THE UNIVERSITY OF EDINBURGH

INFORMATICS HONORS PROJECT

Generative Adversarial Networks Generating Novel Reinforcement Learning Policies

Author:
Giovanni ALCANTARA

Supervisor:
Dr. Timothy HOSPEDALES

A thesis submitted in fulfilment of the requirements for the degree of Bachelor's of Engineering

in the

School of Informatics
College of Science & Engineering

February 9, 2018

THE UNIVERSITY OF EDINBURGH

Abstract

Faculty Name
College of Science & Engineering

Bachelor's of Engineering

Generative Adversarial Networks Generating Novel Reinforcement Learning Policies

by Giovanni ALCANTARA

TODO:

Contents

Chapter 1

Introduction

TODO:

1.1 Motivation

1.2 Structure of the report

1.3 Data pipeline

1.4 Main contributions

Chapter 2

Background

2.1 Reinforcement Learning

2.1.1 Markov Decision Processes

Environments in traditional reinforcement learning application are usually modelled as Markov Decision Processes or MDPs.

These can be formulated as systems with the following components:

- Finite set of states $S = \{s_0, \ldots, s_n\}$ and actions $A = \{a_0, \ldots, a_m\}$.
- A distribution of probabilities $P_a(s, s')$ for transitions from state s to s' for each possible action a.
- A reward function $R : S \mapsto \mathbb{R}$ for being at a particular state.
- The goal in reinforcement learning is to maximise the final reward that an agent achieves in the environment.

As the name suggests, MDPs obey the *Markov property*, whereby the probability of the system being in a certain future state exclusively depends upon the present state, and not upon an arbitrarily-long sequence of past states.

In figure 2.1 we show a sample schematic of a Markov Decision Process, and how states, actions, and rewards could be connected between each other.

2.1.2 Q-learning

Now that we contextualised reinforcement learning environments as Markov Decision Processes, we can introduce the final objective of reinforcement learning tasks: finding a function $\Pi : S \mapsto A$ called **policy**, that maps the appropriate action $a \in A$ given the current state $s \in S$, as to maximise our agent's final reward.

In particular, we will now present Q-learning, an algorithm that yields an optimal policy given an MDP.

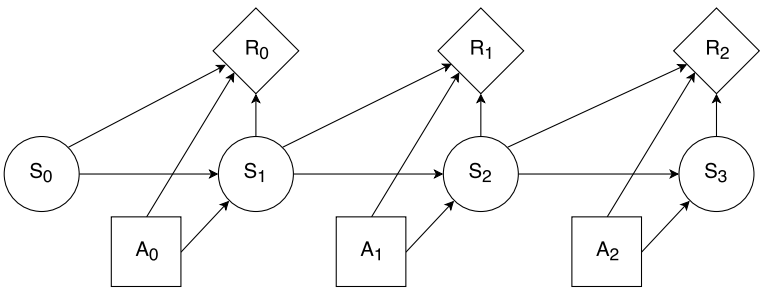

FIGURE 2.1: Sample schematic of an MDP.

Q-learning lets us learn the *quality*, or expected utility, for each state-action combination. That is, for each state, let's estimate all the expected rewards we obtain by taking each possible action at that particular state.

More formally, we estimate a function $Q : S \times A \to \mathbb{R}$. We can model Q as a mapping table (initialised with some uniform values), whose value we update at each time step of our simulations.

Here's how we update our Q-table at each time step t:

$$Q(s_t, a_t) = \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \times \Bigg[\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg]$$

where:

- $\alpha \in [0, 1]$ is the *learning rate*, a coefficient that regulates how much the newly learned values will contribute in the update
- $\gamma \in [0, 1]$ is the *discount factor*, a coefficient that controls the weight of future rewards. Values closer to 0 will make our agent "short-sighted", considering only the immediate rewards.

What is our optimal policy when we do Q-learning then? After training, it is simply that function $\pi : S \to A$ that, for each state, returns the action with maximum expected utility in our Q-table.

2.1.3 Exploration/Exploitation tradeoff

An important theme in reinforcement learning, and that we heavily focus our attention on in our project is the idea of the tradeoff between Exploration and Exploitation.

2.1.4 Deep Reinforcement Learning

DQGAN

2.1.5 Problems with reinforcement learning techniques

2.1.6 Actor critic

2.2 Generative Adversarial Networks

2.2.1 Architecture of GANs

2.2.2 Successes

DCGAN

2.2.3 Conditional GANs

2.3 Existing related work

2.3.1 Generative Adversarial Imitation Learning

Chapter 3

Environment

In this chapter we report the process that went into choosing the environments that we will be using in the project's simulations, and from which we will be basing our simulations.

There are many choices of environments and tasks that are publicly available. Some of these we will be introducing in this chapter.

What makes this step non-trivial and deserving of its own chapter is that different reinforcement learning techniques are more suitable to different categories of tasks. Similarly, different machine learning and deep learning techniques are more or less efficient when applied to different tasks.

In a research effort that is heavily dependent on building reinforcement learning and deep learning models, the choice of environment is a critical one.

Furthermore, we also need to achieve this without losing focus on the main motivations for the whole project (Chapter 1): *optimising reinforcement learning algorithms by adding transferability of pre-trained models on unseen maps or configurations of a task*.

This last point implies that a substantial part of the computational work in the project will be about training hundreds of thousands of reinforcement learning models to build a dataset over a distribution of different maps (we present this in Chapter 4). This is an important point: in the many experiments we ran, this turned out to be the biggest bottleneck, and required plans to distribute computations across multiple machines to make the computational time feasible in the timespan allocated to the project.

With these points in mind and given the experimental nature of the work, we conclude that a bottom-up approach in complexity is preferable. Given successful results with "easier" tasks, we can scale up in complexity and hopefully formalise and generalise our approach to more tasks (Chapter 7).

Easier tasks will enable us to explore different reinforcement learning approaches that we introduced in Subsection 2.1 with the guarantee that they will give satisfiable results. We can use these results as a foundation of the further steps (specifically Generative Adversarial Networks training in Chapter 5).

Before introducing candidate environments, let us define what is meant by an "easy" reinforcement learning task. What we are looking for is ideally a task with a discrete and relatively small set of observable states and actions. Why does this condition make the task easier?

Imagine building a tree (such as the one shown in figure 3.1 with all possible states-action transitions, until we either: 1) reach a goal state, or 2) reach an arbitrarily maximum iteration time step $t = \eta$ or depth of the tree (to prevent infinite iterations). Also assume we were building this tree in a bruteforce manner (worst-case scenario of reinforcement learning resolutions), such that we need to build all possible paths or trajectory that the agent will need to take. Therefore, we would

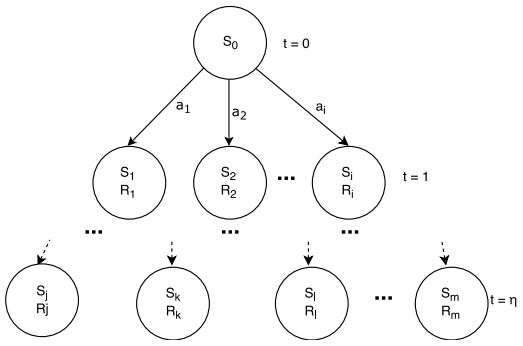

FIGURE 3.1: Sample state-action tree

need to visit each node of the tree, until we arrive at the leaves. At the leaves, we can find the achieved reward for the agent given the path it has taken to get there.

The breath and depth of the state-action tree will increase as we increase the possible set of states we would need to traverse, adding up to the space and time complexity of our solution, which is exponential in $\#S$ and $\#A$ for both space and time ($\#S$ indicates the cardinality of a set S).

With that said, there is a solid line between our ideal environment and a trivial one that could be solved without the aid of reinforcement learning techniques.

Let's explore some candidate environments next.

3.1 OpenAI Gym

OpenAI Gym (Brockman et al., 2016) provides a toolkit that aids in the process of building reinforcement learning systems and evaluating algorithms to solve such tasks.

OpenAI Gym provides an environment interface `Env`. The interface abstracts the following operations on an environment:

- `step(action)` – Simulates one time step by executing the provided action. This returns `observation` (the new current state), the `reward` for taking such action, a flag `done` indicating whether the system has reached a final state, and `info` providing additional information dependent on the environment.
- `reset()` – Resets the environment, i.e. the initial state is restored
- `render()` – Renders the environment in human-readable format.

Now, we could either build implementations for such interface (if we were to implement our own environment's logic), or use the provided implementations for several environments in the OpenAI Gym library, which includes board games, algorithm-based or physics-based environments, Atari games, etc.

3.1.1 Motivation

In this project we will mostly deal with environments provided in the OpenAI Gym. There are several reasons for such decision:

- It abstracts the need to implement the logic of a separate environment. Implementing our own environment adds a point of failure to our whole (quite

experimental) work, as well as imposing a bigger time constraint to the one we already have;

- OpenAI Gym has become a standard academic tool for reinforcement learning researchers, therefore many papers and articles build on top of this framework;
- Environment implementations in the OpenAI Gym library are constantly expanding and being revised by an active community, also thanks to the support of the overarching organisation (OpenAI);
- The core implementation of the `gym` module (*openai/gym on Gihub*) allows for straightforward extensions on existing environments, which, as we will see in subsection 3.3, will be critical in this project;
- While sadly not applicable anymore, OpenAI Gym used to provide and support an online platform for developers to compare the performance of different reinforcement learning algorithms for each task. This was in a form of a leaderboard, measuring the performances, as well as providing the implementation and data of winning techniques. Such data could have been used directly as input to our Generative Adversarial Networks.

3.1.2 Algorithmic environments

3.1.3 MuJoCo and physics environments

3.1.4 Other environments

3.2 Baseline: `FrozenLake-v0`

One of our baseline environments is `FrozenLake-v0` (*OpenAI Gym*), one of the algorithmic environments provided in the OpenAI Gym. In this section we describe the task, motivation for choosing it as one of our baseline models, and some shortcomings that we faced in using this environment in our project's pipeline.

3.2.1 Description of the task

In `FrozenLake-v0` we control an agent in a grid world, more precisely the grid shown in figure 3.1. The objective is to go from a starting tile (`S`) to a goal state (`G`), by moving in four possible directions from a tile: `up`, `down`, `left` and `right`.

What differentiates this from a trivial path-finding or planning problem? A couple of things:

1. there are both walkable and non-walkable tiles in the grid (these are respectively frozen tiles `F` and holes `H`), and
2. tiles are "slippery", as in the agent could "slip" while navigating the grid world, meaning that the movement direction of the agent is uncertain and only partially depends on the direction we tell the agent to follow, i.e. the direction is non-deterministic.

The reward in `FrozenLake-v0` is 1 for reaching the goal state, and 0 for being in any other state. The system stops executing when the agent falls into a hole, and the environment needs to be reset.

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	**G**

TABLE 3.1: `FrozenLake-v0`'s default 4x4 configuration

3.2.2 Motivation and shortcomings

`FrozenLake-v0` is generally classified as a straightforward reinforcement learning task. An optimal solution could be found by creating a model of the environment, that is merely recording where frozen tiles are while exploring the grid world. This is a *model-based* approach to reinforcement learning, and it is perhaps less interesting than learning by exploration, without being biased by the particular configurations of the map.

Model-free algorithms, like Q-learning, need no accurate representation specific to the environment, and they are therefore more transferable to different configurations of the map or even to different tasks, which is the final objective of our project.

In fact, let's take the case of Q-learning applied to `FrozenLake-v0`: we do not need to have an explicit knowledge of the dynamics of each different tiles. We do not build a policy that explicitly favours movements towards frozen tiles or towards the goal. In fact, the agent does not have a model of what a frozen tile is, nor a model of the goal tile or a hole–it just learns by exploration that there it is rewarded when it gets to the goal state, and that is what it implicitly aims for.

This is the sort of behaviour that we can transfer to different tasks–again our ultimate objective.

In its current OpenAI Gym implementation, `FrozenLake-v0` is a static map with the fixed configuration that is shown in figure 3.1. There is currently no way to generate random configurations of the map, so next up, we will be extending this implementation to account for that.

3.3 Extended baseline: Randomised Frozen Lake

So we need to extend OpenAI Gym's implementation of `FrozenLake-v0` so that it can generate random configurations of the map.

Before we move on, let us contextualise this in the bigger picture as to not lose focus of what we are trying to achieve. Why do we need different configurations of the map, again? We want to train reinforcement learning models on different configurations so that we can have a distribution of policies over different maps which we can use as input to our GAN. After training our GAN, we will have a Generator network spawning new policies for unseen configurations without having to find it through (computationally expensive) reinforcement learning algorithms!

Listing 3.1 shows how we implemented the algorithm to generate random maps. The critical line is line 4, which uses `numpy`'s `random.choice()` method that samples a matrix of a given size, given probabilities for each element (`'F'` and `'H'` tiles in our case).

So, we can pass in the desired size of the map. By default, it will generate 4x4 grids like the one in `FrozenLake-v0`, but we could generate maps of arbitrary size,

which will result in higher task complexity. We explore these harder extensions in Chapter 7.

We can also pass it the probability that a tile will be a frozen one through the parameter `p`. The presence of fewer frozen tiles, and therefore of more holes, makes the goal harder to achieve for the agent.

```
1 def generate(size=4, p=0.8):
2     valid_map = False
3     while not valid_map:
4         config = np.random.choice(['F','H'], (size, size), p=[p, 1-p])
5         config[0][0] = 'S'   # set top left to be the starting tile
6         config[-1][-1] = 'G' # set bottom right to be goal tile
7         valid_map = is_valid(config)
8         p *= 1.05            # increase probability of frozen tile
9     return ["".join(x) for x in config]
```

LISTING 3.1: Algorithm to generate random configurations. Utility function `is_valid()` of line 7 is shown in listing 3.2

Notice how the `generate()` function in listing 3.1 only returns valid maps, that is, maps that have at least one frozen path from start to goal. Surely, we could train models on environment configurations that are not solvable. Q-learning, for example, would just return a `Q` matrix with all Q-values equals to 0, since the agent will never get to the goal tile and get its reward. If it is unclear why, refer to the Q-learning subsection 2.1.2.

Using such constraint on map validity, we can limit the number of models we need to train by a significant amount, therefore reducing training time.

To check whether a map is solvable we use depth-first search from the start tile to the goal. If there is such path, then it is a valid map. Listing 3.2 shows the algorithm:

```
1 def is_valid(arr, r=0, c=0):
2     if arr[r][c] == 'G':
3         return True
4 
5     tmp = arr[r][c]
6     arr[r][c] = "#" # temporary mark with '#' to remember visited tiles
7 
8     if r+1 < size and arr[r+1][c] not in '#H': # go down
9         if is_valid(arr, r+1, c) == True:
10            arr[r][c] = tmp
11            return True
12    if c+1 < size and arr[r][c+1] not in '#H': # go right
13        if is_valid(arr, r, c+1) == True:
14            arr[r][c] = tmp
15            return True
16    if r-1 >= 0 and arr[r-1][c] not in '#H': # go up
17        if is_valid(arr, r-1, c) == True:
18            arr[r][c] = tmp
19            return True
20    if c-1 >= 0 and arr[r][c-1] not in '#H': # go left
21        if is_valid(arr,r, c-1) == True:
22            arr[r][c] = tmp
23            return True
24 
25    arr[r][c] = tmp
26    return False
```

LISTING 3.2: Depth-first search to check if a frozen lake map is valid

So the `generate()` function returns a valid random frozen lake map represented as a list of strings. An example of an output is $[\text{"}SFHH\text{"}, \text{"}HFHH\text{"}, \text{"}HFHH\text{"}, \text{"}HFFG\text{"}]$. Each string in the list encodes a row configuration of the frozen lake.

What is left to do now, is to extend `FrozenLake-v0` so that we could pass in any map configuration, therefore exposing our randomly generate maps to the `gym` environment interface we described in earlier in section 3.1. Listing 3.3 shows how to register a new environment by extending a pre-existing one.

```
random_map = generate(size=4, p=0.8)
register(
    id='RandomisedFrozenLake',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={'is_slippery': True, 'desc': random_map},
    max_episode_steps=100,
)
```

LISTING 3.3: Code to extend `FrozenLake-v0` with random maps.

Now, to start doing simulations on our new environment we can just initialise it just like any other OpenAI Gym environment:

```
env = gym.make('RandomisedFrozenLake')
```

We created a pull request to the OpenAI Gym's Github repository integrating this feature supporting random maps for `FrozenLake-v0`, so that developers and researchers could make use of these functionalities (*Add ability to generate random frozen lake maps by gvsi · Pull Request #835*).

Chapter 4

Dataset creation

In Chapter 3 we finalised our OpenAI environment choice and have a systematic way to generate different configurations of our environments.

The next step is to build up our dataset by solving as many of these configurations as possible with the use of reinforcement learning algorithms.

We justified in previous sections (specifically section 2.1.2 and section 3.2.2) how using *model-free* reinforcement learning algorithms will put us in the right direction to achieve the project's goals highlighted in section 1.1: improve the transferability of pre-trained models to different configurations, environments, and tasks.

One choice that is left to make before we move on to train models using our randomised environments is what reinforcement learning algorithm we should use.

4.1 Q-learning on `RandomisedFrozenLake`

We have presented Q-learning in section 2.1.2 as a method that generates a policy by building up a table Q of values corresponding to the expected utility of taking each action at each observable state.

This table Q is represented as a 2D-matrix of size (`env.observation_space.n`, `env.action_space.n`), which in the case of `RandomisedFrozenLake` is just a (16x4) matrix.

This is good news. Having a policy that is encoded as a 2-dimensional data structure makes it an obvious input to our Generative Adversarial Network later on in the next chapter. In fact, as we presented in section 2.2.2, GANs have proven successful in image synthesis applications, where inputs were images, that is 2D matrices.

The choice of Q-learning as our reinforcement learning technique therefore becomes preferable. Other model-free algorithms like policy search may not have a clear 2D representation in the way the trained parameter set θ is encoded.

4.2 Experiment set up

Before we proceed, we need to be able to record some details about the process of training our data. If our ultimate objective is to benchmark performance of traditional reinforcement learning agains our proposed approach, we need to record running time of our training, and see if that improves at the end.

Let's encapsulate training of a configuration in an `Experiment` class, whose code is shown in listing 4.1. An `Experiment` is initialised with an OpenAI Gym environment, which in our case is an instance of a `RandomisedFrozenLake`, and an integer value `num_episodes`, indicating the number of independent simulations the Q-learning algorithm will be doing. By calling `Experiment`'s `run()` instance method, we actually start training the model, with Q being updated at each iteration.

The instance variable `score` could be used as an evaluation criteria of the effectiveness of our training. It records the average reward the agent achieves for each episode during training.

`Experiment` also has a utility method called `dumps()` that serialises all this data and allows us to save it on disk.

```
class Experiment(object):
    def __init__(self, env, num_episodes=10000):
        self.env = env
        self.Q = np.zeros([self.env.observation_space.n, self.env.
    action_space.n])
        self.num_episodes = num_episodes
        self.score = None
        self.valid_score = None
        self.start = None
        self.end = None

    def run(self):
        self.start = datetime.now()
        # ------------------------------------------------------------
        # Run Q-learning algorithm, saving the rewards of each episode
        # ...
        # ...
        # ------------------------------------------------------------
        self.end = datetime.now()
        self.score = sum(rewards)/self.num_episodes

    def dumps(self):
        return dumps({'Q': self.Q, 'start': self.start, 'end': self.end, '
    score': self.score, 'num_episodes': self.num_episodes})
```

LISTING 4.1: `Experiment` wrapper class to train one instance of `RandomisedFrozenLake`

It's critical to be able to evaluate the quality of the Q-table after training. To do so we just use the optimal policy (that is the policy that picks the action that has the maximum expected utility according to the Q-table) and run if for a certain number of episodes. A validation score could be then defined by the average reward achieved at each episode. Listing 4.2 shows the code to achieve that.

```
def validate(self):
  rewards = 0
    for i in tqdm(range(num_episodes)):
        while j < 200: # Limit to 200 time steps
            j+=1
            # Choose an action by pick best action from Q-table
            a = np.argmax(Q[s,:])

            # Get new state and reward from environment
            s1,r,d,_ = env.step(a)
            rewards += r
            s = s1
            if d == True:
                break
     self.valid_score = rewards / num_episodes
```

LISTING 4.2: Code to validate a trained Q-table

4.3 Distributed Q-learning

Now that we formalised our experiment setup, we can run Q-learning on each of the map configurations of our `RandomisedFrozenLake`. For the 4x4 grid there are 3827 possible valid map configurations. It takes an average of 15 seconds to train each Q-learning table on a 2.5 GHz Intel Core i7 processor. On a single machine, it would take around 16 hours to run each experiment.

To make this step of our pipeline faster and scalable to more data, we decide to set up a distributed processing on a cluster with multiple machines. More precisely, we set up a MapReduce framework (Dean and Ghemawat, 2004) implementation running on a Hadoop cluster (Shvachko et al., 2010).

A MapReduce program is composed of a Map procedure that takes in some (large) input and performs a particular operation whose output is then fed into another Reduce procedure, which outputs the final result. The power of MapReduce is that the framework orchestrates the processing of these procedures by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

Figure 4.1 shows the distributed architecture to train multiple Q-learning instances. The input to the system is a list of strings, each representing the map configuration of a `RandomisedFrozenLake`, e.g. *"SFHHHFHHHFHHHFFG"*. These inputs are fed into Mapper programs running on different machines. The mapper's task is to initialise the experiment and run Q-learning on the environment. It outputs a key-value pair, where the key is the string that uniquely identifies a map configuration, and the output is the `Experiment` object that encapsulates the already trained Q-table. We have yet to assign a validation score to this table we trained, and that's the job of the reducer.

Each experiment's result is then written on an output file by the reducer. Each line will be again a key-value pair, with the map configuration string as key and the `Experiment` result dumped in string format. In our following sections, we can just parsing and load these results to conduct our further analysis.

While this particular architecture does not make full use of the power of MapReduce (combining, partitioning and sorting), it is an optimal and convenient pipeline to distribute our computations across a cluster of machines, thereby drastically reducing our training time.

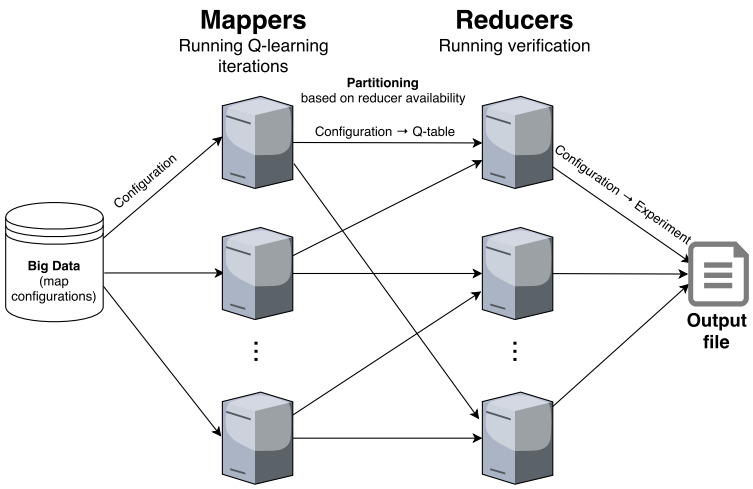

FIGURE 4.1: Schematic of distributed Q-learning with validation on MapReduce

Chapter 5

Adversarial Networks Training

TODO: Here I'll describe my architecture for both networks, show some interesting snippets of the PyTorch code to run the training on GPU. I'll motivate design decisions for the architecture of both generator and discriminator (why the number of hidden layers, and the number of nodes).

Chapter 6

Generative Adversarial Q-learning

6.1 Using the trained Discriminator

6.1.1 Speeding up Q-learning on unseen maps

6.2 Using the trained Generator

6.2.1 Initialisation

6.2.2 Exploration

Chapter 7

Scaling up to more complex tasks

7.1 Larger Frozen Lakes

7.2 Physics environments

Chapter 8

Conclusion

Bibliography

Add ability to generate random frozen lake maps by gvsi · Pull Request #835. https://github.com/openai/gym/pull/835. (Accessed on 01/26/2018).

Brockman, Greg et al. (2016). *OpenAI Gym*. eprint: arXiv:1606.01540.

Dean, Jeffrey and Sanjay Ghemawat (2004). "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

OpenAI Gym. https://gym.openai.com/envs/FrozenLake-v0/. (Accessed on 01/25/2018).

openai/gym on Gihub. https://github.com/openai/gym/tree/master/gym. (Accessed on 01/25/2018).

Shvachko, Konstantin et al. (2010). "The hadoop distributed file system". In: *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, pp. 1–10.