Reinforcement Learning in Pacman

Abeynaya Gnanasekaran, Jordi Feliu Faba, Jing An
SUNet IDs: abeynaya, jfeliu, jingan

I. Abstract

We apply various reinforcement learning methods on the classical game Pacman; we study and compare Q-learning, approximate Q-learning and Deep Q-learning based on the total rewards and win-rate. While Q-learning has been proved to be quite effective on `smallGrid`, it becomes inefficient to find the optimal policy in large grid-layouts. In approximate Q-learning, we handcraft 'intelligent' features to feed into the game. However, more powerfully, Deep Q-learning (DQL) can implicitly 'extract' important features, and interpolate Q-values of enormous state-action pairs without consulting large data tables. The main purpose of this project is to investigate the effectiveness of Deep Q-learning based on the context of the Pacman game having Q-learning and Approximate Q-learning as baselines.

II. Introduction

We want to train the Pacman agent to perform cleverly by avoiding the ghosts and eating the food and scared ghosts as much as possible (i.e. to get higher scores). The motivation of working on this project is that we not only want to do reinforcement learning and implement a neural network on our own, but also we think seeing a trained Pacman agent is visually attractive. All the reinforcement learning methods we implemented in this project are based on the code that implements the emulator for Pacman game [1].

For Q-learning (SARSA), the inputs are the states, actions and rewards generated by the Pacman game. For Approximate Q-learning the inputs are the hand-crafted features in each state of the game. Images are fed as inputs to the Deep Q-network. We track the scores and the winning rates as outputs to measure the efficiency of our implemented methods.

III. Related work

Most of the work in literature deal with Pong, Atari games in general. The most relevant work is done by Mhin et al. ([2], [3]), where they use the Deep Q-Learning (DQL) to train the player in Atari games. The idea behind DQL is to approximate the Q function with a deep convolutional neural network (Deep Q-Network). We have based our implementation of DQN on these two papers. Since it was proven that DQN could have really good performance in Atari games, even better than human performance in some games, DQN has received a lot of attention and many improvements have been proposed. Remarkable improvements are:

- Deep Recursive Q-network [4] (DQRN) which is a combination of a Long Short Term Memory (LSTM) and a Deep Q-Network, better handles the loss of information than does DQN.
- Dual Q-network [5], where different Q-values are used to select and to evaluate an action by using two different DQN with different weights. This helps to avoid overestimation of Q-values
- DQN with unsupervised auxiliary tasks can be used to improve performance on DQN, as having additional outputs will change the weights learned by the network [6]
- Asynchronous gradient descent for optimization [7] and prioritized replay [8] have also been proven to increase the efficiency on DQN.

All these alterations to the basic DQN have been proved to stabilize or increase performance on Atari games. From our point of view, the use of prioritized replay is a clever idea. In the back propagation step, we believe it should make a lot of difference to take the samples from which we can learn the most instead of sampling randomly. Even if all these alternatives seem good, we decided to implement a DQN and proposed all this related work as future work that could be added to our implementation of DQN for the Pacman.

IV. Dataset and Features

The dataset is obtained while running the Pacman game. A simulator of the game in python was used from [1]. Since we train our methods while running the game, the dataset keeps changing (this is important to keep in mind when doing ablative analysis for feature selection). We have used 3 different layouts of the Pacman game as shown in Figure 1 and Figure 2: `smallGrid`, `mediumGrid` and `mediumClassic`. For approximate Q-learning, we hand-crafted several features from the Pacman game state and did an ablative analysis which is explained in section 6.

V. Methods

We assume that our reinforcement learning task follows a finite Markov Decision Process (MDP). Let us define some important terms:

- State space $\mathcal{S}$: All possible configurations in the game, including the positions of the points, positions of the ghosts, and positions of the player, etc.
- Action space $\mathcal{A}$: A set of all *allowed* actions: {left, right, up, down, or stay}.

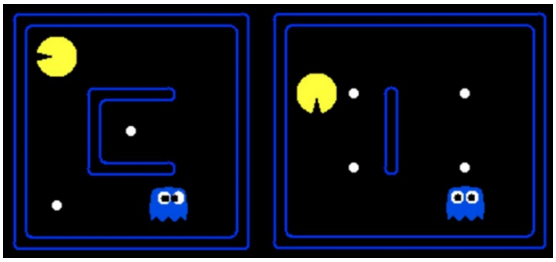


Figure 1: Pacman with `smallGrid` and `mediumGrid` layout

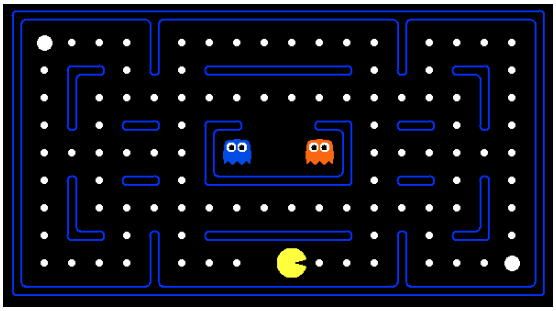


Figure 2: Pacman with `mediumClassic` layout

- Policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$: A mapping from the state space to the action space.

The objective of the player is to take a series of optimal actions to maximize the total reward that it could obtain at the end of the game. Q-values are learned iteratively by updating the current Q-value estimate towards the observed reward plus the max Q-value over all actions in the resulting state. Let us define the Q function as

$$Q(s_{t+1}|s_t, a) = r(s_t, a) + \mathrm{E}(\sum_{k=1}^{\infty} \gamma^k r(s_{t+k}, a_{t+k}))$$

This function defines the Q-value of a particular state s_{t+1}. We choose our action at state s_t such that we maximize the Q-value at the next state s_{t+1}. Thus we define the optimal policy as follows:

$$\hat{\pi}(s_t) = a_t = \arg \max_{a \in \mathcal{A}(s_t)} Q(s_{t+1}|s_t, a)$$

which requires to estimate optimal Q. However, the expectation term in Q is generally expensive to calculate exactly. Therefore we instead could fit the model into a neural network. By introducing a parametrization w (that we will encounter in Approximate Q-learning and DQL), we can try to minimize the following loss function given T training samples:

$$L(\theta) = \frac{1}{T} \sum_{i=1}^{T} (r_i + \gamma \max_{a_i'} Q(s_i', a_i'; \theta) - Q(s_i, a_i; \theta))^2$$

The particular reinforcement learning methods that we use in this project are summarized below.

A. Q-learning (SARSA update)

SARSA is an algorithm for learning a Markov decision process policy, where the Q values are updated according to the following rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \tag{1}$$

B. Approximate Q-learning

In approximate Q-learning (also called Q-learning with function approximation), $Q(s, a)$ is obtained from a linear combination of features $f_i(s, a)$ (including the bias term):

$$Q(s, a; w) = \sum_{i=1}^{n} f_i(s, a) w_i \tag{2}$$

The Pacman agent has to learn the weights for the features extracted from the game states. A feature $f(s, a)$ is defined over state and action pairs, which yields a vector $(f_0(s, a), f_1(s, a), \cdots, f_i(s, a), \cdots f_n(s, a))$ of feature values, being $f_0(s, a) = 1$ the bias term. The weights are updated according to the following rule:

$$w_i \leftarrow w_i + \alpha \cdot (r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \cdot f_i(s, a) \tag{3}$$

C. Deep Q-Learning

Handcrafting features is not very efficient as there is always a possibility we miss out an important feature of the states. Therefore we implement a Convolutional Neural Network (CNN) that can implicitly "extract" the features and output the Q-values for different possible actions in a given state. Neural networks for DQL normally comprise of convolutional layers followed by fully connected hidden layers. In this case $Q(s, a) = Q(s, a; w)$, where $Q(s, a; w)$ is the function approximated by our neural network and w includes the bias terms and weights of the neural network.

Our implementation closely followed the algorithm provided in [2].

VI. Results

We implemented SARSA update, approximate Q-learning and deep Q-learning for PACMAN by building on top of UC Berkley's CS188: Introduction to Artificial Intelligence course's PACMAN [1] implementation which was built by John Denero, Dan Klein, and Pieter Abbeel(http://ai.berkeley.edu).

A. Q-Learning (SARSA)

The QLearning agent learns well when trained on 2000 games for `smallGrid` with one ghost and one point. It shows a 100% win-rate when tested on 100 games. Also during the training phase, we observed that the average reward for 100 games is positive after 1400 games. This shows that even for such a small grid, the agent takes a long time to train and hence, the agent might not be good enough for larger sized

Table I: Ablation Analysis

Component	Avg. Score	Win Rate
Overall System	1608	92%
min. distance scared ghost	1581	90.2%
inv. min. distance active ghost	1583	91.6%
min. distance active ghost	1594	91.8%
min. distance capsule	1591	90.2%
no. scared ghosts 2 steps away	**1545**	92.2%
no. scared ghosts 1 step away	**1438**	91.2%
distance to closest food	**1397**	90%

grids. Our hypothesis was proven right, when we trained the QLearning Agent on `mediumGrid` with one ghost and 4 points and found that agent has a win-rate of 1% after 2000 training games. Just out of curiosity, we increased the number of training games till the average reward in the training set became positive. The agent has an average positive reward after 9700 training games and has a win-rate of 77% when trained on 11000 games. Our experiments here show that a simple Q learning algorithm such as SARSA is definitely not scalable and will fail miserably on a standard Pacman game.

B. Approximate Q-Learning

We hand-crafted several features that we thought would be important including distance to the closest food, number of active ghosts one step away, distance to closest active ghost, inverse of the distance to closest active ghost, distance to closest scared ghost, number of scared ghosts nearby (1 or 2 steps away), minimum distance to a capsule, minimum distance to a scared ghost, activation of number of capsules when ghost is nearby, activation of binary feature when ghost is not nearby and there is food nearby (this means Pacman can focus on "eating food").

We performed an ablative analysis (see Table I) to select the most important features in terms of the average score and the win-rate from 500 training examples and 500 testing examples for each case. Although the win-rate keeps almost the same when we remove features one by one, we notice that the average score reduces when we remove the "scared ghosts 1-step away" and "scared ghosts 2 steps away" features since eating the scared ghost helps boost the rewards significantly. Also having the feature "distance to the closest food" is important since it motivates the Pacman to take the shortest path to eat food. "Active ghost one step away" and the binary "eating food" are not included in the analysis because if we remove any of them the win rate drops to 0. Indeed they are essential for winning a game in terms of teaching the agent to avoid dangerous ghosts and gain rewards.

We can conclude that the best features are: the number of scared, active ghosts one and two steps away, eating food if there are no active ghosts nearby, and the distance to the closest food.

The performance of the Approximate Q Agent was tested on the `mediumGrid` and `mediumClassic` game. We want to point it out that with just 50 episodes of training, the agent has a 100% winrate on the mediumGrid and 87% winrate on the mediumClassic game. Such a drastic improvement in performance shows the advantages of defining explicit features. This is due to the fact that states can share many features, which allows generalization to unvisited states and makes behavior more robust: making similar decisions in similar states; otherwise, the agent needs to have explored each unique game state during the training phase before it can perform well in the test phase. As the state space increases exponentially in terms of the complexity of the game, the size of the training set also has to grow exponentially before the agent starts learning to play.

C. Deep Q-Learning

We use TensorFlow to implement a Convolutional Neural Network (Q-network), to generate the Q-values corresponding to the five possible directions at each state: North, South, East, West, Stop. DQN architecture is shown in Figure 3, and it is composed of three convolutional layers (3x3x8, 3x3x16 and 4x4x32), a flattening layer and a fully connected layer with 256 neurons. All layers use a Relu activation function. In order to contruct our neural network we have used the following concepts:

1) Create **equivalent images** of frames with each pixel representing objects in the Pacman grid (Figure 3). This reduces the image data by a factor of 100 without loosing relevant information for the training. This resulted in a tremendous increase in the training speed!
2) Use **replay memory** to store the last 100000 experiences ($state$, $action$, $reward$, $next\ state$, $legal\ actions\ of\ next\ state$). These are stored in each transition of Pacman game.
3) In order to train the DQN by minimizing the loss we sample a minibatch of size 32 randomly from the replay memory. This helps to avoid correlation between samples.
4) Use an additional **target network** ($\hat{Q}$-network) to generate the target values for Q-network. Once every 100 iterations the target network is updated with the learned parameters from Q-network by minimizing the loss. This helps to reduce oscillations and make the method more stable.

$$L = \frac{1}{T}\sum_{i=1}^{T}(r_i + \gamma \max_{a_i'} \hat{Q}(s_i', a_i') - Q(s_i, a_i))^2$$

It is important to notice that a_i' includes only the legal actions. Thus we had to ensure that we vectorized the maximization over the legal actions to improve the speed. We also performed the gradient clipping to avoid exploding gradients.
5) We use **Epsilon-greedy** strategy for exploration. We gradually reduce ϵ from 1 to 0.1 during training.

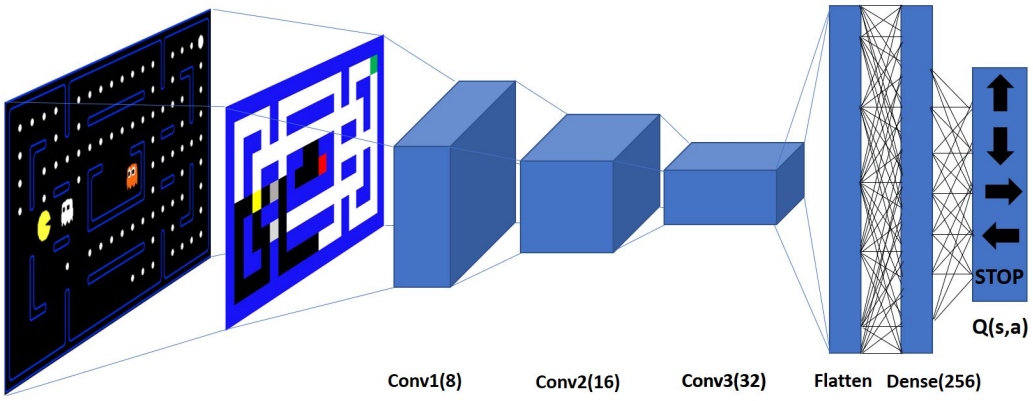


Figure 3: DQN pipeline with an equivalent image, three convolutional layers, a flattening layer and a fully connected hidden layer

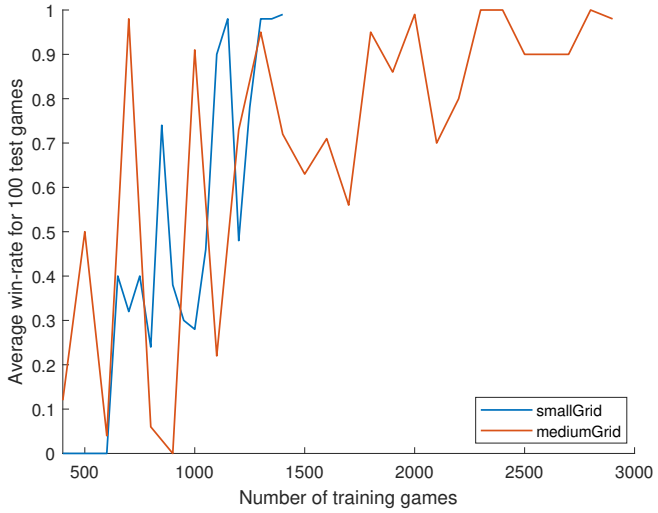
The learning rate used is 0.00025 and reduces to 0.00005 as training goes and we train the DQN in every step of the game. We observe (Figure 4) that DQN is able to win and get high score in `smallGrid` and `mediumGrid` layouts, performing better than SARSA update. Using the equivalent image trick, the training is much faster than using the original frame and we are not losing any information that the original frame could give us. In `mediumClassic` layout, the training takes a really long time even on GPUs. On the 25000 games we trained, the average reward remained negative. We think that around 1,000,000 games will be needed before the agent starts to learn with our current tuning of hyper parameters.

VII. Other attempts

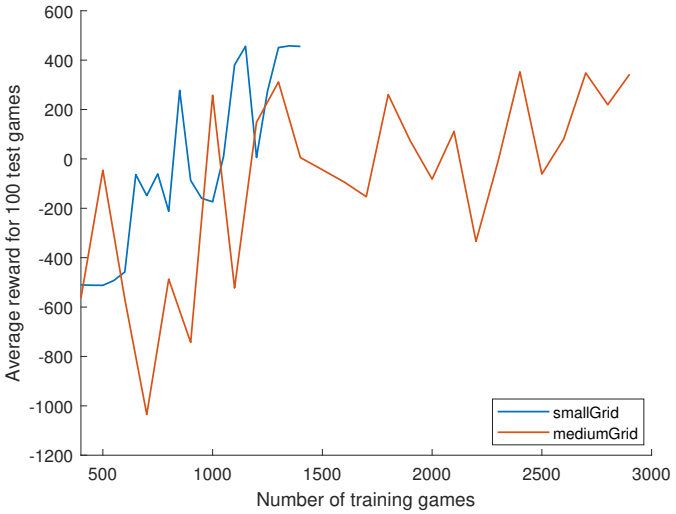
While it is important to know how to make things work, it is also important to know what does not work. We tried a number of things in our implementation of DQN before we reached a configuration that works with reasonable speed and accuracy. This section documents these efforts:

A. Layers of the DQN

What are the best layers for DQN? How many layers should we use? What is the activation function? Kernel size? Stride? From our experience and literature survey, these are some of the questions in Deep Q-learning that are not rigorously answered. Researchers choose these parameters based on experience or by tweaking around till it works well for their purpose. While we are not happy about following this approach, we played around with different layers to find the best one. Most of the architecture we used is similar to the one used in the classical Deep Q-learning paper by Mhin et. al. [3]. We added in max-pooling layers after every convolutional layer but this resulted in poorer performance. To select the number of convolutional layers and update frequency we have performed an analysis for `smallGrid` layout, and use these parameters for other layouts. We observe in Figure 5 that having 3 convolutional layers instead of 2 improves

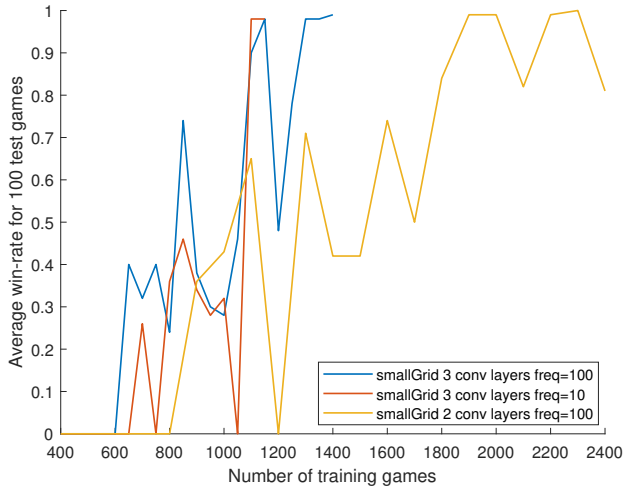


(a) Average win-rate

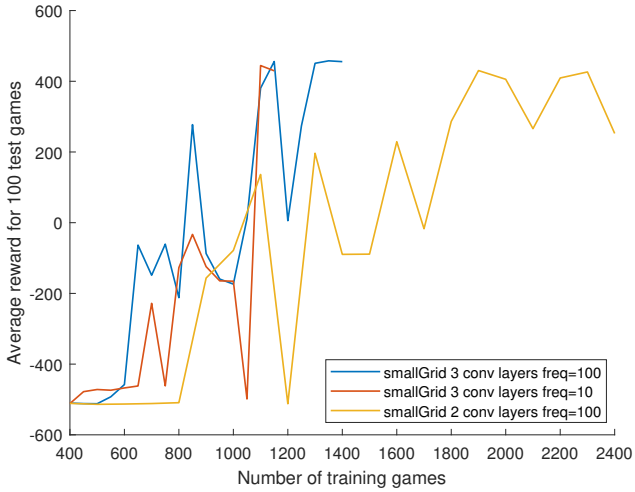


(b) Average score

Figure 4: Average win-rate and score for DQN



(a) Average win-rate for `smallGrid`



(b) Average score for `smallGrid`

Figure 5: Average win-rate and score for using a 3 and 2 convolutional layers and an update frequency of 10 and 100

performance and learning, as well as using high $\hat{Q}$-network update frequencies (100 as opposed to 10).

B. Input to the Neural Network

Initially we tried to feed in the pixel data from the images captured as the game proceeds. This proved to be very cumbersome and expensive. With this limitation, we could not train the network on GPUs. To overcome this problem, we developed equivalent images that can very efficiently represent the image information without the need to capture the images. In this representation, we directly access the positions of Pacman, ghosts, food, capsules and represent them as one pixel in the game space. This drastically improved the training and produced good results in less than 2 hours of training.

C. Storage and access from Replay Memory

We store state space information to the replay memory at every step of a Pacman game and we access information from it at every step of training. Thus, it is essential that we use the apt data structure to efficiently add, remove and sample data. The first implementation of our replay memory class used `numpy` arrays to store data. We later rewrote the class to use the `deque` data structure from the collections class. Note that, `deque` is a high-performance container that implements fast appends and pops [9].

D. Simple Neural Network

Before diving into CNN, we started by implementing a simple neural network with a single neuron using TensorFlow. The features (distance to closest food, number of ghosts one step away, bias and feature indicating if there is food 1 step away and no ghost 1 step away) extracted from (50 games of mediumClassic) of the test phase of the Approximate Q Agent were input to the network and the output was the Q-value. The weights of the features were optimized using Gradient Descent with a small regularization. It was interesting to note that the weights learned by the network were different as compared to the weights from Approximate Q Learning. The results of this simple neural network were comparable to the results of approximate Q-learning.

VIII. Conclusion

From this project we can conclude that the SARSA update for reinforcement learning performs poorly in Pacman game. Approximate Q-learning can improve on SARSA update and the use of intelligent features for Approximate Q-learning works well for small and medium grids. If we don't want to miss any of the important features in the game Deep Q-learning it is a good alternative that can work well, as tested here for small grids. With our efficient representation of the state space the computational cost for DQN was reasonable.

IX. Future work

There are many paths that we would like to explore to increase performance and learning:

- Adding prioritized replay by choosing the samples from which the agent can learn the most [8]
- Adding auxiliary rewards [6]
- Adding asynchronous gradient descent for optimization of the deep neural network [7]
- Adding a LSTM to our DQN in order to make the training more stable [4]
- Implement double-DQN and compare performance with DQN to see if Q-values may be overestimated for Pacman game [5]

Finally we would also like to train the agent for larger Pacman grids.

X. Contributions

Abeynaya: Implemented Q-learning, Approximate Q-learning; Revised and implemented the class replay memory to use `deque` data structure; Generated the equivalent image representation for the state space; Helped Jordi in debugging the Deep Q Network; Contributed to the poster and the report.

Jordi: Implemented feature selection for Approximate Q-learning; Implemented Deep Q-learning from scratch; Implemented the first version of replay memory; Contributed to the poster and the report.

Jing: Performed Ablative analysis for Approximate Q-learning and experimented with different features; Helped debug the codes; Contributed to the poster and the report.

References

[1] Berkeley Pacman Code. $http://ai.berkeley.edu/project_overview.html$.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[3] 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 Volodymyr Mnih, Koray Kavukcuoglu and Demis Hassabis. Human-level control through deep reinforcement learning. *doi:10.1038/nature14236*, 2015.

[4] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[6] Mnih V. Czarnecki W. M. Schaul T. Leibo J. Z. Silver D. Jaderberg, M. and K. Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[7] Mehdi Mirza Alex Graves Tim Harley Timothy P. Lillicrap David Silver Volodymyr Mnih, Adria Puigdomenech Badia and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

[8] Ioannis Antonoglou Tom Schaul, John Quan and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2016.

[9] Python documentation.