistic behavior while high values lead to more visionary behavior. The learning process is quite sensitive to the setting of these parameters. Initial $Q(s, a)$ values are set to zero.

One of the most common action selection strategies is *ϵ-greedy*. When prompted for the next action to take, from state s, the agent acts greedily w.r.t the value estimations by exploiting current information: $argmax_a\, Q(s, a)$. By acting greedily, the strategy may get stuck in a local optimum. To enable exploration of alternative, potentially better routes, a random action is selected, with probability ϵ.

4.2 Q-LEARNING EXTENSIONS

While it is a simple algorithm, QL can receive a couple of basic techniques which improve its convergence speed and stability. The slight downside of extending the algorithm is that each technique brings additional parameters which need to be tuned to obtain optimum performance.

Exploratory starts Instead of setting all initial $Q(s, a)$ values to zero, they can be initialized randomly. Usually, they are drawn from a normal distribution $N(\mu, \sigma)$ of zero-mean μ and small standard deviation σ. This provides artificial incentives for the agent to try different actions, especially initially, and reduces the chance of getting stuck in a local minimum.

Decaying learning rate Instead of keeping the learning rate α constant, it can be decreased over time, down to a set minimum. Usually, the decay is an exponential (multiplying by a sub-unitary coefficient) or step function (divided by a factor after a set period of episodes). This allows the agent to learn a lot from the environment at first, making rapid improvements at the beginning of training and fine refinements near the end, when finesse is required not to overshoot the optimum.

Annealed exploration rate Instead of abiding by the same exploration rate ϵ the whole training time, it can be annealed down to a set minimum. Usually, it starts very high (near 100% chance of exploration) and is decreased linearly over the course of a set number of episodes. This allows the agent to learn to explore a lot at first, when it does not have a good idea of which strategy works best, and later focus on improving the constructed policy, after gaining enough experience.

Idealization Instead of optimistically estimating next state value reward by looking at the maximum action value available then, the agent can take a more conservative approach and average the action values. This is known as the SARSA learning algorithm [Sutton et al. (2016)]. We introduce a generalization of these approaches, in the form of an *idealization* coefficient. It allows for cautious, but not cowardly behavior. Estimation of future value changes as follows:

$$f \leftarrow avg_a + \eta(max_a - avg_a)$$

where $0 \leq \eta \leq 1$ is the idealization coefficient, and *avg* and *max* refer to action a value estimates in the next state s'. Setting it to 0 corresponds to pure SARSA while setting it to 1 corresponds to QL.

Just by using these simple extensions, the problem of combining them becomes apparent. Perhaps decaying the learning rate over 300 episodes increases performance, and, separately, annealing the exploration rate over 200 episodes boosts it as well. But when both extensions are applied together, it could turn out that an even better performance is achieved when the decay happens over 150 episodes and the anneal over 100.

4.3 DEEP Q-NETWORK

The state-action value table used by QL can be replaced by an approximate model, such as a Neural Network (NN) Bishop (2006). This algorithm is called Deep Q-Network (DQN) Mnih et al. (2015) and is able to leverage state similarities to obtain better generalization. There is one input for each state feature and one output for each action. The pseudocode is presented in Algorithm 2.

Algorithm 2: Deep Q-Network algorithm with ϵ-greedy policy

Initialize Q-network weights randomly
foreach *episode* **do**
  $s \leftarrow$ environment's initial state
  repeat
    $q(i) \leftarrow Q$-network value estimation of each action i in current state s
    if *exploring (with probability ϵ)* **then**
      $a \leftarrow$ sampled randomly from $\mathcal{A}$
    else if *exploiting* **then**
      $a \leftarrow argmax_i\ q(i)$
    $r, s' \leftarrow$ environment's reaction to taking action a
    $f \leftarrow$ future value $\begin{cases} 0 & \textbf{if } s' \text{ is terminal} \\ max_i\ Q(s', i) & \textbf{otherwise} \end{cases}$
    $q(a) \leftarrow$ computed action value $r + \gamma f$
    fit Q-network on (s, q)
    $s \leftarrow$ next state s'
  until *s is terminal*;
end

The algorithm's parameters include the discount factor γ and the exploration rate ϵ. The remarks made in 4.1 apply here as well, including the description of ϵ-greedy action selection policy. Joining these hyper-parameters are the NN's learning parameters such as layer sizes, activation functions,

loss function, optimizer, weight initialization method and so on. As with any NN problem, they affect the outcome of the algorithm dramatically.

The presented algorithm is a slightly simplified version, with the purpose of easier comprehension. In practice, each transition (s, a, r, s') observed is stored in a circular memory buffer, with limited space, where old transitions are replaced by newer ones. Afterward, instead of fitting the Q-network on a single example (s, q), a batch of transitions is sampled from the memory buffer. This not only improves the practical performance of the agent, by allowing itself to replay past experiences after evolving but also satisfies the theoretical requirements of points to be identically and independently distributed, which is missed when transitions are processed sequentially.

While the increased generalization power of NNs can lead to better understanding of the environment's features, it also comes with the drawback of needing significantly more data, longer training times and having many more hyper-parameters to tune.

4.4 Deep Q-Network Extensions

This section presents recent RL advancements pertaining to DQN. This is far from an exhaustive list and focuses on methods that have been reported to yield great results in the literature together with ones the authors believe have great potential.

4.4.1 Reinforcement Learning Algorithm

These extensions target the RL flow itself, not so much the NN used to model the reward function.

Multi-step returns Described by Sutton et al. (2016). Instead of looking just one step ahead, taking an aggregate (average, maximum, etc) of action values in the next state and discounting them by γ, the agent can look further into future steps and discounting by $\gamma^2, \gamma^3, \gamma^4$ and so on. This improves the accuracy of future value estimation.

Double Introduced by Hasselt (2010). A second, identical, network is used for value predictions, updated (or slowly, but constantly) periodically to match the main network. This stabilizes the learning process by avoiding spiraling out of control when chasing a moving target. Additionally, it helps fight overly optimistic value estimations.

Dueling Introduced by Wang et al. (2015). The state-action value function $Q(s, a)$ is decoupled into state value $V(s)$ (the intrinsic utility of being in state s) and action advantage $A(a)$ (how much better it is to take action a compared to the other choices). This has the benefit that the value stream is updated more often, especially in environments where many actions are available.

Prioritized Experience Replay Introduced by Schaul et al. (2015). Instead of sampling transitions uniformly from the memory buffer, they can be sampled proportional to the NN error produced. This way transitions the model has much to learn from are favored and training time is used more fruitfully.

Distributional Introduced by Bellemare et al. (2017). Instead of estimating a single number for the state-action values, a distribution can be estimated. This way, an aggregation of the possible reward outcomes is not forced and non-normal reward distribution shapes can be modeled more efficiently. This technique is particularly effective when the environment presents high stochasticity or is influenced by information not reflected in the state features.

Noisy Nets Introduced by Fortunato et al. (2017). It is a way to inject exploration constraints straight into the estimation process, by adding parametric noise to the network weights.

Boltzmann Exploration The ϵ-greedy action selection policy can be exchanged for one that enables more guided exploration. Instead of sampling randomly, actions are selected proportional to their estimated values. There are multiple ways of controlling the amount of exploration done. The tendency to sample uniformly (as opposed to focusing on high-values) can be decreased as more

experience is gathered. Alternatively, in a *Max-Boltzmann* policy, proportional exploitation is done with probability ϵ, and the maximum value is selected otherwise, similar to ϵ-greedy.

State History Show more than just the latest state for deciding on an action, in order to capture temporal relationships. Previous states can have their features concatenated, or they can be explicitly handled by the network's architecture.

Asynchronicity Key component of Salimans et al. (2017). Instead of having a single agent, train multiple, independent ones and periodically sync their experiences. This enables fresh perspectives and teleportation out of local minima.

Other Directions More drastic extensions involve incorporating Policy Gradient [Sutton et al. (2016)] elements, as exemplified by Actor-Critic methods [Mnih et al. (2016)]. More exotic approaches make use of concepts such as *intrinsic motivation*, as is the case of Hierarchical-DQN [Kulkarni et al. (2016)]; or perform trust-region optimization [Schulman et al. (2015)].

Each of the described extensions changes the vanilla DQN algorithm significantly. Combined, they make an almost unrecognizable amalgamation. But the heart of the algorithm stays the same, only its performance is the one being altered (hopefully in a positive way).

4.4.2 Neural Network Model

These extensions target the NN predictive model but are limited to ones having particularly high applicability in RL.

Bayesian Networks Introduced as a RL technique by Gal & Ghahramani (2016). Bayesian NNs are better equipped to deal with uncertainty. Their theoretical properties can successfully be satisfied by interspersing Dropout [Srivastava et al. (2014)] layers.

Weighted Importance Sampling When computing the loss gradient, focus more on samples having greater errors. This allows the model to pay more attention to mistakes.

Batch Normalization Introduced by Ioffe & Szegedy (2015). It normalizes activations values among different batches, making the learning more robust in the face of high fluctuations.

Loss Function The Huber Loss function acts like the Mean Squared Error when the difference is small, and like the Mean Absolute Error when the difference is large. It increases the learning's robustness to outlier reward values.

Architecture If shown multiple previous states, a Recurrent layers [Goodfellow et al. (2016)], specifically Long Short Term Memory units, can successfully deal with temporal interactions. Even though the described environments do not have homogenous features with a spatial neighboring relationship, Convolutional layers [Goodfellow et al. (2016)] can still be used on time slices.

Other Components The usual moving parts of NNs, of course, have a big impact on model performance. Layer activation function, weights initialization method, and regularization need to be considered.

Ensembles As is the case of many NN applications, multiple models in cooperation usually outperform any single contender. To this end, a more complex model can be created by compounding multiple different configurations. This way, the strengths of one can complement the weaknesses of another and vice-versa. Their decisions can be aggregated by a majority-voting system, or a more sophisticated meta-model, such as a Decision Tree [Bishop (2006)], or another NN.

These model changes can have a dramatic impact on the NN's ability to successfully model the reward function. Not only by themselves but also when taken in combination with RL techniques described previously.

5 COMPLEMENTARY TECHNIQUES

This section presents some techniques that do not belong to the RL domain but are an essential part of their successful application. They target input transformation (5.1), algorithm evaluation (5.2) and the meta-optimization process of tuning hyper-parameters (5.3).

5.1 STATE BUCKETING

The description of QL (Section 4.1) uses a table as its key element. Being a two-dimensional data structure, it needs to assign one identifier to each state array. As most features (e.g.: x-position, angle, etc) are real-valued, there would be next to little use in using the uncountable-many options separately. It is highly unlikely that the *exact* same coordinates will ever be encountered again. Thus, no learning could occur.

The solution to this is to discretize the real-valued space. If feature a takes values in the $[-1, +1]$ domain, we can split it into 4 buckets, namely $[-1, -0.5]$, $(-0.5, 0]$, $(0, +0.5]$ and $(+0.5, +1]$. This allows for rules targeting $a = 0.6$ to also be applied to $a = 0.65$, and so on. This turns a continuous space of $[-1, +1]$ into a discrete space of $\{0, 1, 2, 3\}$, corresponding to each bucket. If there was an additional feature b, with the same number of buckets, then discretization would yield the space $\{(a = 0, b = 0), (a = 0, b = 1), (0, 2), (0, 3), ..., (3, 3)\}$. In a general way, n features segmented into b buckets create the space $(0, 1, ..., b)^n$.

DQN does not require state discretization and can work on the state vector directly.

5.2 PERFORMANCE METRICS

The official specifications define the goodness of an algorithm solely based on how quickly it can reach an average threshold of total reward. This metric fails to distinguish between two algorithms that do not reach the solve episode, even though one may be very close, while the other far away. Moreover, even for solving algorithms, the metric does not take into consideration how their performance fluctuates from one episode to another, or whether the model will start to overfit after reaching the solving threshold.

The following set of measure dimensions for total reward achieved, taken together, fix these shortcomings and provide a more robust evaluation of algorithm performance. *Tail* refers to the last episodes of an algorithm, usually 100.

- final reward – the algorithm should reach a good result
- tail average – even if the final reward is small, that could be due to fluctuations and the previous runs can show great performance
- tail standard deviation – the algorithm should be stable
- maximum reward – indicating the algorithm's potential, as its peak's location could not coincide with the last episodes

In order for the comparison to be fair, these metrics must be compared between algorithms that ran for the same amount of time.

5.3 HYPER-PARAMETER TUNING

To author's best knowledge there is currently no open-source hyper-parameter optimization library that meets all requirements of tuning hyper-parameters in many dimensions when evaluation is expensive and the scoring surface highly irregular. Some of these requirements would be:

- accept both continuous (linear/logarithmic scale), discrete (ordinal/categorical distinction) domains
- allow conditions to be imposed on parameter combinations (e.g.: *target model update frequency* should be non-zero only if *double DQN* is used)
- run in parallel, able to search faster the more cores it is allowed to use

- transparent inner process (e.g.: allow the tuning of exploration rate or other parameters)
- work on any kind of optimization problem, not just supervised
- able to be interrupted, and resumed, dealing effectively with shared-machine disturbances
- able to learn from prior knowledge (i.e.: integrate previous experiments results: configuration/outcome)
- able to take suggestions (i.e.: allow human guidance)
- work in Python, specifically recent versions of Python 3
- be stable and mature (have community support)
- come with sufficient documentation/examples
- easy to install (from a package manager) and able to be deployed on a shared-machine (e.g.: no system-wide dependencies)
- preference: multiple algorithms (different Gaussian Process kernels, Random Forest, etc)
- not a requirement: computational efficiency (since the cost of evaluating configurations vastly overshadows it)

We tested many (*hyperopt, bayesopt, metaopt/orion, gpyopt, comet.ml, spearmint, pybo, optunity*) available popular, and also emerging libraries. Only one, *skopt*, met most requirements and allowed for the others to be implemented manually.

6 OPTIMIZATION JOURNEY

This section is the heart of the report. It chronicles the methodology, techniques, and results, as well as questions, motivations, and opinions of the authors during the development process. It starts with a simple algorithm in a simple environment (6.1) then moves on to a more complex environment which also warrants a complex algorithm (6.2).

6.1 CART POLE AND Q-LEARNING

We begin this journey by tackling a simple problem at first: the Cart Pole environment (detailed in 3.1) on which the Q-Learning (described in 4.1) is set to learn. We present baseline results of the vanilla algorithm, followed by improvements brought upon by state pre-processing and various improvements, and ending with a short analysis of hyper-parameter relationships.

6.1.1 VANILLA BASELINE

When running the vanilla QL algorithm we see a great amount of randomness, among different random seeds. Figure 4a shows three selected random seeds, ranging from relatively good to really bad. Some are runs are lucky (as is the case of seed 7) and solve in as few as 4,000 episodes, while others are unlucky (e.g.: seed 6) and need as many as 187,000 episodes to solve the environment. Regardless, all runs display great variance from one episode to the next.

Because of the high number of episodes, we plot periodical evaluations instead of the reward obtained after each episode. After every 1,000 episodes, the agent is evaluated over 100 episodes. Its exploration functions are disabled and the total reward received after each episode is averaged. For easier visualization, a rolling average of 10 previous and following observations is plotted in an opaque trace, while the transparent one behind shows the un-smoothed values.

6.1.2 PRE-PROCESSING

After an examination of the state features, we had a hunch that cart's lateral position not matter that much, and should be given less attention compared to the pole's angle. Thus we make the following changes to state discretization:

- decrease the number of bins to 4 bins for lateral position and velocity, allowing for higher generalization
- increase the number of bins to 12 for angle and angular velocity, allowing for better focus on these attributes
- clip values to the 1 and 99 percentiles of observed values in each dimension, instead of the minimum and maximum values, making bins more relevant

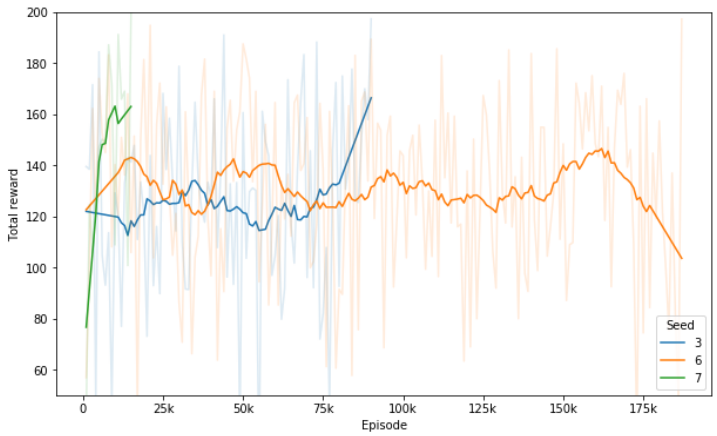
The change brought great success. The median number of episodes decreased from 62,600 to just 6,000 – over 10x improvement! The fastest run solved in just 2,000 episodes. Figure 4b shows, three relevant runs. Notice the x-axis scale, which ends at 10,000 episodes compared to the previous Figure, which ended at 200,000.

6.1.3 EXTENDING THE ALGORITHM

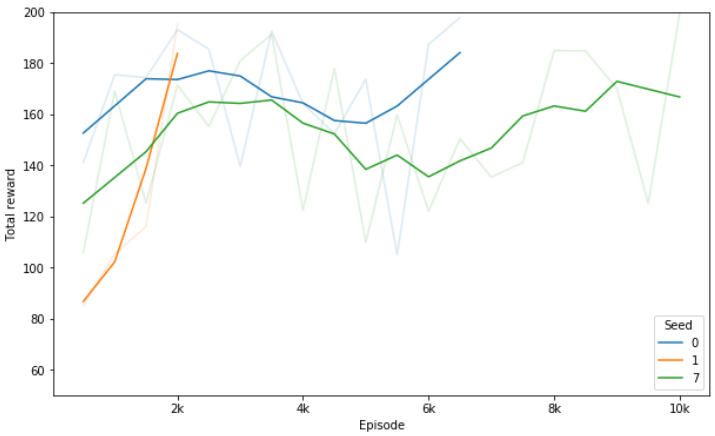
Next, we apply the QL extensions described in Section 4.2. Figure 4c shows the three most different runs, based on the random seed. They are all very similar. Overall, the training progress is a lot less noisy, shows more consistent convergence and is also a bit faster (again, notice the x-axis changes to 6,000 episodes). Median solve-episode drops to 4,500 episodes, down from 6,000; and the standard deviation from 3,800 to 1,600.

6.1.4 TUNING PARAMETERS

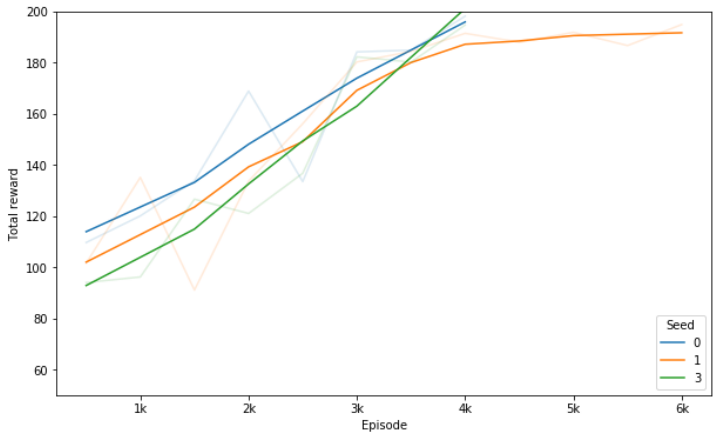
Running a quick parameter search on the extended QL algorithm, brought further improvements in speed and stability. Figure 4d shows the best performing run. It manages to solve the environment



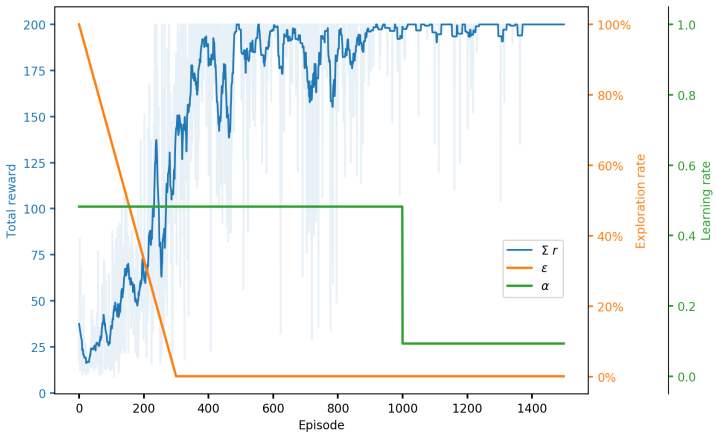
(a) Vanilla QL



(b) Vanilla QL, unequal discretization



(c) Extended QL



(d) Extended QL with tuned parameters

Figure 4: Training progress of the Q-Learning agent on the Cart Pole environment, under various configurations. Each sub-figure shows progressively better performance.

in just 950 episodes. Because of this, look directly at the total reward per episode, instead of doing periodical evaluations.

The impact of annealing the exploration rate is obvious: there is a clear inverse correlation between the exploration rate and the reward obtained, in the first 400 episodes. The agent is given some time to explore at first and then it switches to full exploitation. The impact of decreasing the learning rate is pronounced as well: after its value gets dropped, the noisiness of reward obtained immediately diminishes, after the 1,000th episode. The agent learns a lot in the majority of the training and then it is able to make small refinements at the end.

6.1.5 Conclusion

The best QL configuration manages to solve the Cart Pole environment in under 10 seconds (950 episodes).

We do not offer an analysis of DQN algorithm on the Cart Pole environment. QL does a splendid job solving it, and the increased complexity of DQN is not warranted by such a simple environment.

6.2 LUNAR LANDER AND DEEP Q-NETWORKS

Now that the simple environment is conquered, it is time to tackle the real problem: the Lunar Lander environment (detailed in 3.2), on which we use the DQN algorithm (described in 2). The approach follows the same structure as before: the vanilla algorithm is first tried, then input transformations and algorithm extensions are applied and finally, parameter tuning and analysis is presented. Only this time, the path to success is tortuous and riddled with difficulties.

6.2.1 VANILLA BASELINE

Inspired by the great success on the Cart Pole environment, we try to use QL on the more complex Lunar Lander environment as well. We quickly come to the realization that QL is no match for this environment. From solving Cart Pole in under 10 seconds to doing not much better than acting randomly after over 9 hours of training on Lunar Lander. It is clear that the problem is in a different league, and requires a much more powerful algorithm.

Thus, we let DQN have a shot at it. It does not do well out of the box. After 60 hours, it does even worse than QL. So we turn our attention to the technique that enabled an entire order of magnitude improvement: changing the method of pre-processing state features. Only this time, it was much trickier.

6.2.2 PRE-PROCESSING

One of the selling points of DQN is the fact that they do not rely so much on state-processing, the way QL does (which require buckets). DQN can accept raw state feature values. But even so, pre-processing has been shown to have a great impact on model performance – inputs are recommended to have zero mean and unit standard deviation. Thus, we applied the same method as we did for Cart Pole. Measure the range of values of each state feature, observed by acting at random.

It took significantly longer to just simulate the Lunar Lander environment, with no learning involved. Alas, centering and scaling the network inputs produced no noticeable improvement. We then realized that, because the environment is much more complex, random behavior is very different from what a trained agent exhibits. While the random agent spun erratically and went off-screen quickly, a trained agent almost never experiences such high values for angular velocity, and often encounters more values on the y axis, as it descends to its target.

To mitigate this shortcoming, we configured an agent according to successful ones reported in the literature [Keng (2016)] and used it to observe the state feature ranges it encounters. State features distributions of both random and trained agents are plotted in Appendix B. Running our model with this source of state pre-processing, brought no noticeable improvements yet again. One cause might be that at first, a learning agent acts randomly. Only late in training does it act closely to optimum. This is also reflected in the values it encounters. So it is of little use to focus on value ranges obtained by an optimal agent when at first it acts randomly. Another cause for the ineffectiveness for state pre-processing could be the fact that raw values already follow a distribution which is satisfactory for NN input.

Even though state range transformation did not affect performance much, one aspect regarding input values proved to be absolutely essential. Instead of presenting just the current state, for the model to select an action, it is presented both current and previous state vectors. They are being concatenated to form a 16-dimensional array.

6.2.3 EXTENDING THE ALGORITHM

Alas, our big weapon proved to be ineffective. Pre-processing brought nowhere near close to the 10x improvement it saw before, and the agent still failed to adapt significantly even after 60 hours of training. Thus, we try adding the extensions. Vanilla DQN did so badly which must mean that a huge number of tweaks are needed for performance to improve. So we implemented nearly all extensions described in 4.4. We let it run again and, disappointingly, it not only improve its performance, it actually degraded.

What could be the cause of this? The algorithms have been proven to work by many instances in the literature, why do they fail so much in this case? There is a major difference: the hyper-parameters

were, likely, tweaked to fit the problem they faced. In our case, we set the parameters to sensible values, but something was clearly done wrong. In the Cart Pole example, correct hyper-parameter settings brought tremendous improvements, so it follows that they should bring the same increase in this case as well. But how can we find out which parameter (or combination of parameters) are wrong, and moreover, which would be the appropriate values, considering the fact that the algorithm has over 30 knobs to be tweaked, and evaluation of a single configuration can take as long as two days?

6.2.4 TUNING PARAMETERS

Turns out there is no easy solution. This is where intuition, understanding of theoretical properties and a bit of luck comes into play. After consulting the literature for configurations that solve similar problems, we made some adjustments, mainly to the NN model. The most surprising are:

- Layer sizes – because the input's dimensionality is so small (8 features) and the output's as well (4 actions), we presumed a small network (one or two layers of sizes 8 - 32) would be able to solve the environment with ease. Instead, the networks yielding the best performance turned out to be surprisingly large. The best one features two layers of sizes 384 and 192 respectively, while the second-best distinct one has a single layer of 512 neurons.
- Hidden layers activation – *ReLU* is widely recommended as a general-purpose nonlinearity, having benefits such as seldom saturation and simplicity of implementation and interpretation. But in this scenario, it was vastly surpassed by the *sigmoid* function.
- Output layer activation – *softmax* seemed like a natural choice, because of the max_a and $argmax_a$ operations done during the RL algorithm. Experiments proved different. A $linear$ activation functions achieved top performance. This can be caused in part because of the need to model negative rewards as well.

Other aspects that helped significantly: clever initialization (randomly uniform, but taking into consideration the number of inward and outward node connections) and surprisingly low learning rate ($\alpha = 5^{-3}$), coupled with fixed-step decay instead of exponential decay. Also, the use of idealization ($\eta = 0.9$) proved superior to using either pure QL or SARSA alternatives.

With these changes, we arrived at a configuration which solves the environment in about 40 minutes.

After establishing a good starting point, we conducted parameter studies. We kept all other hyper-parameters the same and tried multiple values for the learning rate, both higher and lower. Then, with the best value found, we varied exploration rate. Then we repeated for layer sizes and so on. We ended up with a slightly better model (38 minutes to solve). But this way of manually searching hyper-parameter values sequentially suffered from a big drawback. Maybe a higher exploration rate also works best with a higher initial learning rate. Also, maybe it enables larger network sizes. The sequential method makes perturbations only along one dimension at a time. But there are heaps of dimensions, and the majority of them interact in unpredictable ways. Meaning that the relationships between parameters should also be studied, instead of evaluating each of them in a void.

Even though we did not find the optimum configuration, progress done is essential for further search. Reaching a configuration that solves the environment relatively fast, paves the way for manageable experimentation. It now takes under one hour to determine whether an extension or a parameter configuration is worthwhile. Even though they may be completely off, if they fail to solve the environment at least as fast as the current model, then there is no use in letting them run any further.

6.2.5 EXTENDING THE ALGORITHM... METHODICALLY

As it proved to be the case, it is not the quantity, but the quality of algorithm changes that benefit performance the most.

Previously, along with varying hyper-parameters, we also implemented boolean switches for each extension, so we may gauge their impact. We were unpleasantly surprised when observing that techniques which should offer only improvements, actually degraded performance. Double DQN (which oftentimes comes bundled with DQN) and Prioritized Experience Replay (which, when applied by itself in other domains, has been shown to bring great improvements) not only did not boost performance, they actually caused slight degradation.

Thus, we went back to the big question of what could cause such behavior. And narrowed down on the second major difference: the implementation of the algorithms. We started doubting our implementation correctness, so we started from the vanilla algorithm, which was proven to work. Then, we added back, iteratively, extensions one by one, testing learning behavior after each one to make sure that they do not render the algorithm ineffective. After consulting numerous other open-source implementations, we did find some minor mistakes, but nothing game-breaking.

Remember that measuring the impact of adding one extension requires to running the algorithm for over half an hour, just with a single hyper-parameter setting. This made the process of adding DQN extensions slow and progress arduous. The disappointing outcome is that they did not ruin learning, meaning they were implemented correctly. Instead, each of them caused a slight degradation in performance, which could be attributed to poor hyper-parameter settings.

We would have liked to do ablation studies on DQN extensions, and measure the success of each, individually. Then, finally combine them and see whether the sum is larger than the parts. But the results dictate that it is not possible, as neither one of them alone, not all together provide any significant performance boost.

6.2.6 TUNING PARAMETERS... THOROUGHLY

Previously, a great amount of time was spent on testing multiple configurations. It was done by hand, in a sequential way. Having to wait half an hour to see the results of some slight changes proved to be a severe impairment in development speed. Moreover, it is unlikely to find the best combination by searching manually. Performing an exhaustive grid search would, by definition, find the best configuration. But that option is far from feasible. Comparing as few as two options for each of the 30 variables (full list in Appendix C) means 2^30 or over 10^9 possible configurations. Moreover, there are many more options for each parameter (sensible ranges given in Appendix C), with the performance hyper-surface displaying highly non-convex behavior.

An alternative would be to randomly through the parameter search. But we wanted something with a higher chance of success, especially given the high cost of evaluating a single configuration. Thus we use a Bayesian Optimization technique as a black-box optimizer.

To be able to perform optimization in such a high-dimensional and expensive to evaluate context, it is an absolute requirement to be able to parallelize the search process. A Bayesian Optimization process is traditionally sequential, offering a suggestion and then updating its beliefs after being reported its score, and only afterward issue new suggestions. We modify this flow slightly:

1. ask for n suggestions
2. evaluate each suggestion, on n processors
3. report the result of each, and repeat

Also, the optimizer is imbued not only the knowledge of all our previous manual runs but also a number of randomly sampled configurations. They guard the possibility that manual tuning may have tunneled on some parts of the hyper-parameter space and overlooked others.

At first, the exploration of the acquisition function was set to high (not to be confused with the exploration rate of the RL algorithm). Multiple instances of the optimizer, with different random seeds were started. After a considerable number of runs, they were stopped and a second round of multiple optimizers was started, with everyone being fed the knowledge of all others and with the focus set on exploitation. At first, each configuration was allowed to run for 60 minutes. Since some configurations managed to solve in 30 minutes, we cut the maximum time allowed to 20 minutes, forcing faster convergence.

On the Atari benchmark, the best performing algorithms are the ones that steady enough to learn over immense periods of time (in the range of 10 days). On the Lunar Lander, the short time required (less than 30 minutes) to solve the environment, brings the focus to finding configurations that are able to robustly reach the solving threshold as quickly as possible. The single-value score needed by the black-box optimizer is an aggregation of metrics described in 5.2. *Tail average* has the largest impact, with all others contributing a little bit, and *tail standard deviation* having a small negative impact.

6.2.7 CONCLUSION

The best configuration found after an automated hyper-parameter search and the availability of most DQN extensions manages to solve the Lunar Lander environment in just under 25 min, in about 340 episodes. Figure 5 shows the evolution of reward, number of steps, exploration and learning rate over its training.

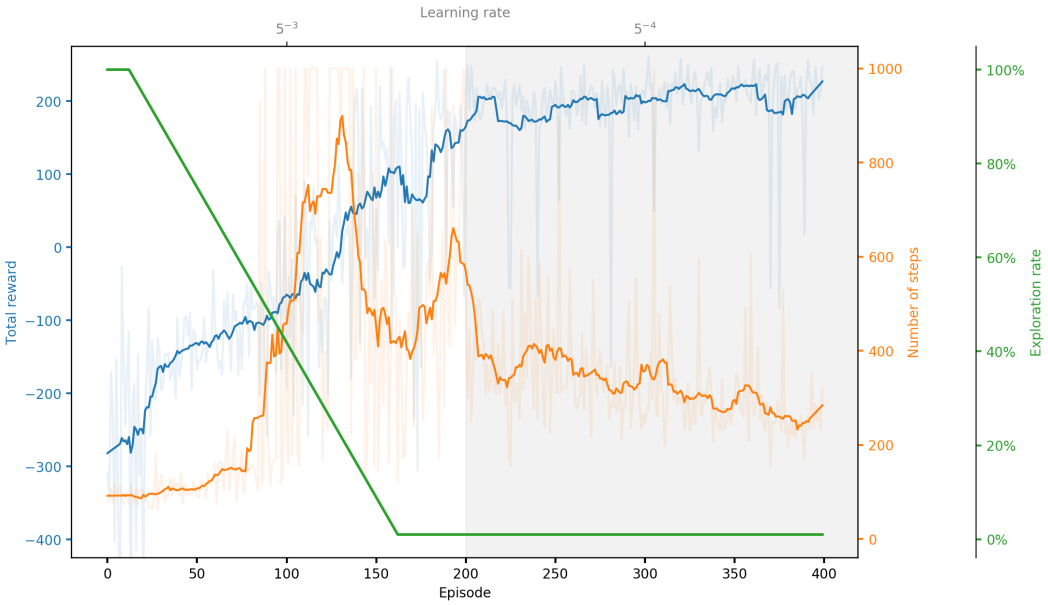


Figure 5: Training progress of DQN on Lunar Lander. Over the course of 400 episodes, the reward obtained (blue) increases and the number of steps (orange) first increase and then decrease gradually. The exploration rate (green) linearly drops in the first 150 episodes and the learning rate (grey shade) is set to 0.005 for the first 200 episodes and to 0.0005 for episodes 200-400.

Like QL on Cart Pole, the impact of exploration and learning rates is immediately obvious: exploration rate affects magnitude, learning rate affects noisiness. The algorithm is given some warmup time at first, then exploration rate starts dropping and total performance raises accordingly. It is interesting to note that the moment the agent is left on its own, with exploration nearing zero, it's the moment its performance sees a significant drop. But after a very short period of time, it goes back up and later surpasses previous performance. The effect of learning rate decimation (division by 10) is the exact opposite. After the drop, the agent's performance spikes up for a little bit, only to regain its natural growth afterward.

The number of steps the algorithm makes in each episode also tells an interesting story. In its nascence, the agent, performing nearly randomly, does not survive long and causes the episode to end quickly. Later, when it is given some room for exploitation, it takes the other extreme: doing too many steps (until the time limit is reached). As exploration dies down, it does fewer and fewer steps, but this time, they are the right ones, as performance is on the rise. Just as it begins to fall into nonsensical long behavior again, the decay of the learning rate comes in and allows for refinement of constructed strategy. Near the end of learning, we see the agent obtaining higher and higher reward while at the same time finishing an episode in fewer and fewer steps.

The progress of other runs is not shown because nearly all high-performing configurations follow the same learning curvature. And the rest of them either flat-line or, seldom rising above 100 reward. The entirety of runs, over 600, is available for inspection in configuration params-resulting metrics format.

We come back one last time to the question of why do proven DQN extensions not bring improvements in the Lunar Lander environment. After we have tuned hyper-parameters and checked the implementation, one last major difference, the largest one, remains: the nature of the problem itself. And this property we cannot change. As noted in previous sections, solving this relatively simple

environment in under half an hour is vastly different from training for days on end in far more complex arcade video games. While we cannot wholly dismiss the former two reasons, we believe this is the one hurdle keeping extensions from shining: the fact that they were developed to overcome issues not present in this simpler environment and consequently manage to degrade performance instead of improving it.

Finally, a final note about the optimality of the result. No doubt there are better solutions that solve the environment amazingly fast, but then again, even hard-coded rules do, and they require zero training time. We suspect such a system would have to specialize so much that its generalization power plummets, useful for little more than solving a toy problem, and not for advancing the understanding of RL algorithms so they could be used in the future for a task more significant.

7 CONCLUSIONS

In this project, we measured the effectiveness of Q-Learning (QL) and Deep Q-Networks (DQN) on the Cart Pole and Lunar Lander environments. QL extensions and pre-processing techniques proved essential to achieving good performance. The agent managed to solve Cart Pole in under 10 seconds. Hyper-parameter searching techniques, not so much extensions, are the ones that propelled DQN. It managed to solve Lunar Lander in under 25 minutes.

Neither environment turned out to be a worthy proving grounds for recent RL techniques. They have been shown to bring great improvements, but they solve problems that arise only when training for a significantly longer period of time.

REFERENCES

Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017. URL http://arxiv.org/abs/1707.06887.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

François Chollet et al. Keras. https://keras.io, 2015.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017. URL http://arxiv.org/abs/1706.10295.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 1050–1059. JMLR.org, 2016. URL http://dl.acm.org/citation.cfm?id=3045390.3045502.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Hado van Hasselt. Double q-learning. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2*, NIPS'10, pp. 2613–2621, USA, 2010. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2997046.2997187.

M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *ArXiv e-prints*, October 2017.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *CoRR*, abs/1803.00933, 2018. URL http://arxiv.org/abs/1803.00933.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

Wah Loon Keng. Open ai lab, 2016. URL http://kengz.me/openai_lab/.

Tejas D. Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Joshua B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *CoRR*, abs/1604.06057, 2016. URL http://arxiv.org/abs/1604.06057.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 00280836. URL http://dx.doi.org/10.1038/nature14236.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL http://arxiv.org/abs/1602.01783.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015. URL http://arxiv.org/abs/1511.05952.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL http://arxiv.org/abs/1502.05477.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15 (1):1929–1958, January 2014. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2627435.2670313.

Richard S. Sutton, Andrew G. Barto, and Harry Klopf. Reinforcement learning: An introduction second edition , in progress. 2016.

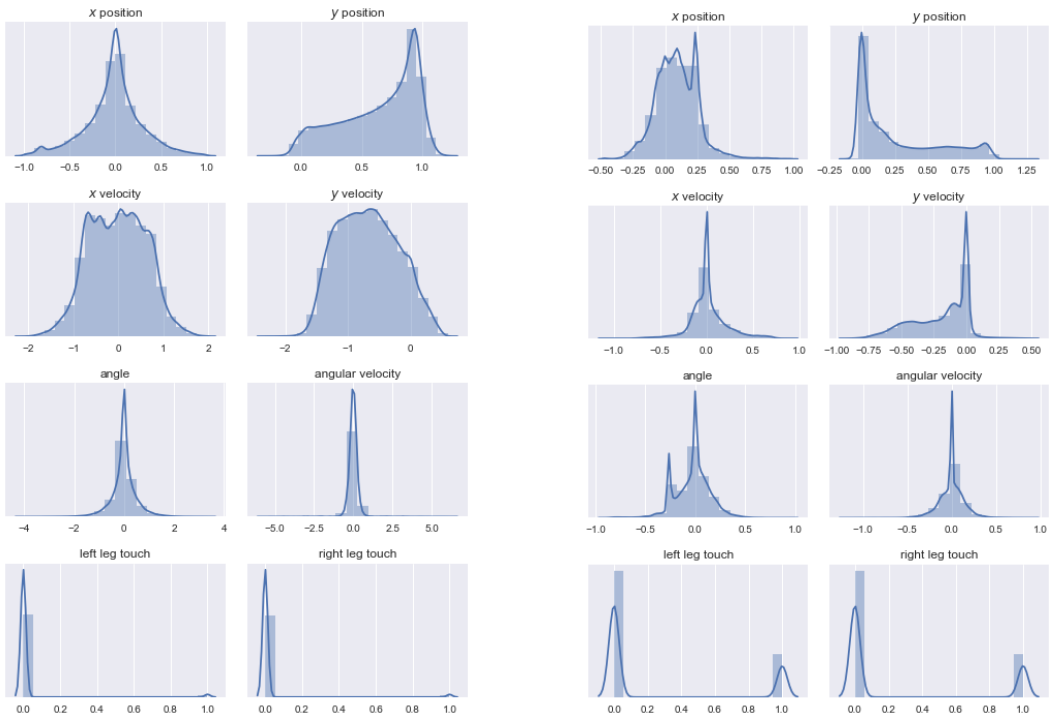
Skopt Team. Scikit-optimize, 2016. URL https://scikit-optimize.github.io.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

A Implementation Details

The environments and agents were run under Python 3.4 in CentOS 7, on Intel Xeon E5v3 CPUs and Tesla Nvidia K80 GPUs. DQN's neural networks use the Keras Chollet et al. (2015) implementation, the Bayesian Optimization process uses an open-source implementation as well Team (2016).

B LUNAR LANDER STATE FEATURES DISTRIBUTION



(a) Random (b) Trained

Figure 6: Histograms of each state feature, as observed by a random and by a trained agent

C HYPER-PARAMETERS

Table 1: Bayesian Optimization process parameters

Parameter	Chosen value
Acquisition function	LCB
LCB κ	1.96, then 0.3
Base estimator	Gaussian Process
Acquisition optimizer	L-BFGS
Initial Observations	64

Table 2: Q-Learning hyper-parameters

Parameter	Found value	Sensible range
Discount γ	0.75	[0.5, 0.999]
Exploration ϵ initial	1	[0.2, 1]
Exploration ϵ min	1e-3	[0.3, 0.0001]
Exploration anneal steps	300	[50, 2000]
Idealization η	0.125	[0, 1]
Learning rate α initial	0.5	[1e-7, 1]
Learning rate α decay	0.2	[1e-3, 0.95]
Learning rate α min	1e-4	[1e-10, 1e-7]
Decay Frequency	1000	[10, 2000]
Initialization mean μ	0	[-3, +3]
Initialization std σ	0.1	[0, 3]

Table 3: DQN hyper-parameters

Parameter	Found value	Sensible options
batch_normalization	no	{yes, no}
batch_size	32	[1, 8192]
discount	0.99	[0.5, 0.999]
double	no	{yes, no}
dueling	no	{yes, no}
exploration_anneal_steps	150	[40; 400]
exploration_min	0.05	[0.3, 0.0001]
exploration_start	1	[0.2, 1]
exploration_temp	2	[0.5, 10]
exploration_temp_min	0.2	[0.001, 1]
hidden_activation	sigmoid	{sigmiod, tanh, selu }
hidden_dropout	0	[0, 0.9]
history_len	2	[1, 5]
idealization	0.9	[0, 1]
input_dropout	0.2	[0, 0.6]
layer_sizes	(384, 192)	$[8, 1024]^{[1,5]}$
loss	mse	{mse, mae, logcosh}
lr_decay	0.1	[1e-3, 0.95]
lr_init	5e-3	[1e-7, 1]
lr_min	1e-5	[1e-10, 1e-7]
memory_size	50,000	[5,000; 1,000,000]
multi_steps	1s	[1, 8]
n_epochs	1	[1, 10]
out_activation	linear	{linear, softmax}
policy	eps-greedy	{eps-greedy, boltzmann, max-boltzmann}
prioritize_replay	no	{yes, no}
priority_exp	0.01	(0, 1)
priority_shift	0.1	(0, 5]
q_clip	(-1,000; +1,000)	$[0.1, 100000]^2$
streams_size	32	[4, 512]
target_update_freq	25	[100; 5,000]
weights_init	lecun_uniform	{simple/lecun/he_uniform/normal}
min_memory_size	1000	[32, 10e4]