Applying Deep Reinforcement Learning to Berkeley's Capture the Flag game

Santiago Rojas Herrera

Supervisor: Prof. Silvia Takahashi

Department of Systems and Computing Engineering
Universidad de los Andes

This dissertation is submitted for the degree of
Major in Systems and Computing Engineering

January 2019

Table of contents

1 Introduction **1**
1.1 Objectives 2

2 Background **3**
2.1 Reinforcement Learning 3
2.1.1 Temporal-Difference (TD) 4
2.1.2 *Off-policy* learning and *On-policy* learning 5
2.1.3 Q-Learning 5
2.2 Convolutional Neural Networks 6
2.3 Deep Reinforcement Learning: Deep Q-Network 7
2.4 Berkeley's CS188 Capture the Flag game 8

3 Methods **11**
3.1 Used Technologies 12
3.2 Image Preprocessing 13
3.3 CNN Architecture 15
3.4 Reward Calculation 16

4 Results **19**
4.1 During Training 19
4.1.1 Agents using A as their reward function 21
4.1.2 Agents using B as their reward function 22
4.1.3 Agents using C as their reward function 24
4.1.4 Agents using *baselineTeam* as their $\varepsilon - greedy$ actions 25
4.1.5 Agents using *random* as their $\varepsilon - greedy$ actions 26
4.1.6 All Agents 28
4.2 After Training 28

5 Discussion **31**
5.1 Effectiveness of the Image Preprocessing 31
5.2 Effectiveness of the Algorithm Implementation 31
5.3 Effectiveness of the Reward Functions 32
5.4 Random or Agent recommended actions? 33

6 Conclusion **35**

References **37**

Chapter 1

Introduction

Artificial Intelligence (AI) has become one of the most promising fields in science and engineering, for that it could define the future of humanity. One of AI's most attractive qualities is allowing computers to learn from examples. One of the approaches used for this is known as Reinforcement Learning. Reinforcement Learning is inspired by the interaction between animals and their environment, particularly in how the environment is affected by what the agent does, and how the agent acts upon seeking its goal [17, p. 1]. In order to train an agent for a particular environment, it is required to provide the agent with a representation of the domain. By using Deep Neural Networks, it is possible to design an agent that perceives the domain in the shape of images, similarly to how some animals may use their eyes to perceive what's around him.

Games provide an interesting means of testing this theory, as they possess domains observable through images, one or many agents —mainly, the player— with a specific goal, and a way to iterate and test the training at hand. Groups like Google's DeepMind [2] or OpenAI [4] have already designed and implemented solutions to use Reinforcement Learning and Deep Learning in many games, which sets a guideline for those interested in learning and applying these concepts on their own. DeepMind's Deep Q Network (DQN) implementation is particularly interesting [14], as it set the guideline for following studies to improve on, or to compare against.

Berkeley's CS188 *Introduction to AI* [8] course designed a game called *Capture the Flag*, which is based on Pac-Man. The game is setup in a way that students can implement agents that can compete against each other. There are many interesting aspects about the game, some of them being: Agents play in teams of two; Agents are required to defend and attack well, in order to defeat their rival; Changes in score are considerable scarce, as it takes agents a significant amount of actions to eat and return food pellets. Also, it is interesting that an implementation of Deep Reinforcement Learning for the game has not been published yet. This sets the possibility to explore the inner workings of DeepMind's DQN algorithm, implement a specific solution, test it on different settings, analyze and compare the results, and set a path for future work.

The final implementation, with installation and testing instructions, is available at: `github.com/srojas19/dqn-contest`.

1.1 Objectives

1. Learn about Deep Reinforcement Learning (Personal).
2. Implement DeepMind's DQN on Berkeley's CS188 *Capture the Flag* game.
3. Train agents making use of the DQN implementation made.
4. Analyze and compare the results of the trained agents.
5. Describe possible improvements over the designed solution, and set a path for future work.

Chapter 2

Background

2.1 Reinforcement Learning

"Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics –trial-and-error search and delayed reward– are the two most important distinguishing features of reinforcement learning." [17, p. 2]

Formalization of Reinforcement Learning is reached mainly through Markov Decision Processes (MDPs), as it allows the definition of the interaction between the learning agent and the environment, in terms of states, actions, and rewards. Also, MDPs allow the modeling of stochastic situations, where an agent might execute an action with a defined discrete probability distribution for its set of actions.

Value functions are specially important for Reinforcement Learning, as it allows the agent to efficiently search through the space of policies [17, p. 13]. This value function is generally described with a Bellman equation, which describes the reward for taking the action giving the highest expected return:

$$V^{\pi}(S_t) = R(S_t, \pi(S_t)) + \gamma \sum_{S_{t+1}} P(S_{t+1}|S_t, \pi(S_t))V^{\pi}(S_{t+1}) \tag{2.1}$$

Note that in deterministic environments, where $(S_t, \pi(S_t))$ always leads to the same following state, the Bellman equation can be simplified to:

$$V^{\pi}(S_t) = R(S_t, \pi(S_t)) + \gamma V^{\pi}(S_{t+1}) \tag{2.2}$$

With:

- S_t: State of the environment for time t.
- $\pi(S_t)$: Action returned by the policy used by the agent, which determines the action that the agent should use for the given state.
- S_{t+1}: Following state, product of using $\pi(S_t)$ in S_t.
- γ: Discount factor, $\gamma \in [0,1]$.

2.1.1 Temporal-Difference (TD)

Temporal-Difference (TD) Learning is a tabular solution method for reinforcement learning problems. TD Learning is a combination of Monte Carlo ideas and Dynamic Programming ideas (both being other tabular solution methods). TD methods can learn without a model of the environment's dynamics. TD methods use bootstrapping, which means that they update estimates based on other learned estimates, without waiting for a final outcome. TD's main focus is the policy evaluation or *prediction* problem, which is estimating the value function V_π for a given policy π. TD methods use a variation of generalized policy iteration (GPI) to approach the prediction problem. [17, p. 119]

TD uses experience to solve the prediction problem. By following a policy π, TD updates its estimate V of v_π for the non-terminal states S_t occurring in that experience. TD methods only need to wait until the next step in an episode to make an update to $V(S_t)$, while other methods like *Monte Carlo* need a full episode to make an update. The simplest TD method makes the update:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.3}$$

With:

- S: State
- α: Step size or *Learning rate*, $\alpha \in [0,1]$
- R: Reward

- γ: Discount factor, $\gamma \in [0,1]$.

2.1.2 *Off-policy* learning and *On-policy* learning

All learning control methods face a dilemma: They seek to learn action values conditional on subsequent optimal behavior, but they need to behave non-optimally in order to explore all actions (to find the optimal actions). To take on this dilemma, there are two approaches: *On-policy* learning and *Off-policy* learning. The *on-policy* approach learns action values not for the optimal policy, but for a near-optimal policy that still explores. On the other hand, the *off-policy* approach uses two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. The policy being learned is called the target policy, and the policy used to generate behavior is called the behavior policy. In this case, it is said that learning is from data *off* the target policy, which is the reason that the overall process is named *off-policy* learning. [17, p. 103]

2.1.3 Q-Learning

Q-Learning is an off-policy TD control algorithm developed by Watkins [18], defined by:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \max_a Q(S_{t+1},a) - Q(S_t,A_t)] \tag{2.4}$$

Q, the learned action-value function, approximates to the optimal action-value function q_*, independent of the policy being followed. The policy, however, still determines which state-action pairs are visited and updated. This means that a policy that allows all state-actions pairs to be updated is required for the correct convergence of Q. The Q-Learning algorithm is as follows: [17, p. 131]

Algorithm 1 Q-Learning for estimating $\pi = \pi_*$

```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon < 0$
Initialize $Q(s,a)$, for all $s \in S^+, a2A(s)$, arbitrarily except that $Q(terminal,.) = 0$
Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R, S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
    until $S$ is terminal
```

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are neural networks that make the assumption that the inputs are images, which allows the encoding of special properties into their architecture. These allow to make the implementation of the forward function more efficient, while also reducing the amount of parameters in the network. Unlike a regular Neural Network, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth. The neurons in a convolutional layer are connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner.

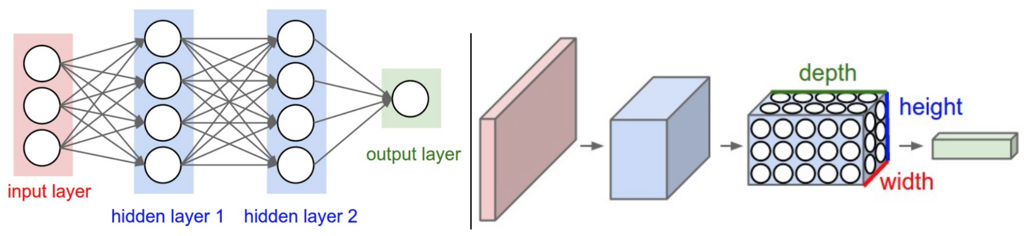


Fig. 2.1 On the left, a traditional neural network with two hidden layers. On the right, a CNN with two convolutional layers. Taken from Stanford's CS231n course's page. [1]

CNNs are built, mainly, with three different types of layers: Convolutional Layers, Pooling layers, and Fully-connected layers. Every layer in a CNN transforms one volume of activations to another through a differentiable function. In this way, CNNs transform the original image input to an output of scores (or values) that determine information about the image. The convolutional and fully-connected layers are trained with gradient descent so that the output computed is consistent with the training labels in the training set for a given image. [1]

2.3 Deep Reinforcement Learning: Deep Q-Network

Deep Reinforcement Learning are implementations of Reinforcement Learning methods that use Deep Neural Networks to calculate the optimal policy. Of these, one implementation that came to prominence is DeepMind's Deep Q-Network (DQN) [14], which uses a CNN to approximate Q, the action-value function. The use of a CNN means that the DQN agent uses a stack of images as inputs, which it then passes to the neural network. Then, the neural network outputs an array, for which each value is the result of $Q(s,a)$, with s being the current state, and a one of the actions that the agent can execute, according to an established order.

DQN has two key components that improve the performance of the algorithm: *Experience Replay* and *Iterative Updates*. Experience Replay consists in storing the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ (a tuple of a state, an action, a reward, and the following state) in a dataset. When applying Q-Learning updates, samples of the dataset are drawn randomly to train the network, which breaks the correlation between consecutive samples, therefore reducing the variance between updates [14, p. 7]. Iterative Updates means that the action-values Q are periodically updated towards the target values, which reduces correlations with the target. For this, *off-policy* learning is necessary, because the current parameters are different to those used to generate the sample.

The training algorithm closely resembles Algorithm 1 (Q-Learning). The difference resides mainly in the use of two CNNs to represent Q and $\hat{Q}$ (target Q). This means that the action-value updates are done with images ϕ instead of states s, although the states are used to generate the images. Target Q is updated every C steps, representing the use of *off-policy* learning. The algorithm is as follows: [14, p. 7]

Algorithm 2 Deep Q-Learning with Experience Replay

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**for** $episode = 1, M$ **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    **for** $t = 1, T$ **do**
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1. \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise.} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $Q$
        Every $C$ steps reset $\hat{Q} = Q$
    **end for**
**end for**
```

2.4 Berkeley's CS188 Capture the Flag game

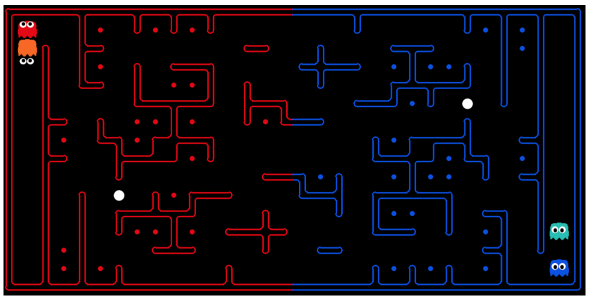


Fig. 2.2 Game of capture the flag on the default layout.

Capture the flag is a game implemented for Berkeley's *Introduction to AI* course. It is used for its final project, setup in a way that students implement a team of agents that can compete against other teams. "The course contest involves a multi-player capture-the-flag variant of

Pacman, where agents control both Pacman and ghosts in coordinated team-based strategies. A team will try to eat the food on the far side of the map, while defending the food on its home side."[8]

The game's layout is divided in two halves (red and blue), one for each team. When a team's agent is on its own side, it acts as a ghost which should attempt to defend its own team's food, while being able to eat an opponent that is attacking. If a ghost eats a Pacman, the food pellets captured by the Pacman will spread out to the closest available spaces. When a team's agent is on its rival's side, it acts as a Pacman, which should attempt to eat the opponent's food, avoid ghosts, and return the eaten food to its team's side. An attacking agent can eat a *power capsule* in its rival's side to *scare* the opponent's agents, which means that it can eat them, returning them to their initial position.

Score only changes when a team's agent returns the food pellets it ate from the opponent's side to it's own side. Each piece (white dot) eaten earns the team one point. Eating an opponent, eating power capsules, or eating food pellets without returning them won't result in a score change. Contestant's agents can access to state information such as: Food pellets' positions in each side (and thus, the quantity), power capsules' positions, walls' placement in the layout, the distance or position of the opponent's agents (depending on how far they are). **This project uses a modified version of the game that allows all agents to access the exact position of the other competing agents.**

Finally, the game ends when a team returns all but two of the opponent's food pellets, or if 1200 agent moves have occurred. Each move represents a game state, for which one specific agent must act. An agent's actions $A = \{NORTH, SOUTH, WEST, EAST, STOP\}$ are restricted by its surroundings (e.g. if there is a wall immediately west of the agent, it can't move $WEST$). The team that returned the most food pellets wins. If the final score is zero (both teams returned the same amount of food pellets), the game finishes as a tie.

Chapter 3

Methods

The full implementation is available on `github.com/srojas19/dqn-contest`, with installation and testing instructions. The implementation is based on an existing project that uses DQN on *Flappy Bird* (See Section 3.1), for which several changes/additions were made:

- All logic that interacts with the game was changed to be compatible with *capture the flag*. For this, a function that creates games and loads agents was implemented. Also, all other instructions that require the game's state information, access it with the API implemented for the game.

- The way actions were handled was changed to use objects of the class `Directions` of the game. Furthermore, the function `getLegalActionsVector(state, agent)` was implemented to restrict agents to use only possible actions (e.g. if there is a wall immediately west of the agent, The $WEST$ direction is blocked). The function returns an array of numbers, with a value in each position equal 0 or -1000, depending on whether the action in that position is possible or not. Then, the array is summed to the prediction of the model (i.e. if Q is used), restricting the use of illegal actions.

- Training data is captured to *CSV* files, which are later used to generate figures and statistics of the training.

- The algorithm was modified to use **iterative updates**, by using a *target model* that copies the weights as the trained model (See Section 2.3).

- Variables' names were changed to resemble DeepMind's DQN algorithm (Algorithm 2) with more accuracy.

- The model was changed to receive only one image as input, instead of a stack of four images. This is because, unlike Atari games (used by DeepMind) or *Flappy Bird* that are meant to be played by humans, *Capture the Flag* is meant to be played by computer agents. For this reason, the agents trained for *Capture the Flag* need to act for a given game state, which can be represented by one frame of the game (created with the game state), while DeepMind tries to simulate a human's reaction time, by stacking four frames of the game.

- The function `createMapRepresentation(state, agentIndex)` returns an image representation of `state`, as seen by the agent identified by `agentIndex`, which is then used as an input for the CNN model. Section 3.2 details the solution.
- The function `getSuccesor(game, state, agentIndex, action)` returns a tuple $(newState, reward, terminal)$, where *newState* is the product of the agent identified by `agentIndex` using `action`, and the remaining agents using their preferred action; *reward* is the value received by the agent for using `action` on `state` (defined on Section 3.4); and *terminal* is a boolean value that signifies if `game` has finished.
- While $\varepsilon - greedy$ is still used for exploration purposes, ε is reduced linearly during the training. For the majority of the experiments, ε starts at 1 and is reduced to 0.1 in the first million steps. Then, ε remains constant for the following five hundred thousand steps.

On top of this, the implementation was designed so that a trained model can be used by any of the four agents (red, orange, blue, and cyan). Originally, agents implemented by the students for CS188's contest (that uses *Capture the Flag*) must be enclosed in a team file that implements a set of functions used by the game to access the action chosen by the agents. `DQNTeam.py` creates a team of two `DQNAgents`. A `DQNAgent` loads a trained neural network and its correspondent weights, for the given path name (the one set on the training stage). When it has to choose an action, `DQNAgent` creates an image representation (equal to the one used on the training stage), which it then used as an input to the loaded neural network. Finally, `DQNAgent` chooses the action that has the maximum *Q-value*, according to the output of the neural network, and is not illegal.

3.1 Used Technologies

The implementation is based on Ben Lau's *Keras-FlappyBird* repository [12], which in turn, is based on Yen-Chen Lin's *DeepLearningFlappyBird* [13]. In short, Lau's implementation provided a simple python implementation of DeepMind's DQN algorithm (as seen in Algorithm 2), with the logic necessary to generate stacks of images specifically for the *Flappy*

Bird game. Lau's implementation differs from Lin's implementation in that they both use different deep learning platforms: Lau's uses *Keras* [3], while Lin's uses *Tensorflow* [6]. The decision to use *Keras* on top of *Theano* [7], over *Tensorflow*, was motivated by these reasons:

- *Keras* is a *high-level API*, which allows for a simpler use of the required functionalities for the project (that is, model creation, batch training, cloning and loading of weights into models).
- *Keras* is capable of running on top of different APIs, like *Tensorflow*, *CNTK*, or *Theano*. This is important because *Capture the Flag* is developed on Python 2.7, making it incompatible with *Tensorflow*. Using *Keras* allowed the use of *Theano* as its backend, which is compatible with Python 2.7.
- Diego Montoya's thesis [15], a project that aimed to implement a version of DQN (without the use of a CNN) on Berkeley's CS188 Pacman game, used *Keras* with *Theano* as its backend. Montoya's thesis served as a motivator to this thesis, thus promoting the use of *Keras* in this project's implementation.

It's important to note *PyTorch* [5], another deep learning platform, as an interesting alternative to *Keras* and *Tensorflow*, since it offers the granularity of *Tensorflow*, while also being compatible with Python 2.7 like *Keras* on top of *Theano*. Also, *PyTorch* has grown in popularity within the research community, making it an attractive option for follow-up work on this project. The main reason *PyTorch* wasn't used for this project is because I didn't have prior knowledge of it as an alternative.

Additionally, `numpy` was used for arrays and matrices, and `matplotlib` was used to test the results of the image generation algorithm.

3.2 Image Preprocessing

A challenge that arose from attempting to implement DQN over *Capture the Flag* was that there was no simple way to capture frames of the game to pass to the Neural Network. Some different approaches were already designed to circumvent this in Berkeley's CS188 Pacman game (which shares many implementation aspects to *capture the flag*), namely two:

- Ranjan et al.[16] used *raw pixel images* of the game, by using screen shots of the display captured with `ImageGrab`. This resulted in 540x540x3 (height x width x color channels) images, which then were downsampled to 224x224x3 images.

- Gnanasekaran et al. [10] created equivalent images of the game's frames, with each pixel representing objects in the Pacman grid. This resulted in an increase in training speed according to their results.

In the end, an image generation approach (similar to Gnanasekaran et al.'s) was used for the following reasons:

- Generating images from the game's state information allows for a simpler, easier to understand, implementation of the training algorithm, similar to DeepMind's approach to training for Atari games (by using a defined API).

- By using generated images over raw frames from the game, one can maximize the information per pixel. For example, while an agent would take a 20x20x3 image to be represented in a raw frame, it would require a single, one-channel, pixel (1x1x1) in a generated image.

- **Generating images is the only way to add sufficient information to allow a model that can be used by all agents.** If raw images from the game were used, the CNN wouldn't be able to differentiate which agent it is representing. For instance, if the algorithm trained the CNN with games for all agents (red, orange, blue, and cyan), the CNN wouldn't be able to recognize which agent is the one that is using it. Instead, by using a generated image from the state, it is possible to assign a unique color to the agent that is using the CNN, another color for the agent's partner, and another color for both of its rivals. This allows all agents to use the same trained neural network to play the game.

- As a consequence of maximizing information per pixel, generated images are of low dimensions (16x32x1 for the default layout and 18x34x1 for random layouts), and thus, faster to train on.

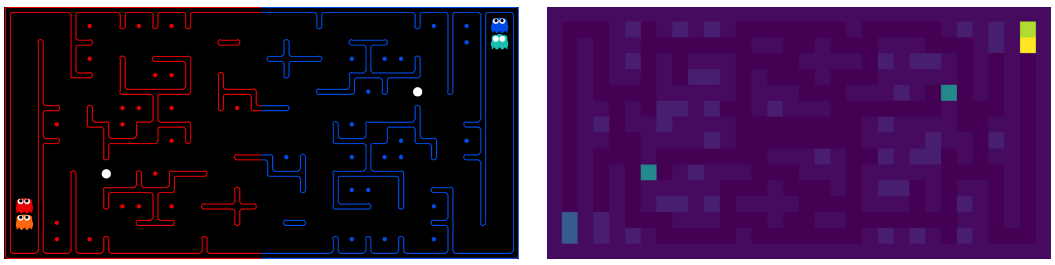


Fig. 3.1 Comparison of a raw frame from the game and a generated image from the game state. The generated image is shown as plotted by `matplotlib` (in reality, it is in grayscale). The second image is shown as it was generated for the cyan agent.

Table 3.1 Values for objects in generated image

Object	Value/Color
Walls	37
Food Pellet	46
Space	32
Power Capsule	112
Observing Agent (defending)	200
Observing Agent (not scared, attacking)	220
Observing Agent (scared, attacking)	230
Agent's partner (not scared)	150
Agent's partner (scared)	160
Rivals (not scared)	80
Rivals (scared)	90

All objects in the game are represented by one grayscale pixel. In other words, each object is represented by a number between 0 and 255, where 0 is absolute black and 255 is absolute white. The values for each type of object are defined in Table 3.1.

3.3 CNN Architecture

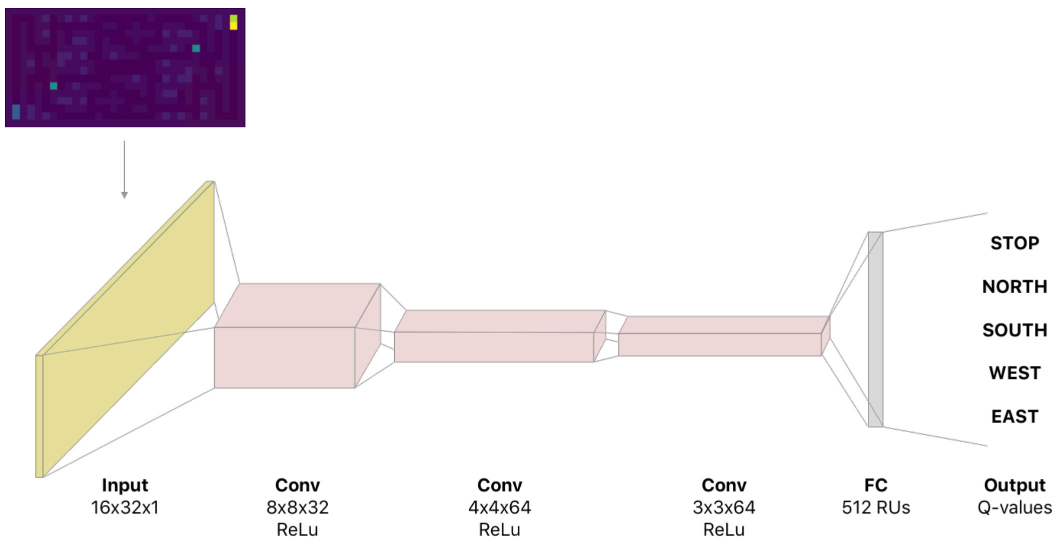


Fig. 3.2 Illustration of the CNN's architecture used for training and prediction.

This project uses a similar architecture to the one used by DeepMind [14]. All hidden layers are equal, and the dimensions of the input and output layers are changed. The input layer's dimensions are 1x16x32x1 (one grayscale 16x32 image) for models trained for the default layout, and 1x18x34x1 for models trained for random layouts. There is a separate output unit for each action that corresponds to the predicted Q-value for using the action in the input state. The experiments were carried with models using *Adam* as the optimizer, with learning rate *0.00025* (same learning rate as DeepMind), although the possibility to use SGD or RMSProp as optimizers exists. *It is important to recognize that more efficient architectures may exist, but finding one wasn't in the scope of this project.*

3.4 Reward Calculation

DeepMind's DQN implementation defines the reward used for action-value updates as the change in the score of the game. Their decision is based in making an implementation that can be used for multiple games, without major modifications. However, this negatively affects games were the score is scarcely changed, like *capture the flag*, because the transitions that make an impact on Q are lower than games than games that change the score with higher frequency. To illustrate, while the game *Pacman* would change the score every time Pacman eats food or a ghost, in *capture the flag* the score changes whenever an agent eats one or more food pellets, and then returns it to its own side. Arjona-Medina et al. [9] have shown that redistributing rewards (i.e. making rewards more frequent) in games with scarce delayed rewards can significantly improve agent performance and training speeds. With this in mind, different reward functions where defined to train agents, by taking into consideration these events of the game:

- sc: Score change after all agents move.
- n: Food pellets returned by the agent.
- s: 0.5 if the action attempted by the agent is $STOP$, 0 otherwise.
- fr: Food pellets recovered by the agent, by eating an opponent.
- fl: Food pellets carried by the agent (not yet returned) that was lost because the agent was eaten by an opponent.
- fe: +1 if the agent ate one food pellet (which hasn't been returned yet), 0 otherwise.

The following reward functions were used to train agents:

$$A: \quad r = sc - s + fe + fr \tag{3.1}$$

A attempts to reward the agent when it recovers food by eating an opponent (with fr), and when it eats one of the opponent's food pellets (with fe). s is meant to motivate the agent to move at all times. sc informs the agent if its partner or itself returned food to its side, by giving it positive feedback. Additionally, sc informs the agent if the opponent's agents returned food pellets to their side, by giving it negative feedback.

$$B: \quad r = sc - s + fe + fr + n - fl \tag{3.2}$$

B gives the agent the same information as A. Also, n attempts to allow the agent to receive additional feedback if it returned food pellets, by scoring for its team. fl tries to make the agent averse to its opponent's agents, by giving it negative feedback when it gets eaten by one. This aversion should be increased by the amount of food pellets carried by the agent.

$$C: \quad r = sc + n \tag{3.3}$$

C is a bare-bones implementation, similar to DeepMind's. The only addition is the use of n to allow the agent to recognize when it scores for its team.

The reasoning behind these reward functions is to test the effectiveness of advancing rewards in the training, regarding how well the resulting agent plays. Chapter 4 shows the results of the training.

Chapter 4

Results

Testing was carried by training a set of 6 agents. The difference between them consists in the use of different reward functions (defined in Section 3.4) and the actions applied when using $\varepsilon - greedy$ exploration. Some agents use the action that a `baselineTeam` agent would do, while others perform a random action. Since `baselineTeam` implements two different agents —one that attacks and one that defends— the solution is designed to choose one of the two agents uniformly. Incidentally, `baselineTeam`'s agents are quite poor, the attacking agent moves towards the closest food pellet, and the defending agent tries to chase down its rival when it sees them. This effectively makes the training agent learn from the choices of two different agents. The trained agents are:

Table 4.1 Trained agents

Agent	Reward Function	$\varepsilon - greedy$ action
A baseline	A	baselineTeam
A random	A	random
B baseline	B	baselineTeam
B random	B	random
C baseline	C	baselineTeam
C random	C	random

4.1 During Training

Training for each agent finalized when the training agents completed 1,500,000 actions. ε is reduced linearly from 1 to 0.1 during the first 1,000,000 actions, to then stay constant for the remaining 500,000 actions.

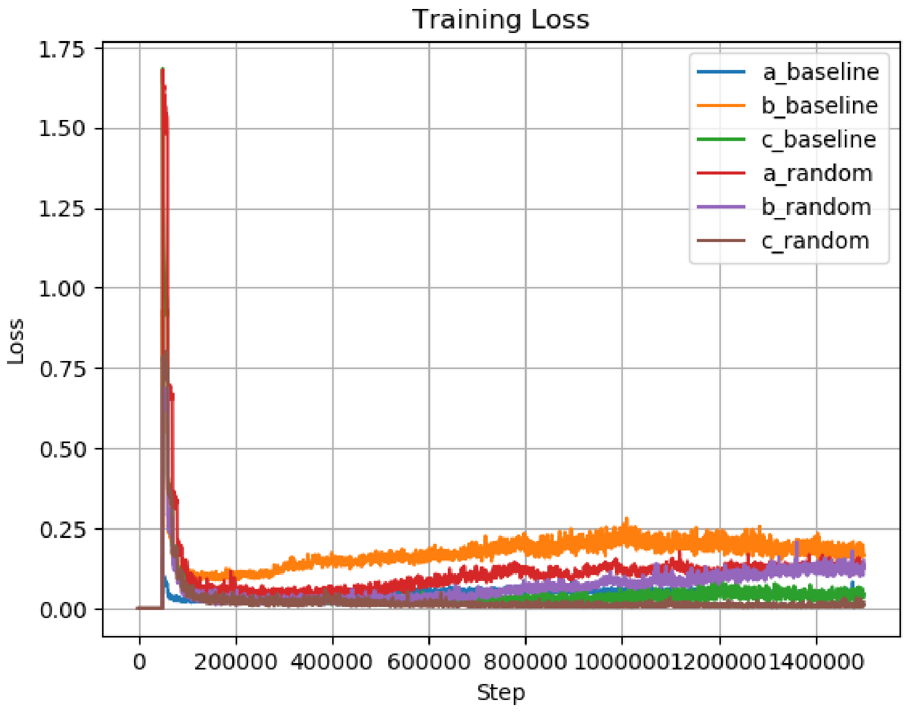


Fig. 4.1 Training loss for all agents.

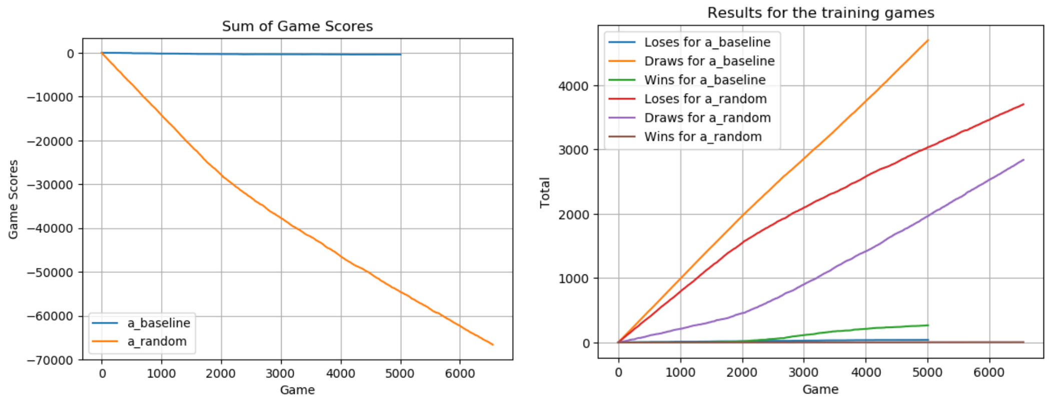
Figure 4.1 shows the evolution of the loss function during the training. The spike at step 50,000 for all agents is because that is when the neural networks start to train on batches from the replay memory. As a reminder, the training loss is equal to the one defined by DeepMind [14, p.1], which is a mean-squared error (MSE). MSE values are considered better when they approach zero. As such, *A baseline*, *C baseline* and *C random* seem to show the best results, because the loss value stays low (below 0.5) and consistent. On the other hand, *A random* and *B random* appear to increase in value, both peaking at $\sim$0.125. This signifies an increment in the deviation of the predictions from the neural network to the target values. *B baseline* shows a bell-shaped progression that peaks at $\sim$1,000,000. This might indicate that the training loss would have decreased if the training carried on further, although that is not guaranteed. Altogether, while this figure gives insight of the training, it is not enough to judge the performance of the agents.

To give more intuitive results, the following comparisons are between agents with equal reward functions, and between agents with equal $\varepsilon - greedy$ actions. The figures show for each comparison are:

- Sum of Game Scores: Plots the aggregated score of all training games prior to the current one. For example, the value for game 3 is the sum of games 1, 2, and 3.

- Game Results: Displays a line for each possible outcome for each agent, with each one showing the aggregated training results until the current one. i.e. The figure shows the number of games that the agent won, lost, and drew, during training.
- Average Q Value: Shows the progression of the Q values predicted by the network. This is heavily reliant on the reward function used to train the agents, as rewarding the agent for more events may result in higher Q values.
- Sum of Rewards: Plots the aggregated reward of all training steps prior to the current one. For instance, an agent that continuously makes wrong decisions while training will lower its sum. Similarly to the last figure, this is reliant on the reward function used to train the agents.

4.1.1 Agents using *A* as their reward function

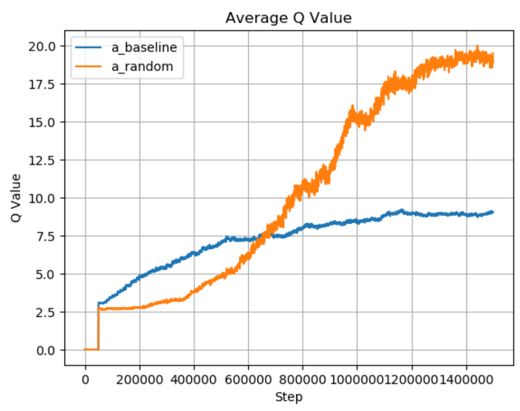


(a) Sum of Game Scores (b) Game Results

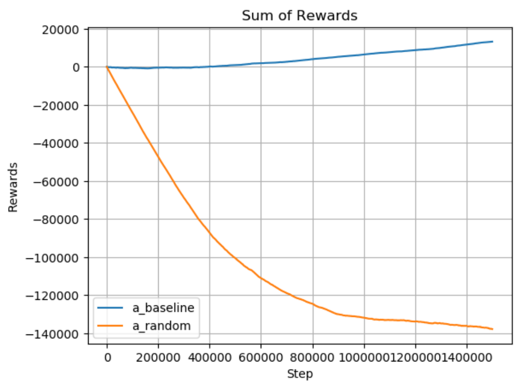
Fig. 4.2 Training results for A agents

As seen in Figure 4.2a, *A baseline* maintained an aggregated score close to 0 for the whole training. Moreover, *A random* always has a negative slope, that becomes less enunciated around the game 2000.

Figure 4.2b shows that *A baseline* drew the majority of its games, with a flux of wins beginning at the ~2500th game. In addition, more than 50% of the training games played by *A random* resulted in a loss, while close to 43% of the games resulted in a draw. Overall, this figure suggests that *A baseline* performed better than *A random* while training, since *A baseline* drew a higher percentage of games, lost close to 0, and won a considerable amount of games, while *A random* lost the majority of games, and seemed to win 0 games.



(a) Average Q Value



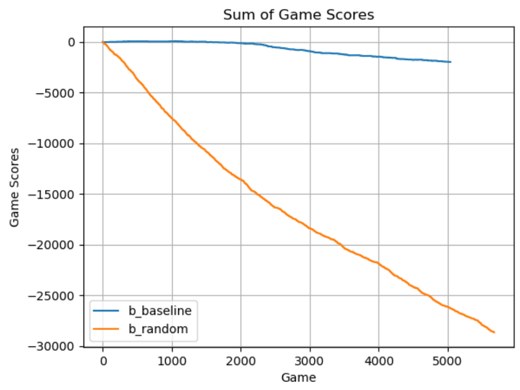
(b) Sum of Rewards

Fig. 4.3 Training results for A agents

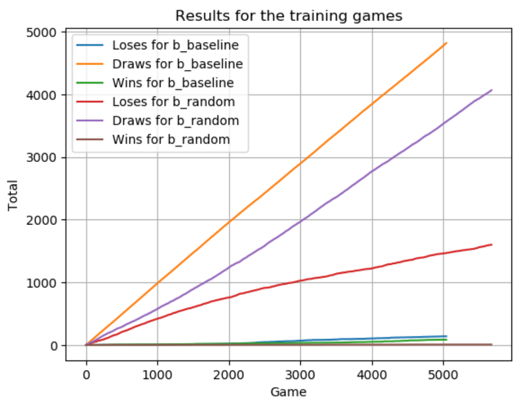
Figure 4.3a demonstrates that *A random* had a higher average Q value than *A baseline*. *A baseline* seems to converge around 9. While *A random* grew to 20, it might have converged later to a lower value.

Figure 4.3b presents the general performance of both agents during training. While *A baseline* sees a steady incline all the way, *A random* sees a sharp decline from step 0 to step 600,000, that later steadies to a lower slope, from step 1,000,000 to step 1,500,000.

4.1.2 Agents using *B* as their reward function



(a) Sum of Game Scores

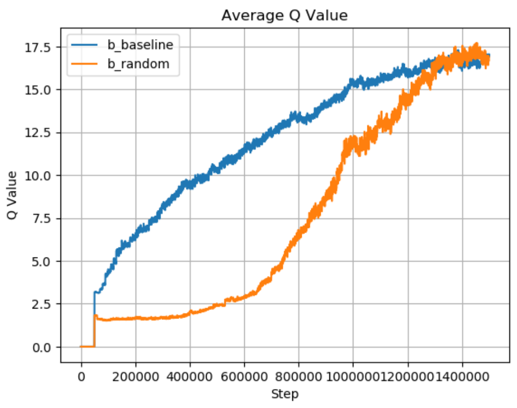


(b) Game Results

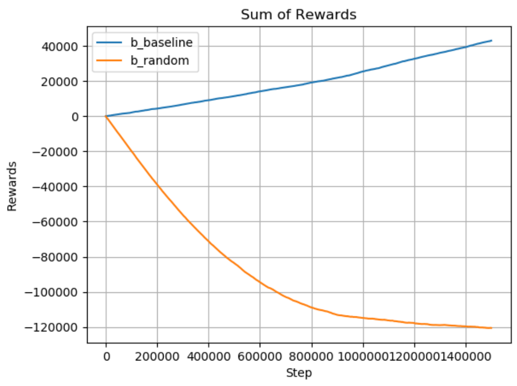
Fig. 4.4 Training results for B agents

A higher aggregated score for *B baseline* than *B random* is shown in Figure 4.4a. A transition from a near constant value to a steady decline appears from game ∼2200 for *B baseline*. At any rate, *B random*'s value ended close to 11.5 times lower than *B baseline*'s value.

Figure 4.4b shows, once again, better performance from *B baseline*. In this case, both *B baseline* and *B random* drew the majority of their games, although the proportion of lost games is larger for *B random* ($\sim$ 30% vs. $\sim$ 2%). *B baseline* won a small amount of games, while *B random* won close to 0 games.



(a) Average Q Value



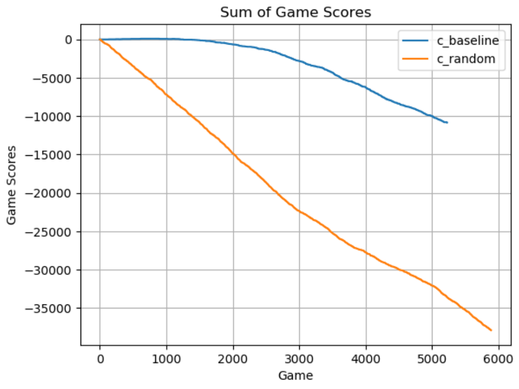
(b) Sum of Rewards

Fig. 4.5 Training results for B agents

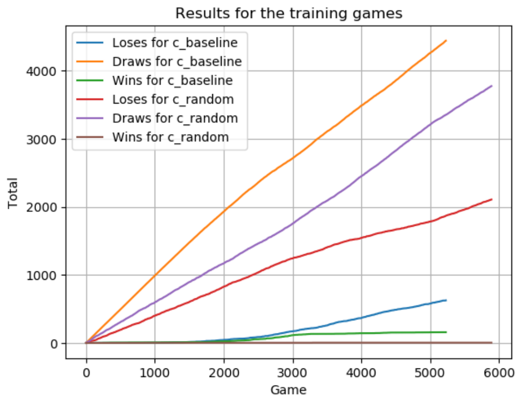
As shown in Figure 4.5a, both agents approach the same average Q values at the end of training. *B baseline* grew at a logarithmic rate throughout training, whereas *B random* grew exponentially for the first million steps, and at a logarithmic rate for the remaining steps.

A disparity between both agents' rewards sum is presented in Figure 4.5b. While *B baseline* always possessed a positive aggregate, *B random* always possessed a negative aggregate. *B baseline*'s training finalized with an aggregate of 40,000, and *B random*'s finalized with -120,000.

4.1.3 Agents using *C* as their reward function



(a) Sum of Game Scores

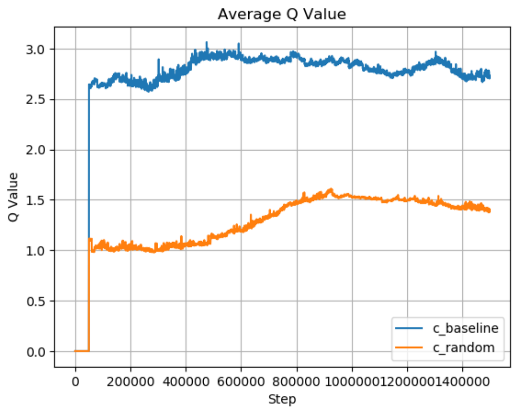


(b) Game Results

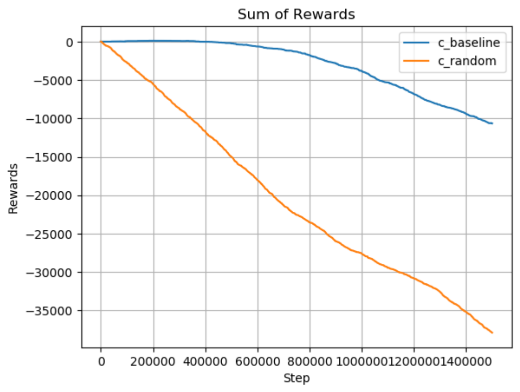
Fig. 4.6 Training results for C agents

Figure 4.6a continues the pattern set by Figures 4.2a and 4.4a, where the *baseline* agent had a higher aggregate score over its *random* counterpart. In this instance, *C baseline*'s aggregate finished at $\sim -11,000$, and *C random*'s aggregate finished at $\sim -38,000$.

C baseline performed better than *C random* on all stages, as seen in Figure 4.6b. For instance, *C baseline* lost $\sim$ 12% of its games, while *C random* lost $\sim$ 36%. Furthermore, *C baseline* drew $\sim$ 84% of its games, whereas *C random* drew $\sim$ 64% of its games. Lastly, *C baseline* won $\sim$ 4% of its games, while *C random* won $\sim$ 0% of its games. In sum, *C baseline* lost less, drew more, and won more games than *C random*.



(a) Average Q Value



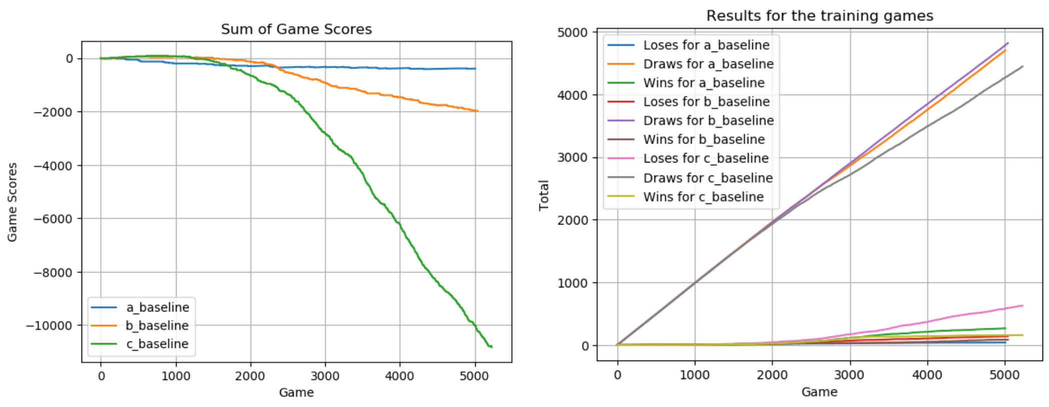
(b) Sum of Rewards

Fig. 4.7 Training results for C agents

Figure 4.7a shows that *C baseline*'s average Q value is always between ~ 2.5 and ~ 3, and *C random*'s average Q value is always between ~ 1 and ~ 1.5. Taking into consideration the reward function –*C* for both– the reason for these results are because *C baseline* returned more food pellets than *C random*.

Figure 4.7b shows essentially the same information as Figure 4.6a, but by representing the information in steps, instead of games. Consequently, this proves that the additional boost for returning food pellets in *C* wasn't used frequently enough during training to make a significant impact.

4.1.4 Agents using *baselineTeam* as their $\varepsilon - greedy$ actions

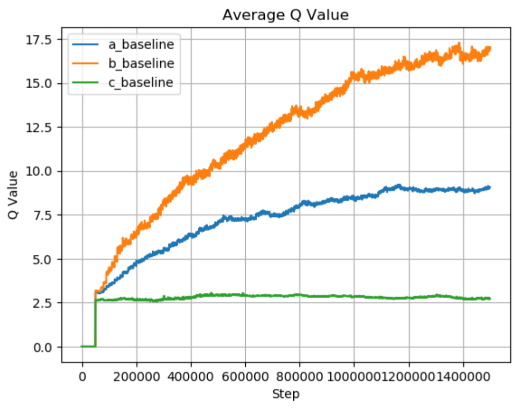


(a) Sum of Game Scores (b) Game Results

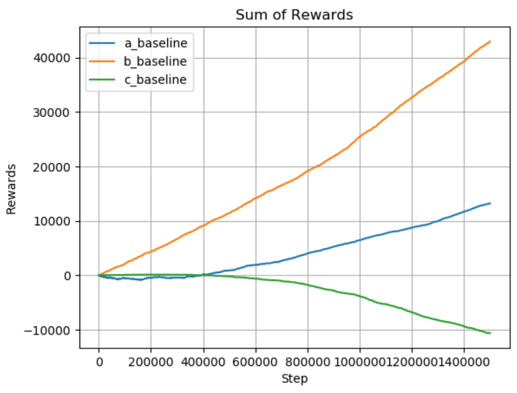
Fig. 4.8 Training results for baseline agents

Figure 4.8a presents the performance during training for all `baselineTeam` agents. All agents performed similarly during the first ~ 2300 games, diverging in the following games. While *A baseline* maintains a constant total score, *B baseline* and *C baseline* get continuously worse, with *C baseline* being the worst of the three.

As shown in 4.8b, the game results for all agents are similar, with all agents being clustered by the result of their games. That is, all agents drew the majority of their games, with the smaller proportion given for loses and wins. The amount of loses for *C baseline* seems to stand out over the bottom cluster, which means that *C baseline* was the agent that lost the most games out of the three agents.



(a) Average Q Value



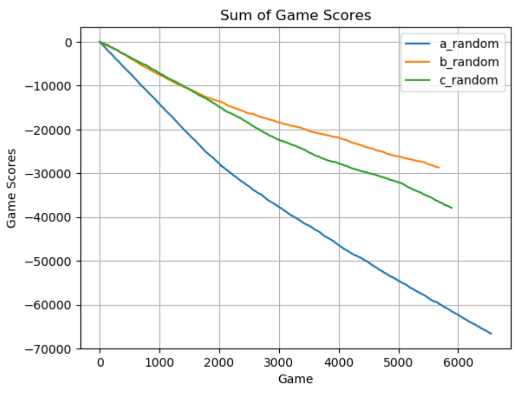
(b) Sum of Rewards

Fig. 4.9 Training results for baseline agents

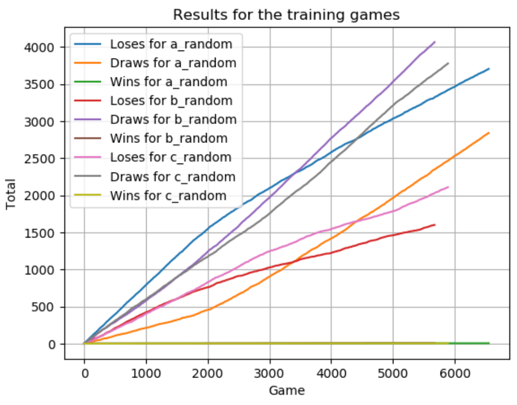
According to Figure 4.9a, *B baseline* reached the highest average Q value, followed by *A baseline*, and ending with *C baseline*. This correlates with the reward function used by each agent, with the ones receiving more feedback being the ones with higher Q values. Finally, it appears as though *A baseline* and *C baseline*'s Q values converged, while that may not be the case for *B baseline*.

Figure 4.9b is compatible with Figure 4.9a, as both position the agents in the same order. This is anticipated by the fact that higher Q values will result in a higher sum of the rewards given to the agent. However, it is interesting that *C baseline* is the only agent that achieved a negative sum, which could imply the worst resulting policy out of the three.

4.1.5 Agents using *random* as their $\varepsilon - greedy$ actions



(a) Sum of Game Scores

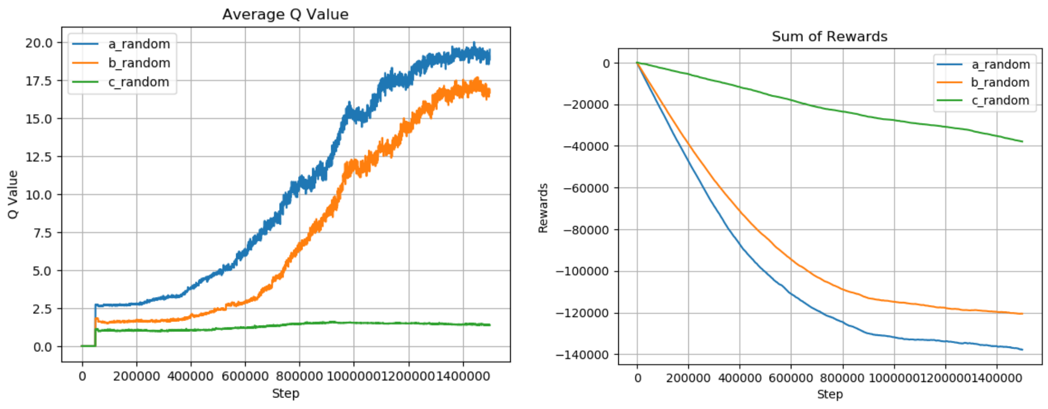


(b) Game Results

Fig. 4.10 Training results for random agents

Figure 4.10a shows that during training, all agents were always on a negative aggregate score. *A random* had the lowest value out of the three, with *B random* and *C random* being between 10,000 points of each other.

Figure 4.10b plots three different clusters of the results for each agent. The top cluster groups the amount of games drawn by *B random* and *C random*, and the games lost by *A random*. The middle cluster groups the amount of games drawn by *A random*, and the amount of games lost by *B random* and *C random*. The bottom cluster groups the amount of games won by all agents, all near 0. Given these points, it seems that *A random* performed the worst of all agents during training, and *B random* and *C random* performed similarly.



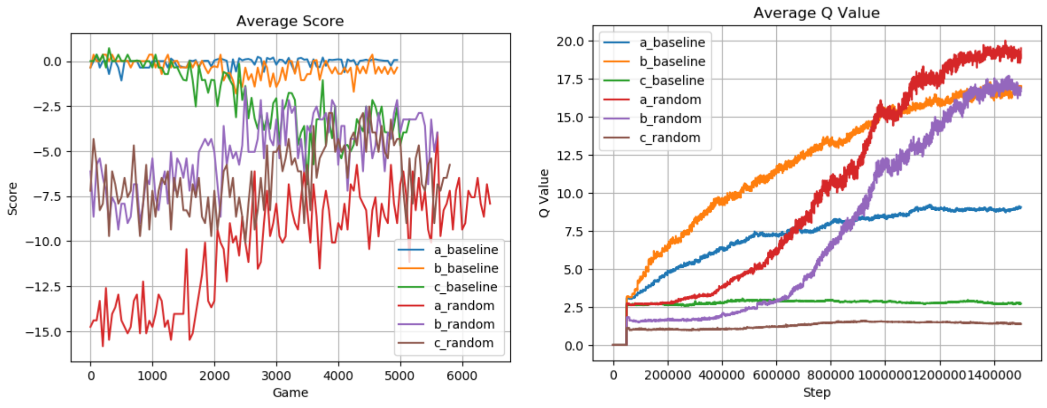
(a) Average Q Value (b) Sum of Rewards

Fig. 4.11 Training results for random agents

Figure 4.11a shows the average Q values for all *random* agents. *C random*'s values are considerably lower than *A random* and *B random*'s values. Interestingly, while *B random*'s reward function is designed to provide higher rewards than *A random*'s reward function, *A random*'s average Q value was always higher than *B random*'s.

Figure 4.11b presents all agents' aggregate rewards throughout training. All agents were always on negative values, with *C random* having the highest value of the three. These values are inversely proportional to the values in Figure 4.11a, meaning that the higher the average Q value, the lower the sum of the rewards.

4.1.6 All Agents



(a) Average Scores (b) Average Q Values

Fig. 4.12 Training results for all agents

The average score achieved for all agents during training are presented in Figure 4.12a. *A baseline* appears to be the best performer throughout training, with *B baseline* performing similarly. *C baseline* begins with almost identical scores to *A baseline* and *B baseline*, but it starts declining at the 2000th game. *B random* and *C random* begin at lower scores, fluctuating along the games without improving significantly. *A random* begins with the lowest average score out of all the agents, but manages to improve its value, closing on *B random* and *C random* at the ending. Overall, the agents that used *baselineTeam* actions achieved better scores, compared to their *random* actions counterparts.

The average predicted Q values for all trained agents are provided in Figure 4.12b. Both agents using *B* as their reward function reached the same average Q values. *A random* reached the highest average Q value at the end of training, also doubling the value reached by *A baseline*. Both agents using *C* as their reward function had the lowest average Q values out of all the agents, maintaining similar values for the duration of the training.

4.2 After Training

100 games were played between a team of two of the trained agents and `baselineTeam` to test the final performance of each trained agent. Of these games, 50 were played with the training agents as Red (left side), and 50 were played with them as Blue (right side). This allows the identification of the effectiveness of the agents depending on the side of the field they play. Furthermore, putting two agents that use identical neural networks gives insight into if the agent learned two play as a team, or if it works individually. `baselineTeam` was

used as the opponent for all agents, because it was the same team used to train the agents, and also because it reduced the amount of variables between the results.

Table 4.2 Results after 50 games, playing as RED

Agent	Avg. Score	Win %	Draw %	Loss %
A baseline	-14.06	0%	22%	78%
A random	-0.36	0%	98%	2%
B baseline	0	0%	100%	0%
B random	0	0%	100%	0%
C baseline	0	0%	100%	0%
C random	-18	0%	0%	100%

Table 4.2 shows the results after all agents played 50 games, playing as the red team. As seen in the data, *B baseline*, *B random*, and *C random* were the best performers, drawing all of their games. *A baseline* was the worst performer, with an average score of -14.06 and 78% of the games played lost.

Table 4.3 Results after 50 games, playing as BLUE

Agent	Avg. Score	Win %	Draw %	Loss %
A baseline	-7.92	0%	56%	44%
A random	-18	0%	0%	100%
B baseline	0.08	8%	92%	0%
B random	-17.64	0%	2%	98%
C baseline	-5.4	0%	70%	30%
C random	-18	0%	0%	100%

Table 4.3 presents the performance after all agents played 50 games as the blue team. *B baseline* was the best performer, winning 8% of the games and drawing the rest. In contrast, *A random* and *C random* were the worst performers, because they lost all of their games.

Table 4.4 Aggregate results of tables 4.2 and 4.3

Agent	Avg. Score	Win %	Draw %	Loss %
A baseline	-10.99	0%	39%	61%
A random	-9.18	0%	49%	51%
B baseline	0.04	4%	96%	0%
B random	-8.82	0%	51%	49%
C baseline	-2.2	0%	85%	15%
C random	-18	0%	0%	100%

The aggregate results of all the games played are shown in Table 4.4. The team that consisted of two *B baseline* agents was the team that won the most games (4%), and the only team with a positive average score (0.04). The team of *C baseline* agents had the second best average score (-2.2), while also being the second team with the least lost games (15%). At the end of the spectrum, the team of *C random* agents lost of all of its games, with an average score of -18.

Chapter 5

Discussion

Taking into account the results given in Chapter 4, one can assess the effectiveness, potential problems, and improvements of the key components of the solution. Each of the key components will be evaluated in the following sections.

5.1 Effectiveness of the Image Preprocessing

Given that all the trained agents showed meaningful results, it is possible to assume that the images used as inputs for the CNN were successful. To be specific, the fact that the majority of agents are capable of competing against manually designed agents, like the ones in `baselineTeam`, by entering generated images into a neural network, means that both the images and the network serve their purpose.

However, there exists the possibility of creating better images to use as inputs for the network. For instance, changing the values given for the objects in the generated image (Table 3.1 may give different results after training. For example, if the values were distributed in a way that the importance of each object was described by its value, the neural network could output better feature maps. Other option that could potentially improve the performance of the agents is using stacks of images as inputs. A stack of images could provide the CNN with information of the previous movements by the agents playing, which might lead to better decision making. Admittedly, it was unfeasible to implement and test these changes on time.

5.2 Effectiveness of the Algorithm Implementation

Generally, the implementation was capable of producing capable and coherent agents. DeepMind's algorithm (Algorithm 2) adapted successfully to *Capture the Flag*, given that the game provided an apt implementation to run games programmatically during the training of the agents. There were multiple stages during the implementation were bugs appeared, due to inadequate use of the game's methods, but all identified bugs were solved.

Nonetheless, it is possible that the algorithm used could have performed better with different values for the algorithm's hyperparamaters. As a reminder, the initial decision was to use the same values used by DeepMind [14, p.10], only changing the number of exploration frames to 1.5 millions (instead of 1 million) and the number of stacked frames to 1 (instead of 4). Specifically, increasing the number of exploration frames even further could result in better agents, but the training time could increase significantly. For reference, it took 17.46 hours to train each agent on average.

It is important to note that there could be an issue with the training algorithm, by the way the agents performed when playing as red or blue. According to the results in tables 4.2 and 4.3, all agents but one performed differently depending on the team they played as. Although a disparity between the results is expected, the difference in results for *A random* and *B random* are particularly concerning. While the team that consisted of *A random* agents drew 98% and lost 2% of the games as red, it lost 100% of the games as blue. Likewise, the team that consisted of *B random* agents drew 100% of the games as red, and drew 2% and lost 98% of the games as blue. This problem could be linked with the number of exploration frames used to train the networks, the architecture of the network, the nature of the game, or a misdiagnosed bug in the implementation.

5.3 Effectiveness of the Reward Functions

The results in Table 4.4 show that *B* was the most effective reward function. This is evident because: the team with *B baseline* agents was the best performer out of the teams using *baseline*-based agents, the team with *B random* agents was the best performer out of the teams using *random*-based agents, and *B baseline* was the only team that won games. This corresponds with the assertion that redistributing rewards improves the agent performance (Section 3.4, as *B* is the reward function that gives the most feedback to the agent, in turn allowing it to take better decisions.

With regards to *A* and *C*, the aggregated game results show that while *C* performed better with *baseline*-based agents, *A* performed better with *random*-based agents. This could indicate that *C* helps agents imitating actions to perform better, and *A* helps agents exploring randomly to perform better. If the results of the team using *A random* agents while playing as blue are considered as an anomaly (Table 4.3), *A* could be considered a reward function that is nearly as effective as *B*. Given these points, it is possible to consider advancing rewards as a beneficial addition to the training of agents.

As a final point, the fact that the win rate for all teams (Table 4.4) and agents (Figures 4.8b and 4.10b) is low could be justified by the way the agents were rewarded during training. Considering the setup of the game, and the way all functions reward good defending, it is easy to see that all agents find defending easier than attacking. The rewards –and particularly, the score change– are not sufficient to inform the agents if they are winning, drawing or losing. Thus, the agent will think that defending well is as good as attacking well. One analogy to understand this occurs in football: In the eyes of a team, stopping the opponent from scoring a goal is as valuable as attacking and scoring a goal; but, even if the teams defends excellently, it won't be able to win without scoring more than its opponent. Couple this with the fact that it is much easier for an agent to defend than to attack, given the layout of the map, and there is a recipe for good defending agents, but poor attackers. It is important to note that DeepMind's solution found games were the solution played below human-level [14, p.3], and *Capture the Flag* could be comprehended in this group.

5.4 Random or Agent recommended actions?

All results seem to indicate that training agents that use the actions of a manually designed agent when using the $\varepsilon - greedy$ strategy produces better agents. Taking Figure 4.12a as a reference, all *baseline*-based agents had higher average scores than their *random*-based analogs. Table 4.4 shows that a team composed of two *B baseline* agents performed better than a team of two *B random* agents, with *B baseline* being the only team that won against its opponent. The same trend continues with *C baseline* and *C random*, as the team with *C baseline* agents had an average score of -2.2, while the team with *C random* agents had an average score of -18 (it lost all games). Conversely, the team with *A baseline* agents performed worse than the team with *A random* agents, with a difference in average score of 1.81 between both.

One reason that could explain why the agents that used random actions as their $\varepsilon - greedy$ strategy performed worse than the agents that imitated the actions of the agents in `baselineTeam` is the number of frames used to train the agents. This suspicion comes from the instability of the average scores of *random*-based agents in Figure 4.12a, and the apparent lack of convergence for *A random* and *B random*'s average Q values in Figure 4.11a. It is likely that increasing the number of frames used for training, and particularly, increasing the number of frames were ε is reduced (1,000,000 in the tests), could result in more stable average scores and converged average Q values for these agents. Incidentally, this could also improve the results for the agents that imitate the actions of other agents.

Indeed, there is additional potential from training agents using an $\varepsilon - greedy$ strategy that imitates manually implemented agents. To put this into perspective, Table 4.4 shows that a team consisting of two *B baseline* agents –that is, agents that used *B* as their reward function and imitated the actions of the agents from `baselineTeam`– managed to win 4% and draw the remaining of its games. In contrast, playing two `baselineTeams` against each other for 100 games results in draws for 100% of their games. Due to this, it is possible to conclude that the *B baseline* agent performs better than the agents in `baselineTeam`. Allowing the possibility of training agents that imitate the actions of manual implementations of better agents should be considered. By doing this, and allowing the trained agent to learn from a different assortment of agents, it might learn a policy that is capable of competing against every possible agent. This would require additional work on the solution implemented, since it can only learn from one pair of agents (one team) as it stands.

Chapter 6

Conclusion

In conclusion, it was possible to apply Deep Reinforcement Learning to Berkeley's *Capture the Flag* game. DeepMind's DQN algorithm served as a suitable solution to train agents capable of playing the game. An algorithm to generate an image from the game state was implemented, which allowed for a simple, low resolution, representation of the game, while also providing additional information over the standard image displayed in the game, such as a defined color for the playing agent, for its partner, and for its rivals. This allows the trained neural network to be usable by agents in both teams, or with a different index (either the orange or red player in the red team, or the cyan or blue in the blue team). A way to advance rewards was applied, making the training agents recognize if they recovered *food* pellets from their opponents (by eating them), or if they ate food pellets from their opponent's side, even if these haven't been returned yet. Furthermore, the solution was found to be suitable to imitate the behaviour of conventionally implemented agents, by using their recommended actions instead of random actions when using ε-greedy to select the action.

In the future, it would be interesting to test the solution with different training parameters. That is, changing the number of training episodes, the learning rate in the neural network, the reward values (for score, or advanced rewards), the implementation of the image generation, etc. Also, the solution could be improved to allow the training of agents that imitate more than two agents' actions. The results of this project could be compared with an alternative that uses the displayed images from the game. With this project in mind, improvements in Deep Reinforcement Learning could be studied and applied, particularly DeepMind's Rainbow [11] and LIT AI Lab's RUDDER [9].

References

[1] CS231n Convolutional Neural Networks for Visual Recognition, . URL http://cs231n.github.io/convolutional-networks/.

[2] DeepMind, . URL https://deepmind.com/.

[3] Keras Documentation, . URL https://keras.io/.

[4] OpenAI, . URL https://openai.com/.

[5] PyTorch, . URL https://www.pytorch.org.

[6] TensorFlow, . URL https://www.tensorflow.org/.

[7] Theano 1.0.0 documentation, . URL http://www.deeplearning.net/software/theano/.

[8] UC Berkeley CS188 Intro to AI - Contest: Pacman Capture the Flag, . URL http://ai.berkeley.edu/contest.html.

[9] Jose A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, and Sepp Hochreiter. RUDDER: Return Decomposition for Delayed Rewards. *arXiv:1806.07857 [cs, math, stat]*, June 2018. URL http://arxiv.org/abs/1806.07857. arXiv: 1806.07857.

[10] Abeynaya Gnanasekaran, Jordi Feliu Faba, and Jing An. Reinforcement Learning in Pacman. *Stanford University*, pages 1–6, 2017.

[11] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *arXiv:1710.02298 [cs]*, October 2017. URL http://arxiv.org/abs/1710.02298. arXiv: 1710.02298.

[12] Ben Lau. Using Keras and Deep Q-Network to Play FlappyBird., November 2018. URL https://github.com/yanpanlau/Keras-FlappyBird. original-date: 2016-07-12T11:15:35Z.

[13] Yen-Chen Lin. Flappy Bird hack using Deep Reinforcement Learning (Deep Q-learning).: yenchenlin/DeepLearningFlappyBird, November 2018. URL https://github.com/yenchenlin/DeepLearningFlappyBird. original-date: 2016-03-15T03:52:16Z.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 1476-4687. doi: 10.1038/nature14236. URL https://www.nature.com/articles/nature14236.

[15] Diego Montoya. *Exploring how different state representations and configurations affect the learning process and outcome of deep Q-learning algorithms*. PhD thesis, Universidad de los Andes, Bogotá, Colombia, December 2016.

[16] Kushal Ranjan, Amelia Christensen, and Bernardo Ramos. Recurrent Deep Q-Learning for PAC-MAN. *Stanford University*, pages 1–8, 2016.

[17] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, MA, second edition edition, 2018. ISBN 978-0-262-03924-6.

[18] Christopher Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, May 1989. URL http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.