San Jose State University

SJSU ScholarWorks

Master's Projects

Master's Theses and Graduate Research

Spring 2017

AI for Classic Video Games using Reinforcement Learning

Shivika Sodhi
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
Sodhi, Shivika, "AI for Classic Video Games using Reinforcement Learning" (2017). *Master's Projects*. 538.
DOI: https://doi.org/10.31979/etd.g9s3-czdx
https://scholarworks.sjsu.edu/etd_projects/538

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

AI for Classic Video Games using Reinforcement Learning

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Shivika Sodhi

May 2017

© 2017
Shivika Sodhi
ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

AI for Classic Video Games using Reinforcement Learning

by

Shivika Sodhi

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2017

Dr. Christopher Pollett	Department of Computer Science
Dr. Jenny Lam	Department of Computer Science
Dr. Robert Chun	Department of Computer Science

ABSTRACT

AI for Classic Video Games using Reinforcement Learning

by Shivika Sodhi

Deep reinforcement learning is a technique to teach machines tasks based on trial and error experiences in the way humans learn. In this paper, some preliminary research is done to understand how reinforcement learning and deep learning techniques can be combined to train an agent to play Archon, a classic video game. We compare two methods to estimate a Q function, the function used to compute the best action to take at each point in the game. In the first approach, we used a Q table to store the states and weights of the corresponding actions. In our experiments, this method converged very slowly. Our second approach was similar to that of [1]: We used a convolutional neural network (CNN) to determine a Q function. This deep neural network model successfully learnt to control the Archon player using keyboard event that it generated. We observed that the second approaches Q function converged faster than the first. For the latter method, the neural net was trained only using prediodic screenshots taken while it was playing. Experiments were conducted on a machine that did not have a GPU, so our training was slower as compared to [1].

ACKNOWLEDGMENTS

I would like to thank Dr. Christopher Pollett for his incredible guidance throughout the project. He was not only patient with me through the research phase of the project, but also encouraged me to keep working and helped me cross each hurdle with every passing week. I gained a lot through this project because of his profound knowledge in the field of Artificial Intelligence and the brainstorming white board sessions of solving complex mathematical equations.

I am also grateful to my committee members, Dr. Robert Chun and Dr. Jenny Lam for providing their valuable feedback and guidance.

And lastly, I would like to thank my parents for making my dream of pursuing a master's degree come true.

TABLE OF CONTENTS

CHAPTER

1 Introduction 1

2 Background 4

3 Automating playing archon 7

4 The reinforcement learning algorithm 11

4.1 Challenges faced by RL 12

4.1.1 Credit assignment problem 12

4.1.2 Explore-exploit problem 12

4.2 Markov Decision Process 12

4.3 Discounted Future Reward 14

4.4 Q-learning Algorithm 15

4.5 Experience Replay 17

5 Q-learning with neural networks 18

5.1 Convolutional Neural Network 20

5.2 Network Architecture 21

5.3 The DQN Algorithm 21

6 Experiments 23

6.1 Experiment with the Q-function being implemented as a Q table . 23

6.2 Experiment with Neural Networks 23

7 Conclusion and future work 25

7.1 Conclusion 25

7.2 Future Work 25

LIST OF REFERENCES 26

APPENDIX

A Characters of Archon: Light Side 28

B Characters of Archon: Dark Side 30

LIST OF FIGURES

1 Strategy mode of Archon. 7

2 Fight mode of Archon. 8

3 Select the side to fight from 9

4 Reinforcement learning problem. 11

5 Markov decision process. 13

6 Two approaches of forming a deep Q-network 19

7 Result of Experiment 1 23

8 Result of Experiment 2 24

A.9 Valkyrie 28

A.10 Golem 28

A.11 Unicorn 28

A.12 Djinni 29

A.13 Wizard 29

A.14 Phoenix 29

A.15 Archer 29

A.16 Knight 29

B.17 Banshee 30

B.18 troll 30

B.19 Basilisk 30

B.20 Shapeshifter 31

B.21 Sorceress 31

B.22 Dragon 31
B.23 Manticore 31
B.24 Goblin 31

CHAPTER 1

Introduction

Video games are a wonderful sandbox for researchers in the field of artificial intelligence (AI). They provide simplified, yet interesting worlds in which to test our various techniques for generating decision policies. They often involve more complicated visual and auditory inputs than simple board games and they often involve only partially known or unknown environments at any given point in the game. In this project, we are interested in training AI agents to play video games using deep neural networks.

A deep neural network is another name for a multi-layer neural network. Algorithms to train such networks are called deep learning algorithms. Such algorithms are often more successful if the connections between neurons are restricted in some way. A convolutional neural network, one kind of a deep neural network, places restrictions on neurons motivated by attempts to mimic the layers of neurons in the brain's neocortex. Convolutional neural networks have recently led to improvements in computer vision. The aim of our research is to observe the impact of applying a deep learning algorithm to train an agent to play a classic video game, Archon, using a minimal amount of prior information about the game. Our primary motivation for choosing this topic was the recent success by a team at DeepMind at showing how neural networks could be trained to play video games, using reinforcement learning [1].

Earlier research on reinforcement learning with function approximation [3] had shown that reinforcement learning algorithms with lookup tables easily becomes unstable. Through this research project we attempt to find out if that is the case for Archon as well. Thus, we compare two different input policies of reinforcement learning, neural networks and a sparse table, by supplying them separately as an

input in the training of the agent to the reinforcement learning algorithm.

We chose Archon: The Light and the Dark, as there had not been a successful attempt that we were aware of to use reinforcement learning to play the game. Archon is a classic video game, similar to chess, but with an additional feature that when a player lands on asquare occupied by an opponent piece, the game switches to an arcade-style fight between the two pieces, the winner which taking the square and the losing piece being removed. Training of the agent was done for this fight mode of Archon for both reinforcement learning techniques. In the case of neural network, since the task involves visual pattern detection and image recognition, we chose convolutional neural network (CNN) as the most suitable approach to solve the problem. Hence, the inputs to the neural network are sequences of screenshots of the fight mode, and the output in our case is one of six actions: left, right, up, down, fire and do nothing. The conventional approach is to treat it as a classification problem, i.e., for each game screen the agent must decide what action it needs to take. But for the classification to work efficiently, a lot of training examples are needed, which is very different from real-life scenarios, as we need occasional feedback to know that we made the right decision and consequently figure out the rest ourselves.

This report presents a comparison of a deep Q-network (DQN) agent playing Archon, with another agent which is also based on Q-learning, but using feedback computed by a Q-table, instead of a neural network. The remainder of this report is organized as follows: Chapter 2 talks about previous research done in the field of DQN. Chapter 3 gives a detailed overview of Archon and how a bot programmed in Python was designed to play it. Chapter 4 describes the Q-learning algorithm, a type of reinforcement learning algorithm. It describes technique that we used to train an agent to play Archon. Chapter 5 gives an overview of convolutional neural network algorithms. Chapter 6 presents the experiments performed and gives a comparison of

them. Lastly, Chapter 7 discusses the areas of improvement and future work.

CHAPTER 2

Background

In 2013, a major breakthrough was done in the field of AI by DeepMind, through the introduction of their deep learning model in [1], the first ever, that successfully learnt to play seven different Atari games without changing their architecture or adjusting their learning algorithm. The history of reinforcement learning goes all the way back to 1995, when a successful attempt was made by [4] to teach a program, TD-gammon or Temporal Difference gammon, to learn to play backgammon and thus achieve a super human level of play, entirely by using reinforcement learning. TD-gammon employed a reinforcement learning algorithm which was alike the Q-learning algorithm and model-free, and thus approximated the value function with the help of a multi-layer perceptron having only one hidden layer.

It was soon discovered by other researchers that TD-gammon was possibly a special case that might have only worked in the case of backgammon when they applied the same methodologies on other games like chess, Go etc with little success. A research [5] by two computer scientists from Brandeis University brought to the notice that the success of TD-gammon was mostly because the structure of the learning task was co-evolutionary in nature and even the dynamics of the game (backgammon) itself, i.e., the stochasticity in the dice rolls aided in exploring the state space and even made the value function converge smoothly.

Later on, another analysis was done on the temporal difference learning algorithm with function approximation in [6], in which they illustrated a possibility of divergence when the algorithm is used in the presence of a non-linear function approximator. This has been one of the long-standing difficulties facing reinforcement learning, as deep learning models are non-linear in nature, and thus there was no successful attempt at amalgamating deep learning with reinforcement learning to learn control policies

straightaway from the sensory inputs that are high-dimensional in nature.

Early work on deep learning dates was done by Geoffrey Hinton in his research paper [7], in which he stated that, to control behavior the perceptual system needs to be enabled to make fine distinctions, for which the sensory cortex needs to efficiently learn to detect features. Since there are multiple features to be extracted and transformed, deep learning uses a cascade of many layers of nonlinear processing units. The most efficient way to learn more than one layers of feature detectors is by adjusting weights by passing the data in top-down connections in neural networks, as the labeled data is scarce or non-existent.

As suggested by [7], the requirement for a lot of labeled data can be eliminated by learning a particular model that simply generates the sensory data instead of classifying the data. However, generative models with just one hidden layer are far less complex for sensory data that is highly structured, as compared to multilayer, non-linear models. In AI and statistics, hidden variables have discrete values and if some methodology can be figured out to deduce posterior distribution atop hidden variables for each of the data-vector, learning a generative model, that has more than one layer, is relatively uncomplicated. One way of doing so is by learning a single representation layer at a specific time with the help of restricted boltzman machines, which in turn distributes the entire task of learning into various simpler and smaller tasks and thus removes the deduction issues that appear in generative models that are directed. Another idea discussed in [7] is to segregate the fine-tuning stage so that the discriminative or generative qualities of the composite model can be improved. Thus, a good strategy is to initially assimilate a generative model which deduces the variables that are hidden from sensory data like vision and then, instead of learning a complex mapping from sensory data to the labels, learn a comparatively straightforward mapping from variables that are hidden to the labels.

Then, DeepMind presented a deep learning model, using convolutional neural network(CNN), that successfully learnt to control policies directly from a sensory input which is high-dimensional in nature, using the technique described above, termed as reinforcement learning. CNN is a type of feed-forward artificial neural network that has proven to be a very successful algorithm for supervised learning on image data, but has required a lot of hand labeled training data to be efficient.

CHAPTER 3

Automating playing archon

The invention of true artificial intelligence (AI) has become a task of paramount importance for humanity as machines can be trained to think independently in complex situations, as stated by [8]. To create an AI that can interact with the real world, video games form the apt testbed as they take into consideration all the complex situations a person could face. We chose to teach the computer to learn to play Archon: The Light and the Dark on its own.

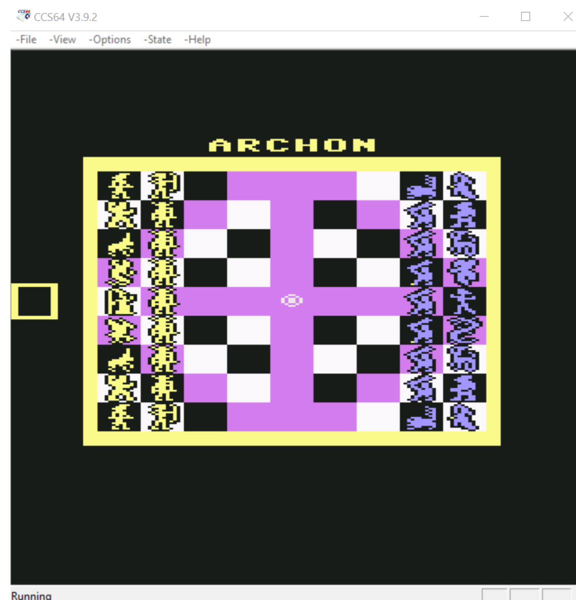


Figure 1: Strategy mode of Archon.

Archon has two game modes as shown in figure 1 and figure 2. The Strategy mode of Archon in figure 1, contains 36 players, 18 on the light side and the other 18 on the dark side. Out of these players, 16 players are unique, 8 on each side. We run Archon on a Commodore 64 emulator sourced from "CCS64.com". In order to be able to play Archon, we need to programmatically control it using keyboard and mouse events. Thus, we interact with the emulator with the help of a python script written using

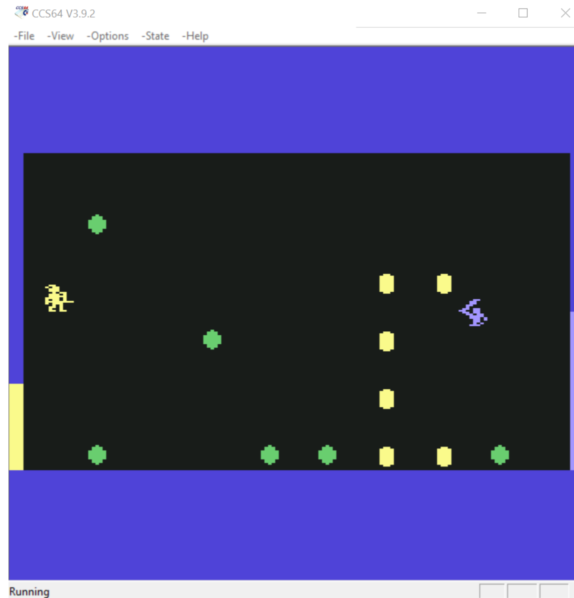


Figure 2: Fight mode of Archon.

the python libraries *pyautogui* and *keyboard*. The library *pyautogui*, a cross-platform module for GUI automation for human beings, controls the mouse functions of the machine and is also useful when a screenshot needs to be taken. The other library *keyboard*, was used to simulate keyboard events on the emulator. We chose to focus on the fight mode for this research. Further in this section, we describe the fight mode of Archon and how we automate playing it using the previously described libraries in python.

The game begins with moving the cursor from a location of the program on pycharm (Python IDE) to the center of the emulator, Commodore 64, which is done by computing coordinates of the screen and then placing the mouse position to where the emulator is on the screen. This is done with the help of two functions of *pyautogui* library, `size()` and `moveTo()`. The function `size()` returns height and width coordinates of the screen and `moveTo()` places the cursor to where the emulator is and after reaching the target position, the function `doubleClick()` points to the

emulator and then we start the game with the help of the library `keyboard`. The game loads when the string 'escape' is passed as an argument of `press()` function of `keyboard`, which in turn emulates the escape key of the keyboard.

Once the game loads, we reach a state where we are required to chose either the light or the dark side as shown in figure figure 3.

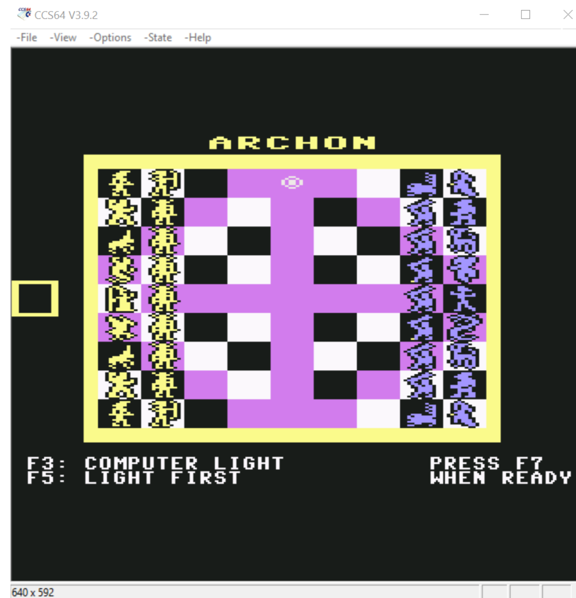


Figure 3: Select the side to fight from

But between the mode when the game loads and select mode, there is some time gap and thus the program needs to wait for a while before passing any other control instruction. Hence, another parameter called interval, is passed as the argument of the press() function. The program then waits for the time specified in the interval parameter. Once the program reaches the strategy mode, as shown in figure 1, it needs to select a player depending upon the side it chose in figure 3.

The light side contains 8 players:

```
1. Valkyrie
2. Golem
3. Unicorn
```

4. `Djinni`
5. `Wizard`
6. `Phoenix`
7. `Archer`
8. `Knight`

The dark side also contains 8 players:

1. `Banshee`
2. `Troll`
3. `Basilisk`
4. `Shapeshifter`
5. `Sorceress`
6. `Dragon`
7. `Manticore`
8. `Goblin`

CHAPTER 4

The reinforcement learning algorithm

A common human approach to solve Archon would be to either take control of five power points located on the board, to eliminate all the opposing pieces, or to eliminate all but one remaining imprisoned piece of the opponent's, which is reaching the goal state. A similar strategy shall be eventually followed by a machine or an agent, once it learns what the goal state is and how to reach it, but the learner or the agent is not told what actions to take to reach the goal state. It must figure out on its own, the actions that yield the maximum reward. This learning approach, RL, as shown in figure 4 is described in [9], where the agent learns to map events happening in a particular environment to actions to maximize a numerical signal, i.e., the reward obtained. Moreover, in the most cases like video games, which are challenging in nature, actions taken might not only affect the immediate rewards obtained, but also the future rewards and situations. Thus, delayed rewards and trial and error search are one of the most significant distinguishing features of RL, and the former is based on only the rewards that the agent must learn to behave in a particular environment.

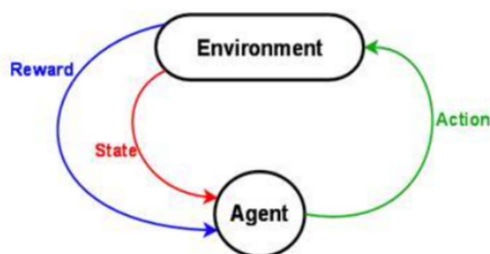


Figure 4: Reinforcement learning problem.

4.1 Challenges faced by RL

The idea of RL seems quite intuitive, but in practice numerous challenges are faced as described by the blog in [10], two of those challenges are discussed in the following subsections.

4.1.1 Credit assignment problem

The credit assignment problem is a phenomenon in which the agent performs an action and receives a positive reward, but that doesn't have any relation with actions performed just before getting that reward. However, it does depend on some of the preceding actions to a certain extent. For example, in the case of Archon, if a Knight hits a Goblin and lowers the latter's power, the fire action by the Knight isn't responsible for Light side getting a positive reward. The (prior) actions leading to this reward had already been taken when the Goblin was placed in the line of sight of the Knight.

4.1.2 Explore-exploit problem

Another challenge of RL is the explore-exploit predicament which brings forward a quandary that once an action plan to obtain certain rewards has been figured out, is it feasible to keep following it or experiment with something new that could result in a larger reward? In Archon, a player should hide behind an obstacle to defend itself from the opponent's attack. Would we be content with this or should we explore further? This phenomenon is termed as the explore-exploit quandary, where one needs to make a choice between the current working action plan and exploring other possible superior tactics.

4.2 Markov Decision Process

Considering the above mentioned challenges, a reasonable reinforcement learning strategy needs to be formalized, and a widely used methodology is to represent it in

the form of a Markov decision process(MDP) as cited in [11], as shown in figure 5. Let us say there is an agent or a bot, positioned in an environment, which in our case is Archon's fight mode. The state of the environment (fight mode of Archon) comprises of the location of the agent and it's opponent. The bot can execute any of the six different actions (i.e., move left, move right, move up, move down, do nothing or fire), the outcome of which might be a positive reward. Moreover, actions lead to the change of state of an environment, from the current state s to a new state s'. The rules that define how those actions can be chosen are called policy. The state of the agent after the execution of an action is somewhat random as the environment is traditionally stochastic.

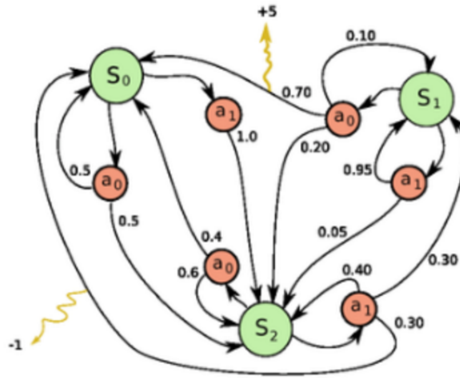


Figure 5: Markov decision process.

An MDP is a mathematical framework that consists of a group of states, actions, decision rules leading to other states and obtaining rewards. An episode of Archon, in the case of Markov Decision Processes forms a sequence of states, actions executed and rewards obtained that is finite in nature as shown below:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...s_{n-1}, a_{n-1}, r_n, s_n$$

In the equation above s_i depicts the state, a_i the action, r_i is the reward obtained

after the action is performed. The terminal state, s_n marks the end of an episode or the game itself. Markov chains are the source of a Markov decision process, where there is only one action for each state and all the rewards are similar. Whereas in an MDP, there are multiple actions and rewards, hence the probability of going to the next state s_{i+1} depends on the current state s_i and action a_i only, and not on the previous states and actions [12].

4.3 Discounted Future Reward

Since, in the long-term, apart from the immediate rewards received, even the future rewards need to be taken into account, [1] introduced a strategy known as discounted future rewards. To calculate the discounted future rewards, we need to compute the total future reward. We know that, the total reward for one episode of the game can be calculated using one cycle of the MDP, as follows:

$$R = r_1 + r_2 + r_3 + ... + r_n$$

Therefore, the summation of future reward from time t onwards, till time n, can be calculated as:

$$R_t = r_t + r_{t+1} + r_{t+2} + ... + r_n$$

However, the environment of our game is stochastic in nature. Hence there's always an uncertainty if the same rewards will be received next time for similar actions performed. It is possible that the more we dive into future, the further the future might diverge. Thus, we resort to using discounted future reward:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-t} r_n$$

In the equation above, γ is termed as the discount factor, whose value lies between 0 and 1, and the reason that it's power is increasing exponentially is because the immediate rewards are considered more influential than future rewards, hence we take

the former more into consideration. The above equation can also be represented with time step t+1, as shown below:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + ...)) = r_t + \gamma r_{t+1}$$

The discount factor γ should be set to somewhere in the middle of 0 and 1 if we aren't sure of our environment being dependent only on immediate rewards ($\gamma = 0$) or if it's deterministic $\gamma = 1$. The best strategy employed be an agent would be in choosing an action should result in maximizing the discounted future reward.

4.4 Q-learning Algorithm

Now that we are familiar with the preliminary concepts of Q-learning, let us discuss the algorithm itself in detail. Q- learning, which is a form of reinforcement learning, is a model-free algorithm in nature, as discussed in the technical note [13]. It is an uncomplicated technique for agents to learn to act in the best possible way in controlled Markovian domains and furthermore, it functions by sequentially making its evaluations of the quality of each action at each state better. The learning of the agent is similar to the one described by [9].

To maximize the total discounted expected reward, the agent needs to determine an optimal policy, which is done in a step by step manner.

$$Q(s_t, a_t) = max R_{t+1}$$

As shown in the equation above, the function Q is the maximum discounted future reward for executing action a at state s, representing the quality of a particular action in a given state. Thus, Q(s,a) is considered to be the best possible score when the game ends, and therefore it needs to converge at some point. For Q(s,a) to converge with an optimal value, an optimal rule is needed. The object in Q-learning, as stated by [13] is to compute the Q values for a policy π, that is optimal. Hence,

through Q function, the best possible reward can be computed by choosing the action that has the maximum Q-value:

$$\pi(s) = argmax_a Q(s, a)$$

Let us take the example of one transition in a game, i.e., $\langle s, a, r, s' \rangle$. The Q-value of state s and action a can also be expressed in the terms of immediate reward and a maximum reward that will be obtained in the future from state s' and action a', as shown below:

$$Q(s, a) = r + \gamma max_a Q(s', a')$$

This equation is also known as the Bellman Ford equation, i.e., the maximum future reward for the current state and the action executed can be computed by adding the immediate reward and the product of a learning parameter gamma and the maximum reward for the next state and action. Through Q-learning, the Q-function can be iteratively approximated using the Bellman equation. The Pseudocode of Q-learning algorithm, where the Q function is implemented in the form of a table is included in Algorithm 1.

State of the environment in Archon can be defined by the position of player on the light side and the position of its opponent, player on the dark side. The positions of players are calculated with the help of an image processing library in python, called *OpenCV*. An image is passed to the *matchTemplate* function of *OpenCV*, which in turn returns the x and y coordinates of player positions, based on the template images (shape of players) provided. The coordinates form the state "s" for the Q-learning algorithm. *OpenCV* is also used to compute lifelines of the players. Reward "r" for the Q-learning algorithm is calculated by computing the difference between lifelines of both the players using images before and after action "a".

Algorithm 1 Q function as a table

```
 1: procedure QLearn
 2:     Initialize replay memory D to capacity N.
 3:     Initialize Q with random weights for each action
 4:     for each episode of game do
 5:         seq ← [s_t, a_t, s_t, a_t, s_t]
 6:         while not goal do
 7:             Based on the value of epsilon
 8:             either select a random action
 9:             or select action with maximum weight
10:             Execute action in emulator
11:             Take a screenshot
12:             Calculate the reward
13:             Calculate the state
14:             Update seq ← [s_{t+1}, a_{t+2}, s_{t+2}, a_{t+3}, s_{t+3}]
15:             D ← [s_t, a_t, r_t, s_{t+1}]
16:             Sample random minibatch transitions from D
17:             Compute: Q(s_t, a_t) = r_t + γ max Q(s_{t+1}, a_{t+1})
18:             s_{t+1} ← s_t
19:         End while
20:     End for
```

4.5 Experience Replay

The above calculated transition, $\langle s, a, r, s' \rangle$ is stored in a replay memory D. This is the observation phase of the algorithm, where the transitions or experiences are stored in a batch. Once we have stored a batch of transitions of a defined length in D, the training begins, in which random mini-batches from D are used. Thus, the similarity of subsequent training samples, using most recent transition is broken. This phenomenon of storing experiences in a replay memory and thus using random samples from it for training is termed as experience replay.

CHAPTER 5

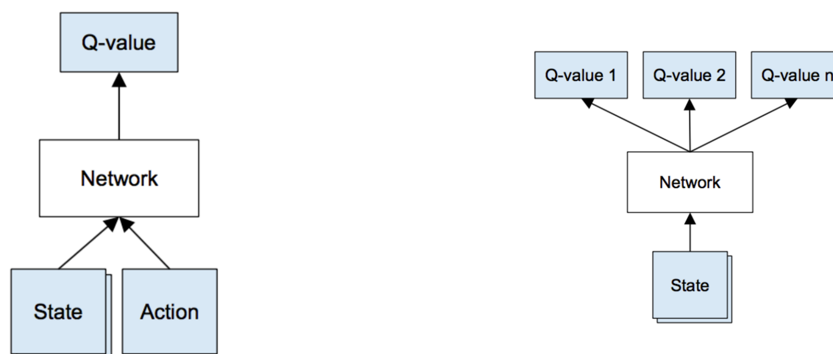
Q-learning with neural networks

A recent research by the team at DeepMind pioneered the idea [1] that neural networks can have a remarkable performance at game playing, using a minimal amount of prior information about the game. As we know from the discussion in the previous chapter that approximation of Q-values using non-linear functions is unstable, hence it might be troublesome to converge it properly using a Q table. Therefore, we resort to using neural networks as a policy for the Q function.

The environment's state in Archon in our previous approach was computed by the position of the players on Dark and Light side of the game on the screen. But in that case, the state becomes very specific to one game and thus [1] came up with a more universal strategy i.e., using raw pixel values. The screen pixels contain all the pertinent details needed for training, apart from the information about the speed with which both the players move. Therefore, [1] suggested using the last four consecutive screens of the game to cover the same, resizing them to 84x84 and then converting them to gray-scale with 256 gray levels. Thus we would have, $256^{84\times84\times4} \approx 10^{67970}$ possible game states.

As it's not very likely that all the states will be visited, in out previous approach we represented the table using dictionary as the data structure, with state as the id and a list of rewards for each action as the value. Each set of id and values were added to the dictionary as the states were being visited by the agent. But ideally, we would want that the Q-table should converge fast so that we get better results, as it would provide us with a good insight for Q-values for states that have never been visited previously.

According to [1] and the facts mentioned in [10], a neural network algorithm will help the non-linear function converge fast. We know that neural networks are



(a) Naive formulation of deep Q-network (b) Optimized architecture of deep Q-network

Figure 6: Two approaches of forming a deep Q-network

remarkably good at computing relevant features for data that is structured in nature. In this strategy, our Q-function is represented with a convolutional neural network, whose input is the game screen, representing the current state and output is a list of Q-values for each action as shown in figure 6b, which is a list of real numbers. Some people might argue that along with a screenshot of the current state, the actions should also be supplied as input and the corresponding Q-values should be evaluated. But in the latter approach, shown in figure 6a, to chose an action that has the highest Q-value, it would take as many passes, through the entire network, as the number of actions in the game.

In deep learning, a brain-inspired architecture is used, in which connections between layers of simulated neurons are strengthened on the basis of experience. When it comes to artificial neural network and deep learning in image processing, CNN is the most successful algorithm around. Moreover, as there hasn't been a successful attempt yet, to use a convolutional neural network to learn to play Archon, so we decided on building our own network to fill this gap. Then comes the application of reinforcement learning, which is a decision making system inspired by the neurotransmitter dopamine reward system in the animal brain. Using only the screen's pixels and game score as

input, the algorithm learns on its own by trial and error, which actions (whether to go left, right, up, down, fire or do nothing) to take at any given time to achieve the maximum reward [14].

5.1 Convolutional Neural Network

After the preprocessing, we design a convolutional neural network model. CNNs are a type of feed-forward artificial neural network, that are made up neurons having learnable weights and biases and the connectivity pattern between the neurons is inspired by the organization of the animal visual cortex. Each neuron receives certain inputs, performs a dot product operation between its weights and a small region it is connected to in the input volume. The entire network expresses a single differentiable score function: from raw image pixels on one end to class scores at the other [15].

We have implemented the above described algorithm for our data using a deep learning library called *Keras*. This neural network library is written in Python and can run on top of other neural network libraries, *TensorFlow* or *Theano*. We have used *Theano* as the backend library to run our model. *Keras* organizes layers in thr form of *models* and the simplest model in *Keras* is a *Sequential* model, that forms a linear stack of layers. We shall explain the model in detail in the next section.

The data is loaded using a predefined function in *Keras*, where it is structured as a 4-dimensional array of instance, image width and image height. But for a multilayer perceptron model, the image needs to be reduced down to a vector of pixels, hence the numpy array is reduced to an 80x80 two dimensional array. Further, the image array is scaled to have pixels between 0 to 255. As we discussed above, we have stacked four images together. Furthermore, the input is supplied to *Keras* as a four dimensional array of 1, image height, image width and channels (number of images, which in our case is 4). The number 1 is supplied because of the manner in which

Keras takes its input.

5.2 Network Architecture

The first hidden layer or the input layer of this model is a convolutional layer called a Convolution2D, having 16 filters, each of size 8 by 8, having stride 4 and a rectifier activation function. This layer expects images with having a structure, [1][width][height][channels]. The layer after that is another convolutional layer having 32 filters, with each filter being of size 4 by 4, and the stride being 2, also having a rectilinear activation function. The next hidden layer, which is a fully-connected layer comprises of 256 neurons rectilinear units. This fully connected layer or Dense layer outputs a list of 6 real numbers, each real number being a reward for an action taken.

This particular problem is not exactly a classification problem as there are no supervised inputs given apart from the inputs from the replay memory. The output of the model is an array of 6 different real numbers, as there are 6 different actions in the game. These numbers are rewards, mapped to each action. During the initial stage of the training, the output of our neural network model is a list of random numbers, when we call the *model.predict* function of *Keras* and supply our image as the input. The actual training phase of the neural network model beings when $model.train_o n_b atch$ function is called, which performs a single gradient update over an entire minibatch of samples as shown in step 21 of Algorithm 2.

5.3 The DQN Algorithm

The Pseudocode of deep Q-learning algorithm or DQN, where the Q function is implemented as a neural network is included in Algorithm 2

As described above, EPSILON is a number between 0 and 1, inclusive, that acts as a probability function for the agent. The agent is believed to take an optimal action based on the probability that if a random number whose value is greater than

Algorithm 2 Deep Q learning algorithm

```
 1: procedure QLearn
 2:     Initialize replay memory D to capacity N.
 3:     Initialize Q with random weights for each action
 4:     for each episode of game do
 5:         s_t ← image
 6:         seq ← [s_t, s_t, s_t, s_t]
 7:         while not goal do
 8:             Based on the value of epsilon
 9:             either select a random action
10:             or select action with maximum weight
11:             Execute action in emulator
12:             Take a screenshot
13:             Calculate the reward
14:             Calculate the state
15:             Update seq ← [s_t, s_t, s_t, s_t, s_{t+1}]
16:             D ← [s_t, a_t, r_t, s_{t+1}]
17:             Sample random minibatch transitions from D
18:             Predict Q values from the neural network model
19:             Compute: Q(s_t, a_t) = r_t + γ max Q(s_{t+1}, a_{t+1})
20:             s_{t+1} ← s_t
21:             Calculate loss to update weights
22:         End while
23:     End for
```

EPSILON is picked, an action would be derived from the output of *model.predict()* function. The output of this function will result into a list, and from that list an index with maximum value will be picked as each index corresponds to each action of the game. If the value of the random number is greater than EPSILON, then any random number will be picked. The function *model.predict()* will start giving relevant actions only after a few times the model has been trained and its weights have been updated according to each feature, based on the loss calculated at step 10 of the algorithm. Step 8 to 16 of Algorithm 2 is the observation phase of the game, where the agent is storing all the information into its brain. And step 17 to 21 is the training phase the game.

CHAPTER 6

Experiments

We performed our experiments on a Commodore 64 emulator for Windows, by loading Archon on it. The bot, programmed in Python, described in Chapter 3, was automated to play the game and then reach the fight mode. Once the fight mode begins, we start our observation. We use the same network architecture but different algorithms for training out bot. Furthermore, for training, we used a frame skipping technique described in [1] and [16]

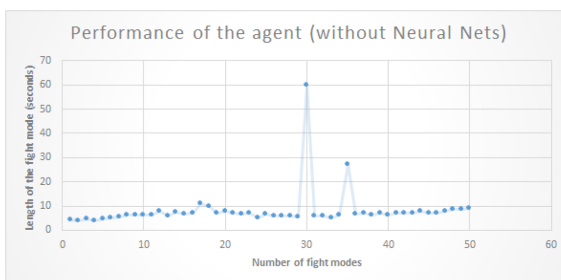


Figure 7: Result of Experiment 1

6.1 Experiment with the Q-function being implemented as a Q table

Our training experiments were performed on 50 iterations of the game, for two different policies. Before starting a new iteration of the game, we load the Q table from the previous iteration of the game, if any, from a pickle file. As we see in figure 7, the agent starts playing better gradually, but its improvement is very slow. It survives the fight for approximately 5 more seconds on the 50th iteration than when we had just begun training.

6.2 Experiment with Neural Networks

We load the previously stored weights of the neural network model from an h5 file before beginning an episode of the game or an epoch in the terms on neural networks. After performing the above mentioned experiment by training for 50 iterations of the

game, we observed that the agent learns comparatively faster with neural network as compared to when trained without neural networks. As shown in figure 7, the agent survives the game for about 8 more seconds on the 50th iteration of the game, in comparison to when it played without any training.

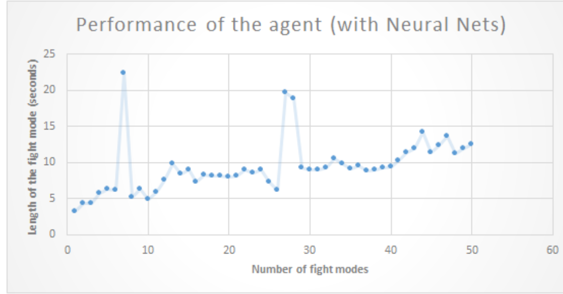


Figure 8: Result of Experiment 2

Moreover, we observe a few spikes in the charts above, the agent survived for so long because it's opponent couldn't figure out a way to reach it as there are obstacles in the game. But, this isn't helpful for training as we want to ultimately win the game and not just survive it.

CHAPTER 7

Conclusion and future work

7.1 Conclusion

This research showed how neural networks, when used with reinforcement learning perform a better training as compared to when the Q function is used as a table. Moreover, we also observed that the difficulty with reinforcement learning problem is the time lag between the action and reward.

7.2 Future Work

Various opportunities exist to improve the training of the Archon agent. First, the game needs to be automated, i.e., no human intervention should be there even to restart the game. This will in turn replace the manual work done to run the iterations of the game. Once the game is automated, we can increase the speed of the game in the emulator to train faster. Thus, the training iterations can be increased as less time will be taken to play the game and no manual work would be needed.

In addition, the bot needs to be trained to fight players other than the Knight (light side) and Golem (dark side). Also, the bot should be trained to play the game from the dark side as well. Lastly, technique similar to the fight mode of the game can be applied to the strategy mode of the game and instead of giving static inputs, the bot can move around randomly. Thus, after training for both the modes of the game, the entire game can be played through the reinforcement learning technique and results can be figured out.

Moreover, we believe that, with more training, the results can be improved and having a GPU would certainly help improve the training. Furthermore, by changing the network architecture, it's possible to converge the Q-function faster. Hence, we might be able to achieve performance significantly better than the current one.

LIST OF REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, ''Playing atari with deep reinforcement learning,'' *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ''Imagenet classification with deep convolutional neural networks,'' in *Advances in neural information processing systems*, 2012, pp. 1097--1105.

[3] L. Baird *et al.*, ''Residual algorithms: Reinforcement learning with function approximation,'' in *Proceedings of the twelfth international conference on machine learning*, 1995, pp. 30--37.

[4] G. Tesauro, ''Temporal difference learning and td-gammon,'' *Communications of the ACM*, vol. 38, no. 3, pp. 58--68, 1995.

[5] J. B. Pollack and A. D. Blair, ''Why did td-gammon work?'' in *Advances in Neural Information Processing Systems*. MORGAN KAUFMANN PUBLISHERS, 1997, pp. 10--16.

[6] J. N. Tsitsiklis, B. Van Roy, *et al.*, ''An analysis of temporal-difference learning with function approximation,'' *IEEE transactions on automatic control*, vol. 42, no. 5, pp. 674--690, 1997.

[7] G. E. Hinton, ''Learning multiple layers of representation,'' *Trends in cognitive sciences*, vol. 11, no. 10, pp. 428--434, 2007.

[8] [Online]. Available: http://togelius.blogspot.com/2016/01/why-video-games-are-essential-for.html/

[9] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press Cambridge, 1998, vol. 1, no. 1.

[10] [Online]. Available: https://www.nervanasys.com/demystifying-deep-reinforcement-learning/

[11] [Online]. Available: https://danieltakeshi.github.io/2015-08-02-markov-decision-processes-and-reinforcement-learning/

[12] M. Frampton and O. Lemon, ''Recent research advances in reinforcement learning in spoken dialogue systems,'' *The Knowledge Engineering Review*, vol. 24, no. 04, pp. 375--408, 2009.

[13] C. J. Watkins and P. Dayan, ‘‘Q-learning,’’ *Machine learning*, vol. 8, no. 3-4, pp. 279--292, 1992.

[14] S. Russell, P. Norvig, and A. Intelligence, ‘‘A modern approach,’’ *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, p. 27, 1995.

[15] V. Mnih, ‘‘Machine learning for aerial image labeling,’’ Ph.D. dissertation, University of Toronto, 2013.

[16] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, ‘‘The arcade learning environment: An evaluation platform for general agents.’’ *J. Artif. Intell. Res.(JAIR)*, vol. 47, pp. 253--279, 2013.

APPENDIX A

Characters of Archon: Light Side

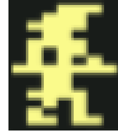


Figure A.9: Valkyrie

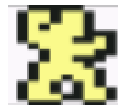


Figure A.10: Golem

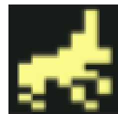


Figure A.11: Unicorn

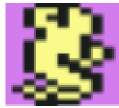


Figure A.12: Djinni

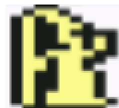


Figure A.13: Wizard

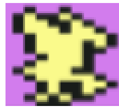


Figure A.14: Phoenix

Figure A.15: Archer

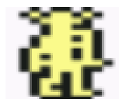


Figure A.16: Knight

APPENDIX B

Characters of Archon: Dark Side

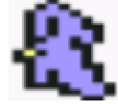


Figure B.17: Banshee

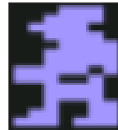


Figure B.18: troll

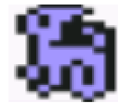


Figure B.19: Basilisk

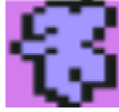


Figure B.20: Shapeshifter

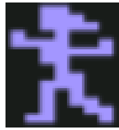


Figure B.21: Sorceress

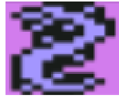


Figure B.22: Dragon

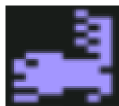


Figure B.23: Manticore

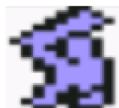


Figure B.24: Goblin