Learning to Play Computer Games with Deep Learning and Reinforcement Learning

Interim Report

COMP4801 Final Year Project
UID: 3035207956

Student
Mak Jeffrey Kelvin

Supervisor
Dr. Dirk Schneiders

January 29, 2018

Abstract

With the integration of deep learning into the traditional field of reinforcement learning in the recent decades, the spectrum of applications that artificial intelligence caters is currently very broad. As using AI to play games is a traditional application of reinforcement learning, the project's objective is to implement a deep reinforcement learning agent that can defeat a video game. Since it is often difficult to determine which algorithms are appropriate given the wide selection of state-of-the-art techniques in the discipline, proper comparisons and investigations of the algorithms are a prerequisite to implementing such an agent. As a result, this paper serves as a platform for exploring the possibility and effectiveness of using conventional state-of-the-art methods, such as Deep Q Networks and its variants, such as Double Deep Q Networks, are appropriate for game playing, with Deep Q Networks successful in playing a randomized map, further work in this project is needed in order for a comprehensive view of the discipline. Such work in the near future includes the investigation of the use of deep reinforcement learning on games unreported in the literature, or potential improvement to existing deep reinforcement learning techniques. In spite of the technical difficulties encountered and minor amendments to the project schedule, the project is still currently on schedule, ie. approximately 50% complete.

Acknowledgements

I would like to take this opportunity to thank my supervisor Dr. Dirk Schneiders from the Department of Computer Science for his advice and careful guidance in this project, as well as permission to use the GPU from the department. I would also like to express my gratitude towards Cezar Cazan from Center for Applied English Studies for various suggestions when drafting this paper. Finally, a final thanks to HKU for providing a variety of quiet learning environments for me to write this paper without hindrance.

Contents

List of Figures

List of Tables

1 Abbreviations

AI	Artificial Intelligence
CNN	Convolutional Neural Network
DRL	Deep Reinforcement Learning
DQN	Deep Q Networks
DDQN	Double Deep Q Networks
Duel DQN	Dueling Deep Q Networks
DRQN	Deep Recursive Q Networks
MDP	Markov Decision Process
PER	Prioritized Experience Replay
PIL	Python Image Library
POMDP	Partially Observable Markov Decision Process
RL	Reinforcement Learning

2 Introduction

Reinforcement learning, a subdiscipline of artificial intelligence, is becoming increasingly popular as a method of creating AI agents. Despite of such popularity, the idea of reinforcement learning is not a recent one, and originated from experiments in behavioural psychology relating to how animals learn tasks by perceiving rewards. In the context of computer science, reinforcement learning deals with how an agent learns an optimal policy π^*, ie. a series of actions, that maximizes the expected cumulative reward $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ by interacting with the environment, where r_t is the reward received at time step t and γ is the discount rate.

In order for an agent to learn, the manner in which it interacts with the environment is key to facilitate learning. In the field of reinforcement learning, there are two major approaches to modelling agent-environment interaction in reinforcement learning, namely Markov Decision Process (MDP) and Partially Observable Markov Decision Process (POMDP). In an MDP, the action At taken by the agent is based on only the reward r_t and state s_t observed from the environment at time step t, as illustrated in Figure 1. However, it is assumed that the agent always observes full information of the current state in each time step. To compensate for such a limitation, one can adopt a Partially Observable Markov Decision Process (POMDP) instead, where the agent only observes partial information of its current state. For example, the interaction would be modelled as a MDP for the game Pacman, since the player can see the entire map in Pacman during gameplay. Conversely, a POMDP would be used as a model for interactions in FPS games [5], since the player's view is often obscured by obstacles. This makes certain game information hidden, such as the position of enemies.

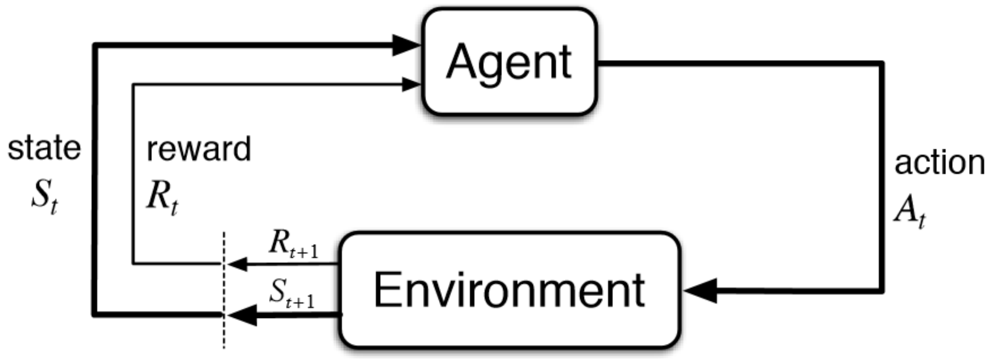


Figure 1: An illustration of the interaction between the agent and the environment through states and actions. Extracted from [1].

One may question the rationale of using reinforcement learning to play games, since the primary purpose of games is for entertainment. Nonetheless, a major reason for this is due to the relatively cheap training cost in a game simulation in comparison to training in the physical world. In particular, training in the real world often requires regular hardware maintenance of physical agents such as robots. For instance, a robot attempting to learn to walk would fall constantly during training, which would become unfavorable due to constant repairing of the robot. Another reason is that replication of agent training is much easier in games than in the real world, since the physical environment is often unpredictable. A consequence of this is that games serve as a convenient platform for benchmarking and comparing various reinforcement learning techniques. Finally, games are traditionally considered as a method of measuring human intelligence. This is especially apparent in chess, where logical reasoning plays a key component for the construction of a winning strategy. As a result, games can be used for comparison between artificial intelligence and human intelligence.

In light of deep reinforcement learning being an active research field and game playing as convenient medium for data collection, this project's objective is to construct a deep reinforcement learning agent in an unexplored game in literature. Nevertheless, proper comparison and investigation of existing deep reinforcement learning algorithms is required. As a result, this paper explore the possibilities of using Deep Q Network and its variants to play the game Pacman through small-scale experiments, while creating software implementations of the aforementioned techniques. With the results in hand, this paper serves as a gateway for an improved understanding on pros and cons of well-studied techniques, as well as their applicability in small-scale problems in game playing.

The remainder of the paper is organized as follows. Chapter 3 surveys existing state-of-the-art deep reinforcement learning algorithms in the literature, and chapter 4 summarizes the list of games that have been investigated in the literature. Chapter 5 details the phases in the project, followed by implementation details of Deep Q Network and its variants and experiments used to evaluate the techniques. Chapter 6 then compares the implemented methods by analyzing experimental results, and concludes by discussing the future work to be completed in Chapter 7.

3 Background

3.1 Deep Q Networks

Similar to tabular Q learning in traditional reinforcement learning, Deep Q Networks [7, 6] also uses Q values in order to predict the expected sum of future discounted rewards. However, a problem with tabular Q learning was the Q values for state-actions pairs are encoded using a table, which uses a large amount of memory in the case of an environment involving a large state-action space. Moreover, each Q-value for different state-action pairs must be learned separately, thereby making development slow. As a result, the replacement of the Q table by a convolutional neural network in the form of a nonlinear function approximator in DQN that minimizes the loss function

$$L_i(\theta_i) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i') - Q(s, a; \theta_i) \right)^2 \right]$$

, where $U(D)$ is the experience replay training minibatch, θ_i is the network parameters for the online neural network and θ_i' is the network parameters for the target neural network. The use of a convolutional neural network greatly reduces the memory required to encode the Q-values, as well as allowing the agent to generalize encoded information learned from the past to unfamiliar scenarios.

Apart from the use of a convolutional neural network, a prominent feature in DQN is the use of experience replay. Such a concept originated from the observed hippocampus activity in the brain. In particular, experience replay involves the use of an experience replay buffer, where the agent's past experience is stored in the form of experience tuples $e_t = (s_t, a_t, r_t, s_{t+1})$. During gameplay the agent learns from past experience by sampling mini-batches from the experience replay buffer in order to train the CNN. Such sampling allows for increased use of past memory when compared to the on-policy variant, which reduces the time taken to learn a policy. Moreover, experience replay prevents the CNN from being trained on consecutive experience tuples, which could cause convergence to a suboptimal policy [7].

To solve the exploration-exploitation dilemma, the DQN algorithm uses an epsilon-greedy algorithm. In particular, the agent picks actions uniformly at random with probability ϵ and greedily with probability $1 - \epsilon$. To encourage exploration during the start of training and exploitation near the end of training, epsilon is set to 1 and linearly annealed to 0.1 over training games [7, 1].

To improve the training of DQN, the original paper proposed several implementation tricks used during coding [7]. First, the raw image is preprocessed in order to reduce the size of the CNN while still retaining sufficient information to solve the desired task. Various techniques can be used for preprocessing, such as downscaling of images and conversion of RGB image to grayscale. Owing to the large variation in reward ranges between different Atari 2600 games, the original papers describes clipping rewards to the range $[-1, 1]$ in order to limit the training loss. Lastly, a target network is used in order to stabilize the Q value during training [7].

An advantage of Deep Q Networks is that minimal prior game-specific information can be used to develop an end-to-end framework, where the input and output of the network would be the preprocessed image from the game and the Q values of the state-action pairs of the current state respectively. This idea is also reinforced by the fact that the agent only learns by playing the game itself. In particular, the agent begins with minimal information about the game's environment (eg. controller inputs), and training data is generated and learned by the agent as training occurs. The use of an end-to-end framework is unlike previous approaches, where game-specific feature extraction to construct an internal grid world representation of the game state coupled with tabular Q learning was required [9] in order for learning to occur, which could limit the policies that the agent could develop.

3.2 Variants of Deep Q Networks

After the development of DQN in 2012, many variations were developed subsequently. Some of the significant variations are detailed in the following subsections. Note that the improvements can be combined together, since they each focus on a different part of the algorithm.

3.2.1 Double DQN

One limitation of DQN was that Q values were often overestimated during training, often resulting in overly optimistic policies. To remedy this situation, Double DQN, ie. DDQN, [10] was introduced where the target value for the CNN to train with was modified to

$$r + \gamma \max_{a'} Q(s', \operatorname*{argmax}_{a} Q(s', a; \theta_i); \theta_i')$$

Such modification increases the stability of the Q values during learning, thus allowing the algorithm to attain improved policies, as shown by the substantially better scores attained in various Atari 2600 games [10].

3.2.2 Dueling Network architecture

Another limitation of DQN is that Q values for different actions for the same state are learned separately. To allow the network to generalize learning of state value across state-action pairs with the same states, one must consider the following definition of the Q value function

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$$

where V is the value function, A is the advantage function, and θ, α, and β are parameters of the dueling CNN. However, the paper describes the use of such an equation for estimating Q values as inefficient, since the value and advantage functions developed are not unique to the problem. To fix this issue, the paper proposes to use the definition of the Q value function

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \max_{a' \in |A|} A(s, a'; \theta, \alpha) \right)$$

instead [2]. The use of the above equation then leads to the modification of the CNN architecture as shown in Figure 2, which allows the dueling variant of DQN, ie. Duel DQN, to learn the two functions separately. When compared to DDQN, the dueling variant of DQN outperforms it in 46 out of 57 Atari 2600 games [2]. It is also mentioned that the dueling variant of DQN with PER outperforms DDQN significantly.

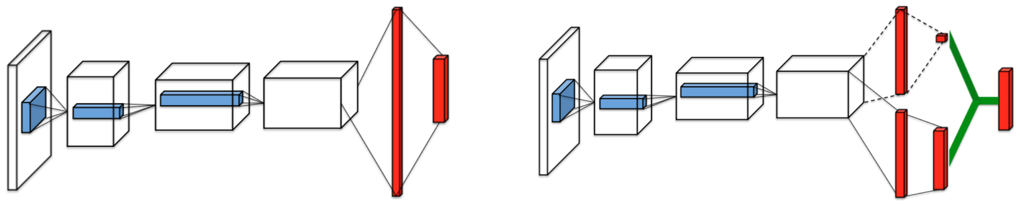


Figure 2: Image extracted from [2]. CNN architecture used in DQN (left) and Dueling DQN (right). Note that the convolutional layers are followed by two separate fully connected layers, which splits the network into two parts, and leads to the value function and the advantage function as outputs of the CNN. The Q function is reconstructed from the the value function and advantage function at the end of the network.

3.2.3 Prioritized Experience Replay

At the end of [6] and [7], Minh et al. described that it may be possible to improve the algorithm via prioritized sweeping, where training would be biased towards significant events. Such idea was subsequently further developed by Schaul et al. [11], where two types of prioritized experience replay (PER) was investigated, namely rank-prioritized PER and proportional-prioritized PER. In particular, prioritized experience replay involves a non-uniform distribution when sampling for experience tuples, where the probability of sampling experience tuple i is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Where in proportional prioritization $p_i = |\delta_i| + \epsilon$, where δ is the training error and epsilon is a small constant, and in rank-based prioritization $p_i = \frac{1}{rank(i)}$, where $rank(i)$ is the rank of i when sorted by $|delta_i|$ in descending order. Results from the paper showed that DQN with PER outperformed DQN in 41 out of 49 Atari 2600 games [11].

3.2.4 Count-Based Exploration and Intrinsic Motivation

Another drawback of using DQN to play games is that the algorithm requires frequent gratification in order to learn efficiently, meaning that such a limitation becomes problematic in games such as Montezuma's Revenge, where rewards are sparse. One solution to this problem is the use of pseudo-counts and bonus rewards with DQN in order to increase the agent's curiosity about the environment and encourage in-depth explorations. Bellemare et al. (2016) showed that the agent with exploration bonus was able to explore 15 rooms in Montezuma's Revenge during training, as compared to 2 rooms for an agent without exploration bonus [12].

3.2.5 Deep Recurrent Q Network

A requirement for the use of MDP for modelling the agent-environment interaction is that the agent must be able to observe all information about the current state. However, this requirement is often not satisfied in games such as Doom where partial information is observed at each frame, resulting POMDP being used to model the agent-environment interaction instead. In order to preserve state information across states, Hausknecht and Stone [13] proposed to attach a Long Short Term Memory (LSTM) [14] to the end of DQN's architecture in order to make the neural network recurrent. Although results showed no significant improvement over DQN in Atari 2600 games if stacked frames are used as input to the neural network, Deep Recurrent Q Networks (DRQN) are still able to play Atari 2600 games even in the presence of a flickering screen [13], and is effective for playing first player shooting (FPS) games such as Doom [5].

3.2.6 Other variants of Deep Q Network

Apart from the variants above, there exists many modifications that can be found in recent literature regarding improvement on the DQN algorithm. For example, a Bayesian exploration approach can be taken by using the uncertainty Bellman equation (UBE) to obtain a tight bound for the true Q value interval. Together with the use of Thompson sampling, the paper reports better performance for UBE in 51 out of 57 games when compared to that of DQN [15]. Another example is the use of optimality tightening through constrained optimization in order for the network to converge faster to a policy [16].

3.3 Other Deep Reinforcement Algorithms

Apart from Deep Q Networks and its variants, other deep reinforcement algorithms have also been developed and experimented in the context of games. For example, A3C [17] and ACER [18] are actor-critic algorithms that were shown have comparable performance to DQN. Policy gradient algorithms, where the network outputs action probabilities instead of Q values, such as DDPG [19], TRPO [20] and PPO [21], can also be used to play Atari 2600 games. In the context of board games, AlphaGo was able to achieve superhuman performance either with [8] or without [22] the presence of prior training data in the game Go through a Monte Carlo Tree Search-based approach.

4 Related Works

The history of using reinforcement learning and neural networks to play games dates back to the 1990's, where agents were being developed to play traditional board games such as Backgammon and chess. when TD-Gammon [9] was developed in order to play backgammon through the use of temporal difference algorithm and a multilayer perceptron network. The incorporation of handcrafted features further improved TD-Gammon's performance. However, progress in the field stagnated subsequently due to lack of success in replicating similar performances in other board games [1]. It was not until the development that a wider variety of games could be played, notably video games such as those in Atari 2600, through the use of Deep Q Networks developed by Google DeepMind [7, 6]. The concept of agents playing highly complex video games was furthered by using a CNN-LSTM network architecture to play Doom, an FPS game. The recent spark in deep reinforcement learning inspired the development in many video games, including the popular games Super Mario world and Flappy Bird. Open-source APIs for games such as Minecraft [23] and Starcraft II [24] were also developed to encourage research on the use of deep reinforcement learning in such open-ended games. Finally, OpenAI Gym [25] provides a simple platform for playing a wide variety of games or simulations, and is often used for establishing baselines for comparing new algorithms against.

5 Methodology

The project is broadly separated into two phases, namely an exploration phase and an implementation phase. In particular, the objective of the exploration phase is to construct one or more deep reinforcement learning agents that can beat the game Pacman. Various RL methods are to be used in this phase, including but not limited to Deep Q Networks and its variants. The results gathered from the selected implemented algorithms are to be evaluated against benchmark algorithms, namely traditional AI agents, ie. Minimax and Expectimax.

The implementation phase involves either selection and implementation of a deep reinforcement learning algorithm on a game with higher complexity or proposing an improvement to an existing deep reinforcement learning algorithm. Similar to the exploration phase, the results of the implementation will be compared against benchmarks algorithms. Further details on the approach taken in this phase will depend on results gathered from the first phase. As a result, the rest of the section deals with details relating to solely the exploration phase.

5.1 Game

Pacman, a popular arcade game, is the game studied in the exploration phase of this project. In particular, the game consists of a map containing dots, capsule, Pacman, and one or more ghosts (see Figure 3), and the objective of the game is to consume all the dots. A score is counted throughout the game, and is modulated as Pacman receives positive and negative rewards (see Table 1). This game was first studied, as the associated game implementation is open-sourced by UC Berkeley, meaning that implementing deep reinforcement algorithms would not be too challenging [3]. In terms of writing code, the game implementation also allows for map customization and control over behavior of both Pacman and ghosts, thus providing a convenient platform for algorithm testing on custom designed maps of varying difficulty.

Table 1: Score modification for various events in the game Pacman. Ghosts become scared for 40 moves when Pacman eats a capsule.

Event	Score
Win	+500
Lose	-500
Eats dot	+10
Eats capsule	+10
Collides with scared ghost	+200
Each turn	-1

For the current phase, algorithmic implementations are tested on a simple map named 506Pacman (see Figure 4). In particular, the map consists of Pacman, one dot and one ghost in a 3×3 grid, and the sprite positions are randomized uniformly each time the map is initialized. Specifically, the map's name originates from the fact that there are 506 possible game states, with 72 accessible states in one game. One of the main motivation for using this map is its randomness, which helps to demonstrate whether the algorithm has performed generalized learning, as opposed to rote memorization of the path to a win the game. Furthermore, the implementations are also tested on a simple realistic map named smallGrid, as shown in Figure 5 in order to see whether the agent is also able to successfully learn in this map as well. The paper starts with 506Pacman and smallGrid, namely small maps, in order to make training of the reinforcement learning algorithms feasible in a reasonable amount of time.

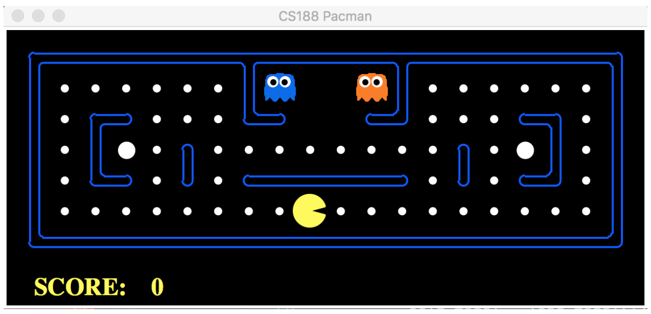


Figure 3: A Pacman map consisting of all game elements. Pacman, represented by a yellow circle with a mouth, is enclosed by walls, as shown in a blue outline. The ghosts are represented by characters with two googly eyes. The dots and capsules are represented by small and large white dots. Image produced in the Pacman software by UC Berkeley[3].

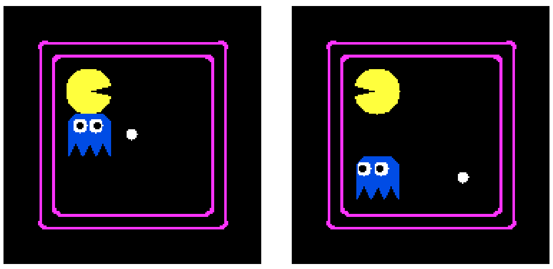


Figure 4: Examples of start game states for 506Pacman, a 3×3 map consisting of Pacman, one dot and one ghost that are randomly placed. Image produced in the Pacman software by UC Berkeley[3].

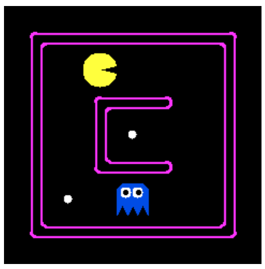


Figure 5: Map of smallGrid. The map consists of pacman, one ghost, two dots and walls in a 7×7 map. Image produced in the Pacman software by UC Berkeley[3].

5.2 Reinforcement Learning Algorithms

Since deep reinforcement learning is currently an active research field, there is a wide selection of algorithms to choose from. As a result, the current phase focuses on simple, yet effective methods that can solve Pacman. Since Deep Q Networks and its variants, ie. Double Deep Q Network, Dueling Deep Q Network, and Deep Q Networks with prioritized experience replay, are well studied in deep reinforcement learning, these algorithms are implemented first. In particular, they are implemented as detailed in [6, 7], [10], [2] and [11] respectively. Due to time limitations, this paper only considers proportional-based prioritized experience replay, as results from Schaul et al. indicated that the two variants of PER had similar performance [11].

5.2.1 Software and Hardware Requirements

Algorithms used in this paper are implemented in Python in view of its conciseness. In addition, neural networks are implemented either in Keras [26] with Tensorflow [27] backend, since Keras provides a platform for convenient and efficient prototyping and experimentation. Since the experiments are conducted in a small scale, they are run on a 2.8 GHz Intel Core i5 CPU on a Macbook Pro if possible. If the time taken to run the experiment is too long, an NVIDIA GPU can be used to speed up training.

Since the original Pacman software does not offer an API for capturing images, an API must be written in order to run the previously described algorithms. There are two approaches for capturing each frame of gameplay in Pacman, namely through direct screen capture by ImageGrab or to redraw the frame through Python Image LIbrary (PIL). Although ImageGrab contains a simple API, it does not exactly capture the correct area properly. This is important since only pixels within the Pacman should be fed into the program. Moreover, using PIL for the frame capture implementation allows training to be done in the absence of a GUI. As a result, this paper adopts the use of PIL for capturing gameplay frames in Python as opposed to ImageGrab.

5.2.2 Neural Network

In light of the effectiveness of the neural network used by Minh. et. al. in playing a wide range of Atari 2600 games[6, 7], similar architectures are adopted for DQN and its variants, as detailed in Table 2. Since the input image to the network has size 32×32, the filter size for the last convolutional layer is adjusted to size 2. Similar to the original paper, the RGB input image is converted into a grayscale image before feeding it into the network to reduce the size of the input layer. The parameter weights of the neural networks are also initialized uniformly at random with zero bias in order to break the symmetry of the networks and to increase the number of features learned during training.

Table 2: Description of the convolutional neural network used in the DQN, DDQN, Duel DQN and DQN with PER implementation for Pacman. The input of the network is an array of 32×32 preprocessed grayscale images for the three implementations, and the output of the network is an array, where each element consists of five Q-values, one for each possible action from the input image's state. The network is trained using Adam [4] as the optimizer, where the huber loss is minimized. The total number of trainable parameters in this network is 87205.

Layer	Input	Filter Size	Stride	No. of filters/Nodes	Activation	Output	No. of parameters
conv2d_1 (input)	(#samples,32,32,1)	8 x 8	4	32	Relu	(#sample, 7, 7, 32)	2080
conv2d_2	(#samples,7,7,32)	4 x 4	2	64	Relu	(#samples,2,2,32)	32832
conv2d_3	(#samples,2,2,32)	2 x 2	1	64	Relu	(#samples,1,1,64)	16448
flatten_1	(#samples,1,1,64)	-	-	-	-	(#samples, 64)	0
dense_1	(#samples,512)	-	-	512	Relu	(#samples, 512)	33280
dense_2 (output)	(#samples, 512)	-	-	5	-	(#samples, 5)	2565

5.2.3 Deep Q Networks and Its Variants

Since DQN, DDQN, Dueling DQN and DQN with PER are very similar value-based RL methods, they are implemented using the same training hyperparameter for ease of implementation. In particular, Table 3 shows the hand-optimized hyperparameters used in DQN and DDQN. Similar to the original publications [7, 6, 10], several trick are used to accelerate network training for both algorithms. Firstly, an experience replay buffer is used to store the last 128 steps of gameplay in order to decorrelate consecutive training data when updating the network. Such storage of game history is especially useful, since training correlated experiences induces a large variance when training a neural network [7]. Secondly, the rewards for each step is clipped to the range [-10,10] as a form of normalization, unlike the original paper, as it appears that clipping the reward to [-1,1] likely hinders the algorithms from distinguishing between the magnitude of rewards when a Pacman dies and when Pacman makes a move. Lastly, the gradient during training is clipped to [-1,1] in order to stabilize network training [7].

To train the network, Adam [4] is used as the neural network's optimizer, and to facilitate convergence of Q values, and the Huber loss function provided by Tensorflow is used as the training loss function in order to clip the gradient of the network.

Table 3: List of DQN training hyperparameters and their values in Pacman

Hyperparameter	Value
No. of frames per input	1
Experience buffer size	2000
Batch size	256
Initial training epsilon	1
Final training epsilon	0.1
No. of exploration frames	2000
Discount factor	0.99
Target network update frequency	20
Replay start size	100
Optimizer learning rate	0.00025

5.3 Benchmark Algorithms

In this experiment, three baseline algorithms are chosen, namely Minimax, Expectimax, and pacmanDQN agent. In particular, minimax and expectimax agents are traditional optimal AI algorithms that have access to game state information. The algorithmic implementations will also be compared against an existing implementation of DQN in Pacman that uses feature extraction in order to confirm whether an end-to-end architecture is better than the use of feature extraction for learning in DQN [28]. The establishment of these benchmark algorithms serve as a medium for showing the optimality and learning progress that the deep reinforcement learning techniques have achieved.

5.4 Data Collection

All four implementations were trained for 7000 games for 506Pacman, where the epsilon value was linearly decreased from 1 to 0.1 over 2000 frames and kept at 0.1 for the rest of training. After training, the trained agents were tested on the same map for 500 games but with a constant test epsilon value of 0.05 for 506Pacman. The epsilon value was set to 0.05 in order to prevent the CNN from overfitting. To vary the map difficulty, the agent is trained and tested with default ghost and Minimax ghost behaviour. In particular, the default ghost makes a random move with probability 20% and moves towards Pacman otherwise, and the Minimax ghost choses moves based on the Minimax algorithm.

Apart from 506Pacman, In order to compare the effect of using map features or raw greyscale image as input data to the CNN on the agent's performance, the two versions of DQN were trained for 5000 games and tested for 1000 games on smallGrid using the same training and testing parameters as previously described.

In terms of metrics measured for both maps, the average score, percentage of games won and maximum Q values were measured over the course of training and testing. In addition, average scores and win percentages are measured as an indication of whether the agent was successful in completing the map. To gain insight on the policy learned, the maximum Q value obtained for the map shown in Figure 6 are recorded.

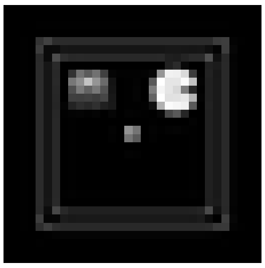


Figure 6: 32 × 32 greyscale image used to observe change in Q values during training. Image produced in the Pacman software by UC Berkeley[3].

6 Results and Discussion

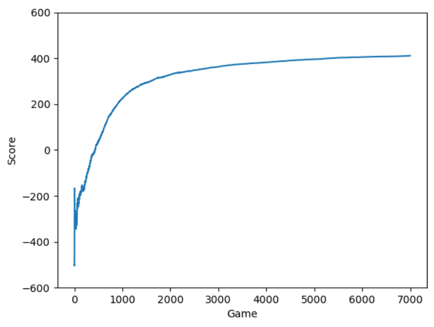
6.1 Initial Findings

Figures 7 and 8 show results from training in 506Pacman using default and minimax ghost AI respective, Figure 9 shows the results obtained from training in smallGrid, Figure 10 show the training losses for training in the two maps and Figure 11 shows the training loss when using DQN with proportional-based prioritization. In particular, the metrics are plotted either against training games or training epochs, where a single epoch corresponds to one neural network batch update in the algorithms.

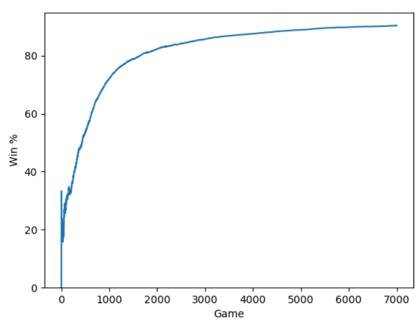
From Figures 7(a), 7(b), 8(a), and 8(b), a gradual increase in average score and win rate is observed, indicating that the agent does perform learning during training. This can be confirmed in Figures 7(c) and 8(c), where the maximum Q value converges to a value close to 10 during training. During training, DQN won 6330 and 6141 out of 7000 games, ie. a win rate of 90.4% and 87.7%, in 506Pacman with default and Minimax ghost behaviour respectively. The amount of training steps taken were around 16000-17000, taking approximately 3-4 hours to train. During testing, DQN won 455 and 451 out of 500 games (91% win rate) for default and ghost AI repectively. These scores are lower than the win rate of a Minimax or Expectimax agent in 506Pacman, which achieved over 98% win rate for both ghost behaviors, implying that the policy attained by DQN is not optimal.

It is interesting to note that by analysing the Q values obtained with the image in Figure 6 as input, it is observed that the action corresponding to the maximum Q value in 7(d) and 8(d) is south, which is consistent with what a human player would do, since moving south in the map would increase Pacman's distance from the ghost and decrease Pacman's distance from the dot.

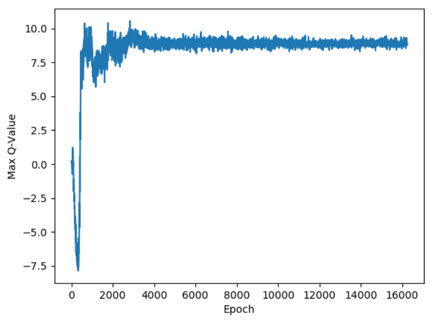
In addition, Figure 9 shows the results obtained from running DQN in smallGrid. Similar to 506Pacman, Figures 8(a) and 8(b) show an increase in average score and win rate respectively, which also shows that the agent has converged to a winning strategy in smallGrid. In particular, 4007 out of 5000 games (80.4%) was achieved during training, and 891 out of 1000 games (89.1%) was achieved during testing. This is in contrast to the results of the feature extraction DQN



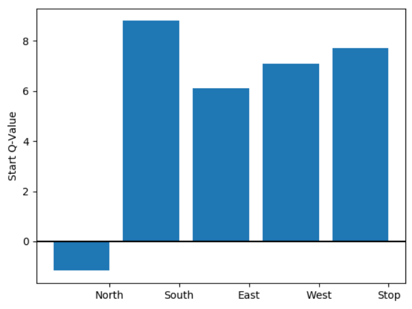
(a) Average scores during training



(b) Win percentage during training

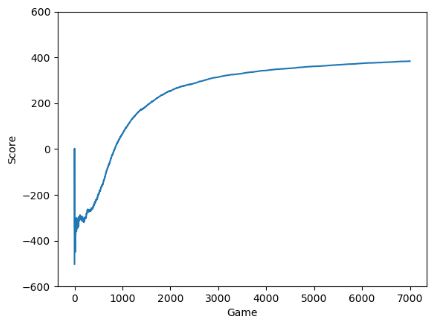


(c) max Q values for a specific start state during training

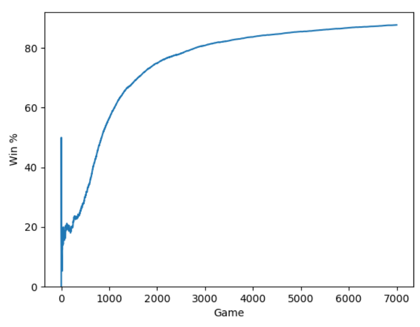


(d) start Q values from map in Figure 6

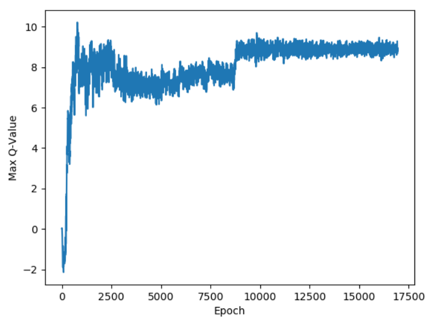
Figure 7: Deep Q Networks training results with default ghost AI in 506Pacman



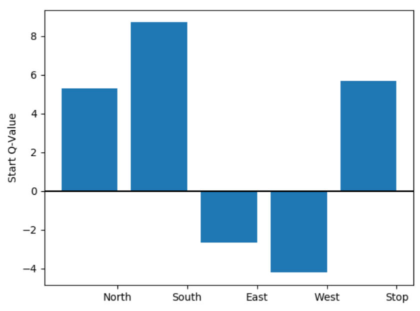
(a) Average scores during training



(b) Win percentage during training



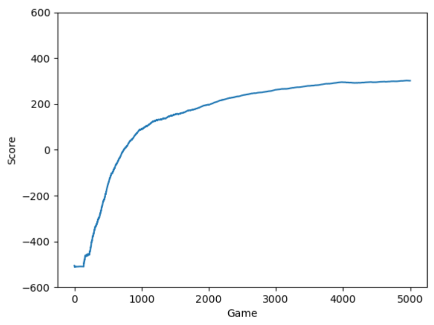
(c) max Q values for a specific start state during training



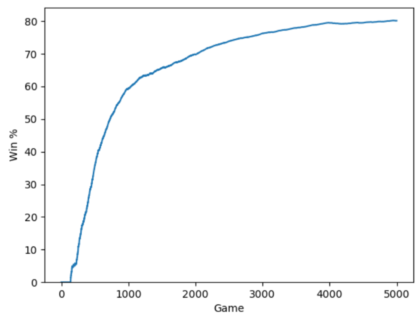
(d) Last training game's start Q values

Figure 8: Deep Q Networks training results with minimax ghost AI in 506Pacman

implementation from [28], where 1267 out of 5000 and 499/1000 games were won during training and testing respectively. From the results above, it can be concluded that . The explanation for such an observation may be due to extra information encoded by the input pixel image that is not conveyed via feature extraction, such as the direction that Pacman is facing, thereby allowing better policies to be made.



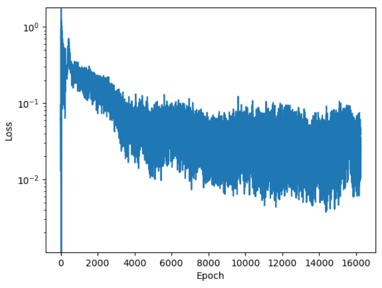
(a) Average scores during training



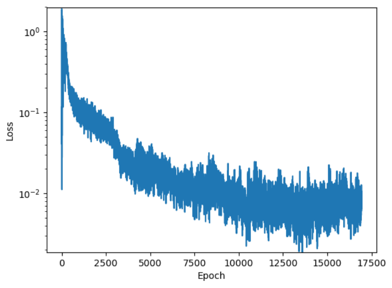
(b) Win percentage during training

Figure 9: Deep Q Networks training results with default ghost AI in 506Pacman

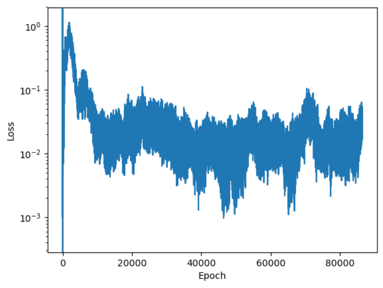
Figure 10(a), 10(b) and 10(c) show the training losses during training in smallGrid and 506Pacman. All three figures show a decline in training losses, indicating learning performed by the agent. However, it is worthwhile to note that the decline stops with training loss at around 10^{-2}, indicating that the agent does not converge to an optimal policy. The training loss not converging to zero is also an indication of the CNN either overfitting or under-fitting. As a result, further hyper-parameter tuning may be required in order to further improve the agent's performance.



(a) Training loss in 506Pacman with default ghost behavior



(b) Training loss in 506Pacman with default ghost behavior



(c) Training loss in smallGrid

Figure 10: Deep Q Networks training losses in Pacman

Apart from performing the aforementioned experiments, there were many preliminary experimentation performed during the process. In particular, there were also attempts to use Double DQN and Duel DQN, as well as combination of the DQN variants in playing 506Pacman. However, the subsequent results indicated that the agent often scored below -550 after training for approximately 1000 games, indicating that the agent learned how to avoid the ghost but did not learn to eat the dot in order to win the game. This may be likely due to the CNN having trouble distinguishing between the ghost and dot sprite. It is also observed that the use of prioritized experience replay leads to a faster reduction in training loss, as well as a substantial decrease in training loss to the magnitude of 10^{-6}, as shown in Figure 11. Moreover, testing of the resulting agent results in a 96% win rate, which agrees with the. Therefore, it can be concluded that DQN with proportional-based PER leads to a better agent than DQN, which is in line with the literature [11].

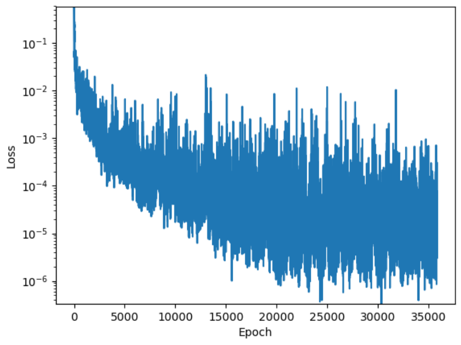


Figure 11: Training loss of DQN with proporitional-based PER in 506Pacman for 6000 games

6.2 Limitation and Difficulties encountered

Several difficulties were encountered during implementation and experimentation of the algorithms used in this paper. In particular, it was challenging to optimize the algorithms, since there seems to be no fast and systematic method of optimizing the hyperparameters for the above algorithms. This was especially the case for tuning epsilon since a balance between exploration and exploitation is required in order for the reinforcement agents to learn. Since hand optimization of hyperparameters is expensive and time consuming, the project progress was slowed by a significant amount. Since little literature regarding this issue has been read, this problem may have to be addressed in the future through further literature search.

Apart from hyperparameter tuning, another problem was that the values of the hyperparameters were dependent on the map being played. As a result, only two map was studied thus far. Nevertheless, similar experiments involving much larger maps with more ghosts are to be performed at a later stage in the exploration phase.

In terms of limitations of the algorithms, one of the problems is that an increase the state-action space causes the deep reinforcement learning agent to take longer to converge to an optimal policy. Such a characteristic of deep reinforcement learning algorithms can be problematic, due to time and resource constraints. Even when a GPU is used, training time can still be lengthy due to image drawing and processing being the bottleneck. As a result, further investigations regarding such bottleneck may be required in order to minimize the use of computational resource and time.

Numerous technical problems appeared when implementing the algorithms. One such issue was that the provided Pacman software did not provide a direct API method for observing the visual output of a game state. As a result, unexpected time was spent on implementing such a feature in addition to the original software, which slowed the project's progress by a small amount.

Another challenge was on the mechanism when handling illegal moves. Since the Pacman API does not accept illegal moves, the algorithms were programmed so that any illegal action would cause Pacman to execute the stop action. Nonetheless, the intended action that Pacman wanted to take is recorded in the experience so that the action taken was propagated back to the associated nodes in the network.

7 Future Work

In view of the major obstacle being the inability for the deep reinforcement learning agent to approach 100% win rate, the next step of the project involves the modification of the existing implementation to mitigate this issue. For example, additional experiments may be devised in order to determine the impact of neural architecture on DQN for a better understanding of the optimal architecture to use in deep reinforcement learning. Apart from tackling with the problems previously mentioned, various benchmark algorithms may also be explored in order to better compare experimental results obtained. Lastly, experiments conducted in this paper will also be repeated with Pacman maps of higher complexity in order to better compare the computational differences between the studied deep reinforcement learning techniques.

Since the survey of deep reinforcement learning algorithms on different games is still incomplete, it is rather difficult to plan in a detailed manner in the exploration phase. Nevertheless, it is projected with a high probability that the project will do some kind of AI agent implementation in either Minecraft or StarCraft, since Microsoft and Deepmind have respectively provided an API for programmers to create agents in these games as part of an initiative to encourage community implementation and exploration of deep reinforcement learning through such games [23, 24].

The current project progress is on schedule, and is summarized in Table 4, where bold items represent strict deadlines and deliverables. Note that significant amendments were made over the course of the first stage of the project in view of the change of the project's scope to narrow the project's scope to temporarily consider DQN and its variants. In terms of met deadlines, all items up to Jan 29, 2017 are currently completed.

Table 4: Project Schedule

Date	Task
Early October 2017	Preliminary Research • Perform preliminary literature research on existing games that use reinforcement learning and deep learning techniques. • Experiment with various classic RL methods, including policy iteration, value iteration and tabular Q-Learning
October 1, 2017	**Phase 1 Deliverables (Inception): Project Scheme, Detailed Project Plan**
Mid October 2017	Stage 1: Algorithmic Exploration • Implement and evaluate the performance of DQN in Pacman • Read up on variants of DQN algorithm
October 31 2017	• Read up on variants of DQN algorithm • Implement DDQN, Duel DQN and DQN with proportional-based PER
Mid November 2017	• Implement the 506Pacman map • Experiment with DRL algorithms in different Pacman maps
December 2017	• Train and Test DQN and its variants on 506Pacman and establish results
January 22, 2018	**First Presentation**
January 29, 2018	**Phase 2 Deliverables (Elaboration): Pacman RL Implementations, Interim Report**
Jan –April 2018	Stage 2: RL implementation and Optimization • Investigate Atari 2600 games (eg. Breakout and Pong) • Pick a game unexplored in the literature to play with • Improve upon existing reinforcement learning algorithms
April 15, 2018	**Phase 3 (Construction): Game RL Implementation, Final Report**
April 16 – 20, 2018	**Final Presentation**
May 2, 2018	**Project Exhibition**

8 Conclusion

This project aims to explore the applicability of deep reinforcement learning methods in the context of playing games. Three algorithms, namely DQN and its variants, were implemented, trained and tested through small-scale experiments in small Pacman maps. In particular, it was shown that DQN was successful in playing OnePacman through the development of human-like decisions during training in 506Pacman and smallGrid. In addition, it was determined that the current paper's DQN implementation is not optimal, and further work is required to determine whether it can be further imrproved. In terms of network architecture, it was confirmed that the use of an end-to-end CNN architecture is more advantageous than the use of feature extraction and a CNN in the context of smallGrid. Several observations were also made, leading to additional motivation to improve the use of algorithms such as DDQN and DuelDQN in Pacman. Despite of the limited selection of techniques and games explored in this paper, steps are to be taken in order to explore other techniques and tricks that ameliorate learning of a task in a wide variety of games, such as Atari 2600 games, in the exploration phase. This is an important part of the project in view of the rapid progress in the field, which could easily render previous surveys outdated. With proper comparisons and investigation completed in the future, the implementation phase will subsequently use the results to make informed decisions on the selection of algorithms to use and apply in a sophisticated video game or to gain insight on drawbacks of existing algorithms and subsequently improvements to them.

9 References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction.* MIT Press, 1998.

[2] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[3] (2017, Oct). [Online]. Available: http://ai.berkeley.edu/reinforcement.html

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[5] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," *CoRR*, vol. abs/1609.05521, 2016. [Online]. Available: http://arxiv.org/abs/1609.05521

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 02 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Work-*

shop, 2013.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, January 2016.

[9] G. Tesauro, "Temporal difference learning and td-gammon," *Commun. ACM*, vol. 38, no. 3, pp. 58–68, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/203330.203343

[10] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, vol. abs/1511.05952, 2015. [Online]. Available: http://arxiv.org/abs/1511.05952

[12] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," *CoRR*, vol. abs/1606.01868, 2016. [Online]. Available: http://arxiv.org/abs/1606.01868

[13] M. J. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *CoRR*, vol. abs/1507.06527, 2015. [Online]. Available: http://arxiv.org/abs/1507.06527

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[15] B. O'Donoghue, I. Osband, R. Munos, and V. Mnih, "The uncertainty bellman equation and exploration," *CoRR*, vol. abs/1709.05380, 2017. [Online]. Available: http://arxiv.org/abs/1709.05380

[16] F. S. He, Y. Liu, A. G. Schwing, and J. Peng, "Learning to play in a day: Faster deep reinforcement learning by optimality tightening," *CoRR*, vol. abs/1611.01606, 2016. [Online]. Available: http://arxiv.org/abs/1611.01606

[17] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[18] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *CoRR*, vol. abs/1611.01224, 2016. [Online]. Available: http://arxiv.org/abs/1611.01224

[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015. [Online]. Available: http://arxiv.org/abs/1509.02971

[20] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust

region policy optimization," *CoRR*, vol. abs/1502.05477, 2015. [Online]. Available: http://arxiv.org/abs/1502.05477

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[22] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354 EP –, 10 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[23] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell, "The malmo platform for artificial intelligence experimentation," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, pp. 4246–4247. [Online]. Available: http://www.ijcai.org/Abstract/16/643

[24] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. P. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "Starcraft II: A new challenge for reinforcement learning," *CoRR*, vol. abs/1708.04782, 2017. [Online]. Available: http://arxiv.org/abs/1708.04782

[25] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[26] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[28] T. van der Ouderaa, "Deep reinforcement learning in pac-man," 2016.