Exploring how different *state* representations and configurations affect the learning process and outcome of deep Q-learning algorithms

Senior thesis

Diego Montoya Sefair

Adviser: Silvia Takahashi Rodriguez

Department of Systems and Computing Engineering

Universidad de los Andes

Bogotá, Colombia

December, 2016

Abstract

Q-learning has advanced greatly in the last years. DeepMind, for example, has done highly extensive investigation using deep convolutional neural networks for Q-learning using the very pixels of the game. However, in reality we have more information that can be valuable for training like information from sensors and other information we can calculate and summarize to more easily understand the environment. This project aimed to understand how changing different parameters can affect training. More specifically it aimed to understand how and if changing the definition of the reinforcement learning state, using a persisted (pre-calculated) experience replay file, or varying the neural network's topology affect the result of the learning process overall in a considerable way. It was concluded that all three aspects affect the learning process considerably. Firstly, it was determined that the state representation should be as simple as possible (but complete) since simpler state representations are considerably easier to train on. Secondly, it was seen that having a pre-calculated, large, persisted replay memory has the potential to improve learning speed notably, but precautions should be taken so having one does not bias what the agent learns. Lastly, it was also seen that changing the topology of the neural network does have an important impact in the learning result. The hypothesis is that larger networks take considerably longer to train and were not able to train on time, so an appropriate topology should be selected (one that is complex enough to be able to approximate the Q-function for the particular problem but not more complex).

Table of contents

Introduction

Computers are very good at solving complex calculations and very strictly defined algorithms, but they come short at performing tasks that humans find particularly easy like playing games, understanding language, and recognizing images. Programs that perform well at such tasks had not been viable until not long ago since they required extremely large, highly structured, complex and poorly scalable algorithms that needed to take into account every possible scenario and variation. Consequently, wanting to solve these kinds of problems has driven a new trend in machine learning. New techniques have enabled the development of algorithms that are able to learn more like the human brain does, extracting features out of unstructured data, which gives them the ability to react to an environment and make decisions based on their experiences.

The applications of learning algorithms have been enormously valuable. Recently, Google DeepMind developed an artificial intelligence (AI) agent capable of playing GO better than the european human Go champion, winning five out of five games [1]. This has profound implications in technology, since it proves that it is possible for a computer to excel at tasks that were previously thought unapproachable to computers due to their immense search space. It moves us one step closer to the development of algorithms which are able to reason and learn more like we do. Indeed, this is important as the problems of everyday life have even more advanced levels of complexity. This type of advancements in machine learning would mean that not long from now we could be assigning computers some of the most challenging problems humanity has had in fields such as medicine, physics, and mathematics. We could be letting them learn to extract conclusions, instead of us coding them explicit algorithms to do so. But AI

is also being already used ubiquitously in our everyday lives. It has allowed for the development of tools that range from deciding the content that Facebook serves its users on their News Feeds [2] to high fidelity image recognition [3] and autonomous cars [4]. Therefore, we see that while research on AI is more active than ever, it is also already used to great extent and it is of great import to society.

Now, one of the most interesting testbeds for AI and reinforcement learning in specific are games, since they provide dynamic conditions which can mimic real life circumstances in which decisions have to be made and a reward or punishment is received. Q-learning has emerged as one of the most widely used and researched reinforcement learning techniques lately since it has proven to be one of the most powerful in solving problems that approach real-world complexity by extracting features and determining policies from high dimensional input data. Combined with deep learning it has been shown by organizations like Google DeepMind in their famous "Human-level control through deep reinforcement learning" paper [5] that it can provide very promising results for very large problems.

However, DeepMind's paper also shows that their AI agent had particular difficulties at learning some specific games like Pacman. While DeepMind's deep Q network (DQN) turned out to be far better than professional players at games like Pinball or Breakout, for PacMan for example, it was not the case. They do not state the reason specifically, but part of it seems to have to do with the relatively long-term planning that the game requires, combined with the fact that Q-learning as it is commonly implemented is known to have these kinds of temporal limitations.

The kind of implementation that DeepMind proposes is one in which the input (the *state*) consists of the pixels of a small number of consecutive screens. This gives the AI agent the same information a human player would get, which also has the benefit of being able to generalize to other games and problems as

well. Yet, some questions arise, namely: how would agents perform when given *states* composed of information other than the pixels? Does the format of the information we provide to the algorithm (e.g. having a state with the positions of the ghosts and PacMan encoded as coordinates as opposed to having the *x* and *y* distances to each ghost) have a notable effect on the way PacMan is able, for example, to learn to escape from the ghosts?

Therefore, and taking these questions into account, this project pretended to delve more into the area of reinforcement learning combined with deep learning. More specifically, this project aimed to understand how and if changing the definition of the reinforcement learning state affects the result of the learning process overall in a considerable way. It was also studied whether learning results were affected notably by using a persisted (pre-calculated) experience replay file or by varying the neural network's topology. The classic game of PacMan was used to look for an answer to these questions. This was done through implementing an AI agent that learns to play Pac-Man without any prior knowledge of the rules or game dynamics, and comparing the different approaches.

Chapter 1

Objectives

General

Determine how different representations of a given reinforcement learning *state* affect the learning process and the final outcome of a deep Q-learning algorithm.

Specific

- Build an AI agent which by using Q-learning in conjunction with neural networks will be able to learn to play PacMan without any prior knowledge of the the rules of the game.
- Determine if having a persisted, pre-calculated replay memory affects the learning process significantly.
- Determine how adding more complexity to the topology of the neural network affects the way the Q-function converges.

Chapter 2

Theoretical background

2.1 Reinforcement Learning

Reinforcement learning is an area of machine learning which is focused on solving the problem of learning what to do in a given situation so an overall reward is maximized in the long term. This is probably one of the areas of most interest within artificial intelligence and this is because many of everyday problems can be modelled as a reinforcement learning problem. In fact, most of the time this is how *we* learn. We are faced with a decision, we make one, and we observe if that was good or not.

Reinforcement learning is not entirely supervised learning, nor unsupervised learning, but rather it goes somewhere in between. In supervised learning there are target labels for the training samples, and in unsupervised learning there are no labels at all. However, in reinforcement learning there are sparse and time-delayed labels (the rewards) from which the agent has to learn in order to make the best decisions. [6].

One of the most important aspects of reinforcement learning is that it takes into account that taking an action does not necessarily only affect the immediate reward, but it may have an impact in the future rewards as well. This way, taking what seems to be the best action now may not be the best decision in the long term. This concept is known as *delayed reward*.

Another of the key aspects of reinforcement learning is that in order to maximize the reward we must follow what we have learned is best, but in order to learn that we have to also explore randomly sometimes. This is called the *exploration-exploitation* dilemma and will be discussed in further detail later.

2.2 Markov Decision Process

The most common way of formalizing a reinforcement learning problem is to represent it as a Markov decision process (MDP). A MDP provides a way of reducing a decision problem to core aspects.

A MDP problem is defined as: a set of states $s \in S$, a set of actions $a \in A$, a reward function $R(s, a, s')$, a start state s_0, and some final states if required. If the environment is stochastic there may also be an equation $T(s, a, s')$ which defines the probability of being in *s*, taking action *a*, and ending up in state *s'* for all states in *S* and actions in *A*. As defined by the equations, a MDP is designed so that action outcomes depend only on the current state and not past states, this is very important because if there is the need to keep track of past decisions this information must be present in the representation of the state.

An important aspect about MDPs is that rewards in the future are less certain than close-by rewards, this means the future rewards should be discounted by a γ factor, called the *discount factor* [7]. This factor goes from 0 to 1, 0 meaning that only the present matters, and a value approaching 1 meaning that the future rewards weigh more in the decision.

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

The objective of a MDP is to find an *optimal* plan, a policy function $\pi(s) \rightarrow a$ which for any state *s* in *S* will yield an action *a* which if taken will maximize the expected utility.

2.3 Q-learning

Q-learning defines a function $Q(s, a)$ which yields the expected utility after taking action *a* in state *s* and continuing optimally from that point on, or in other words, it yields the best possible score that we could achieve at the end of the game after taking action *a* in state *s*. $Q(s, a)$ can be defined recursively as

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q^*(s', a')$$

but, since we do not know $R(s, a)$ and $T(s, a, s')$ *a priori* we have to estimate $Q^*(s, a)$ and update that estimate as we explore the game and try different actions *a* at different states *s*, observing what the reward *r* and next state *s'* were. Taking this into account, we can iteratively update the estimate of *Q* with the following update rule:

$$Q(s, a) = Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

where *s* is the state we were in, *a* the action taken, *r* the reward received, *s'* the state we reached, γ the *discount factor*, and α the *learning rate*.

The learning rate specifies how much is the new experience taken into account when updating the estimate. A learning rate of 1 specifies that past estimates do not matter at all and the update will be equal to $r + \gamma \max_{a'} Q(s', a')$ since the two $Q(s, a)$ factors will cancel.

At first the estimate $max_{a'}Q(s', a')$ may be completely wrong, but with enough updates $Q(s, a)$ is guaranteed to start converging to the true function. [8].

The basic idea of the Q-learning algorithm is the following:

```
initialize Q[|S|, |A|] arbitrarily
observe initial state s

for episode in numEpisodes {
    select and execute an action a
    observe reward r and new state s'
    Q[s,a] = Q[s,a] + α(r + γmax_a' Q[s',a'] − Q[s,a])
    s = s'
}
```

2.4 Artificial Neural Networks

Artificial neural networks are an information processing paradigm loosely based on the way a human brain works. They are composed of small interconnected computational units called neurons, which compute a simple function based on their inputs and output a number that is in turn fed into more neurons of the network.

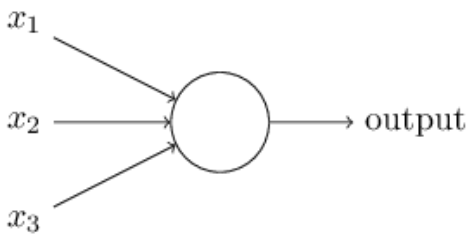

Figure 2.4.1: Single neuron [9]

A very common implementation of the function that each neuron computes looks as follows:

$$f(x) = K(\Sigma_i w_i x_i + b)$$

where K is usually referred as the *activation function, x* is a vector of real values, *w* is a vector of weights (one for each input), and *b* is a *bias* term. The activation function regulates the output of the neuron and there are several used. One of the most common activation functions is the *sigmoid* function, which scales the output between 0 and 1.

The most basic neural networks will contain a set of layers of neurons in which information propagates in only one direction. When a neural network contains one or more intermediate (*hidden*) layers (apart from the input and output layers) the network is called a *deep neural network*.

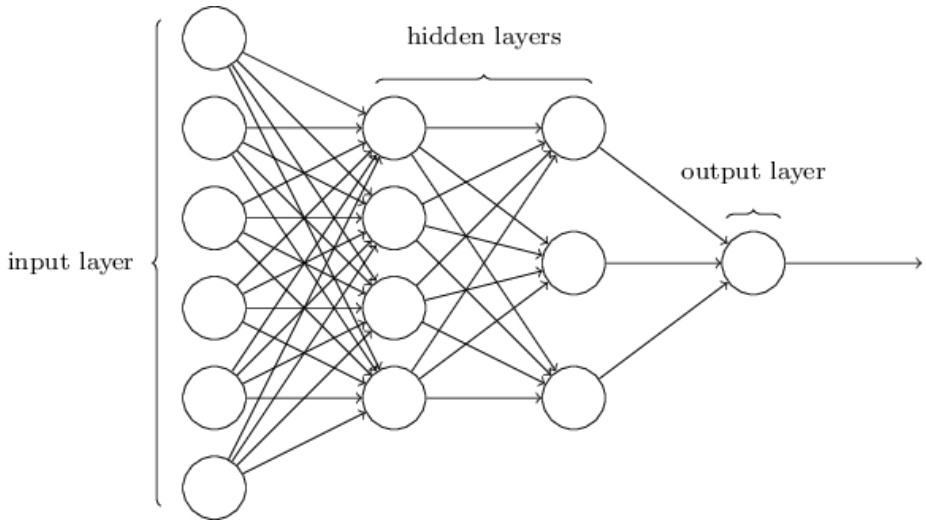

Figure 2.4.2: Deep neural network with two hidden layers and an output layer with just one neuron [9]

Each layer has its own set of neurons and makes a further abstraction of the layer before it. Maybe the most important fact about neural networks is that they can be used as function approximators and one of the core ideas is that they can approximate any function at all [9]. This fact makes them a very powerful tool in the field of artificial intelligence since many problems can be modelled as functions. The basic idea behind this is that the neural network has to learn an appropriate set of weights which will make an objective function (an error function) the minimum for a set of training samples (vectors of numbers which are fed as input). The weights are learned by feeding the training data into the neural network and using an algorithm (an optimizer) which will gradually adjust the weights so the objective function can reach a minimum when evaluated over a set of validation data (data which is not used to train on).

2.5 Deep Q-learning

The basic implementations of Q-learning use a table to store the Q-values for each possible state and action, however this quickly becomes unfeasible in games with large state spaces since the size of the table grows in an exponential fashion as the dimensionality of the state representation grows. Popularized by DeepMind [10], deep Q-learning utilizes the fact that neural networks are very good function approximators, so one can be used to approximate the Q-function with some modifications to the basic Q-learning algorithm. Now, instead of feeding the state *s* and the action *a* to the network and get a Q-value, what DeepMind does is feeding just the state and getting Q-values, one for each of the available actions. This is better from a performance perspective since instead of having to do one feed forward pass for each action to get the respective Q-values we only have to do one.

2.5.1 Experience Replay

One of the caveats of the Q-learning using neural networks is that guaranteeing convergence can be difficult as neural networks have their own set of properties and intricacies. One of the most important tricks to speed up the learning process is using experience replay. Experience replay is a method developed by Lin [11] in which we keep track of past experiences/transitions so we can "replay" them later as if they were actually happening. According to Lin, instead of training the network on single experiences and throwing them away (which is wasteful and we may be getting rid of important experiences which were difficult to obtain) we train the network repeatedly with past experiences. One

simple way in which we can implement this kind of behavior is by keeping track of a *replay memory* with past transitions and randomly sampling a batch to train the neural network with in each iteration.

2.5.2 Exploration VS Exploitation

In order to maximize the reward we must follow what we have learned is best, but we have to also give us a chance to learn it. A basic way of implementing such behavior is to define a parameter ε. Now, instead of always picking what we think is the best action, with probability ε we pick a random action. This ensures that, although most of the time we follow what we know is best, we occasionally try something random and see what happens. This type of approach is called an ε-greedy algorithm.

Finally, the basic idea of the deep Q-learning algorithm (with experience replay and ε-greedy exploration) is the following [6]:

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s

for episode in numEpisodes {

    select an action a {
        with probability ε select a random action
        otherwise select a = argmax_a' Q(s, a')
    }

    carry out action a
    observe reward r and new state s'
    store experience <s, a, r, s'> in replay memory D
```

```
    sample random transitions <ss, aa, rr, ss'> from replay memory D

    calculate target for each minibatch transition {
        if ss' is terminal state then tt = rr
        Otherwise  tt = rr + γmax_a' Q(ss', aa')
    }

    train the Q network using (tt − Q(ss, aa))^2 as loss

    s = s'
}
```

Chapter 3

Methods

3.1 Pacman Engine

Since this research was aimed at identifying differences between several state representations, a PacMan emulator like the one DeepMind uses was not suitable for the project. Instead, UC Berkeley's implementation was used [12]. Berkeley's implementation has several advantages that make the testing process easier. Firstly, it is written in Python, which is a very popular language in scientific research and has a very large community. Secondly, it allows for easy access to all game internals, which in turn lets the Q-learning algorithm extract any necessary information and derive different state representations. Lastly, since graphics are not needed during training they can be turned off, greatly improving training speed.

3.2 Architecture

Modules were built on top of Berkeley's implementation to implement a custom deep Q-learning algorithm and to provide easy testing, parameter tweaking, and statistics tracking. Different parameters can be specified when running the algorithm including the state representation to use, the Neural Network to use (if using a Neural Network-based Q-Function representation), and several other parameters

standard to Q-Learning like the learning rate, the discount rate, epsilon, and the number of training episodes. The game flow is controlled manually by the algorithm, which generates successors for each state or creates new games when needed.

3.3 Epsilon Decrease

To improve learning speed, the exploration rate (epsilon) was not set to a constant. Instead, an approach is taken inspired by DeepMind, in which epsilon is initialized to 1 and decreased overtime in a constant manner until a certain point (0.05 in this case). This allows the algorithm to explore the transition space actively at first and, as time passes and it learns what is good and what is bad, epsilon is decreased and less random decisions are made.

3.4 Persisted Experience Replay

As part of the algorithm, one of the optimization tricks attempted was to build a module dedicated to gather as much random (and distinct) experiences as possible and persisting them to a file. This file could later be used to initialize the replay memory in the Q-learning algorithm. Since the replay memory is already full of transitions from the beginning this could theoretically break biases toward first batches of transitions, as well as improving training speed considerably. Experiments were performed to test this hypothesis.

3.5 Algorithm

The final algorithm was implemented as follows:

```
load persisted or create new replay memory D depending on experiment
ε = 1

initialize Q-function {
    if table-based initialize with 0's
    otherwise initialize with random weights
}

create and observe initial state s
episodesSoFar = 0

repeat numEpisodes times {
    select an action a {
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s, a')
    }

    carry out action a
    observe reward r and new state s'
    store experience <s, a, r, s'> in replay memory D
    s = s'
    if s' is a terminal state then s = initial state

    sample random transitions <ss, aa, rr, ss'> from replay memory D
    update Q {
        calculate target for each minibatch transition {
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γ max_a' Q(ss', aa')

            if table-based then Q(ss, aa) = Q(ss, aa) + α * (tt − Q(ss, aa))
            otherwise if weight-based then w_i = w_i + α * (tt − Q(ss, aa)) * state[i]
        }

        if NN-based {
            train the Q network using (tt − Q(ss, aa))^2 as loss
```

```
            \ using batch of targets calculated above. Targets for actions
            \ different than a are left unchanged so error is 0.
        }

        ε = max(finalEpsilon, 1.00 - episodesSoFar / numEpsilonSteps)
    }
}
```

3.6 State Representations

A number of state representations were taken into account, some of which gave less "raw" data than others, but which still provided a lot more calculated information than state representations which consist of just the pixels.

All representations were normalized to values between -1 and 1, so as to give the same priority to all components and prevent larger numbers (because they move on a different scale) from causing greater impact than smaller numbers.

3.6.1 Common features

The following features were **common** to all state representations:

- **Legal actions:** represented as a subset of four 0's or 1's for each cardinal direction. 0 means "wall in the way" and 1 means "the action is legal".
- **Food around PacMan:** represented as a subset of 0's or 1's for each cardinal direction. 1 means "food is available one step away" and 0 the opposite.
- **Capsules around PacMan:** represented as a subset of 0's or 1's for each cardinal direction.

- **Ghost directions:** represented as a number between 0 and 3 encoding one of the four possible directions of each ghost. This number is normalized dividing by 4.
- **Scared ghosts around PacMan:** represented as a subset of 0's or 1's for each cardinal direction. 1 means "there is a scared ghost one step away" and 0 the opposite.

Unnormalized example:

1, 0, 0, 1 1, 0, 0, 1 0, 0, 0, 0 1, 3

3.6.2 "Binary" representation

Represents the state as a set of:

- **Ghosts around PacMan:** represented as a subset of 0's or 1's for each cardinal direction. 1 means "there is a scared ghost one step away" and 0 the opposite.
- **Closest food distance:** represented as an integer which corresponds to the length of the (legal) path to the closest food particle, normalized dividing by *max(boardWidth, boardHeight)*.

Complete unnormalized example:

1, 0, 0, 0, 5 1, 0, 0, 1 1, 0, 0, 1 0, 0, 0, 0 1, 3

3.6.3 “Positions” representation

Represents the state as a set of:

- **PacMan and Ghosts positions:** represented as pairs of coordinate integers and normalized dividing by *max(boardWidth, boardHeight)*.
- **Closest food distance:** represented as an integer which corresponds to the length of the (legal) path to the closest food particle, normalized dividing by *max(boardWidth, boardHeight)*.

Complete unnormalized example:

2, 5, 4, 4, 1, 2 **5** 1, 0, 0, 1 1, 0, 0, 1 0, 0, 0, 0 1, 3

3.6.4 “Distances” representation

Represents the state as a set of:

- ***x* and *y* distance components to each ghost**: Represented as integers. One pair (dx, dy) for each ghost. The sign of the values indicate the direction of the ghost with respect to PacMan (i.e. above/below, left/right).
- **Closest food distance:** represented as an integer which corresponds to the length of the (legal) path to the closest food particle, normalized dividing by *max(boardWidth, boardHeight)*.

Unnormalized example:

2, 5, 4, 4, 1, 2 **5** 1, 0, 0, 1 1, 0, 0, 1 0, 0, 0, 0 1, 3

3.6.5 “Inverse Distances” representation

Represents the state as a set of:

- **Inverse *x* and *y* distance components to each ghost**: Represented as the pairs defined in the *Distances* representation, but modified using the following expression:

$$\text{For x: } boardWidth * sign(dx) - dx$$

$$\text{For y: } boardHeight * sign(dy) - dy$$

- **Inverse closest food distance:** represented as an integer which corresponds to *max(boardWidth, boardHeight)* minus the length of the (legal) path to the closest food particle.

Unnormalized example:

2, 5, 4, 4, 1, 2 **5** 1, 0, 0, 1 1, 0, 0, 1 0, 0, 0, 0 1, 3

3.7 Map

The map layout used for all the comparison experiments looked as follows:

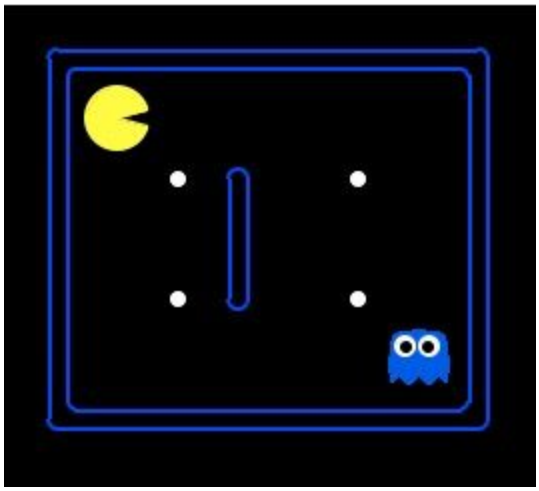

Figure 3.7.1: Medium Grid Layout. PacMan map (layout) used for training and comparisons.

A smaller map can help a lot to improve training speed, since there is a lot less variability than with the larger counterparts, and depending on the state representation the state space may also be considerably reduced.

3.8 Reward Scheme

Situation	Reward
Winning	+ 500
Lossing	- 500
Eating food	10
Eating Scared Ghost	200
Time Penalty	-1

3.9 Neural Network Topology

The topology for the neural network was defined as follows:

Layer	Dimensionality	Activation Function	Initialization
Input	| state |	N/A	N/A
Hidden	$\frac{\lvert \textit{input layer} \rvert + \lvert \textit{output layer} \rvert}{2}$ [1]	tanh	LeCun's Uniform[2]
Output	4	linear	LeCun's Uniform

The neural network accepts a state as the input and outputs 4 real values, each representing the predicted Q-value for each action in the action space (in this case 4 actions constitute the action space). Having each possible Q-value as output helps in efficiency since calculating the best Q-value requires only one feed-forward pass rather than 4 (one for each action).

[1] Integer division
[2] Uniform initialization scaled by the square root of the number of inputs

For the comparison between neural network topologies another neural network was built with two hidden layers:

Layer	**Dimensionality**	**Activation Function**	**Initialization**
Input	| state |	N/A	N/A
Hidden 1	$\frac{\lvert \textit{input layer} \rvert + \lvert \textit{output layer} \rvert}{2}$	tanh	LeCun's Uniform
Hidden 2	$\frac{\lvert \textit{Hidden } 1 \rvert}{2}$	tanh	LeCun's Uniform
Output	4	linear	LeCun's Uniform

3.10 Parameters

Basic common configurations were determined to be suitable as follows:

Persisted replay file (when applicable)	1.5GB corresponding to 362,802 unique transitions (i.e. tuple of the form <s, a, r, s'>).
Optimizer	Stochastic gradient descent
Learning rate	0.01
Batch Size	600
Target function	Minimum squared error
Episode count / Iterations	60,000[3]

[3] Although 60000 episodes were not necessary for some state representations it was defined as a constant so results could be comparable (using graphs for example).

3.11 Metrics

The following metrics were taken into account:

Average training loss	The average of the training loss during the last 100 episodes of training. The training loss corresponds to the sum of the evaluation of the target function (sum of squared errors in this case) for each example in a given batch.
Epsilon	The value of epsilon at any given moment.
Total wins	The number of wins since the beginning of the training session.
Total deaths	The number of deaths since the beginning of the training session.
Reward sum	The sum of the rewards during the last 100 episodes of training.

3.12 Trivial, handwritten implementation

As a means of comparison, a handwritten implementation was implemented, that is, a handwritten algorithm that decides which action to take based on the state. It simply picks a random action which would not lead him to die.

Chapter 4

Results

Training was carried out for each state representation using the *Medium Grid* layout shown in figure 3.7.1.

The results are shown below.

4.1 Without Replay File

4.1.1 Binary State Representation

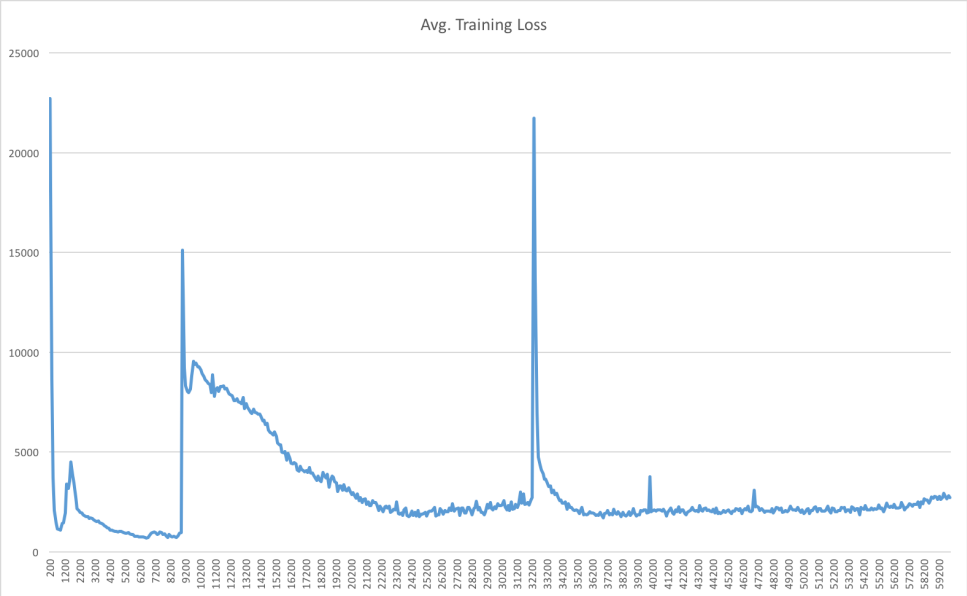

Figure 4.1.1: Training loss against iteration results for the *Binary* representation without replay file.

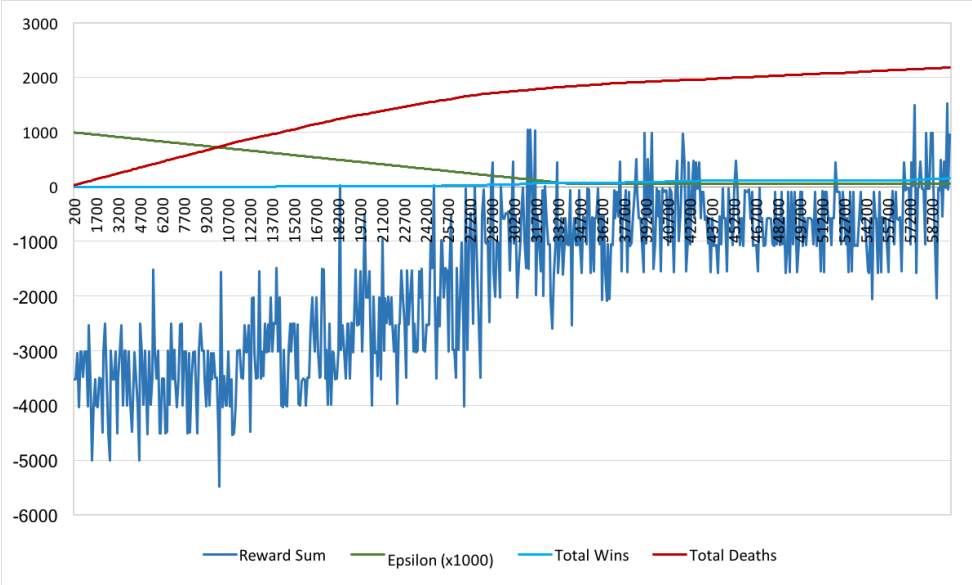

Figure 4.1.2: Reward Sum, Epsilon, and Total Wins against iteration results for the *Binary* representation without replay file.

Behavior notes: PacMan did not seek food in any situation, however he would never let the ghost catch him and would eventually end up winning after eating all food particles by chance.

4.1.2 Distances State Representation

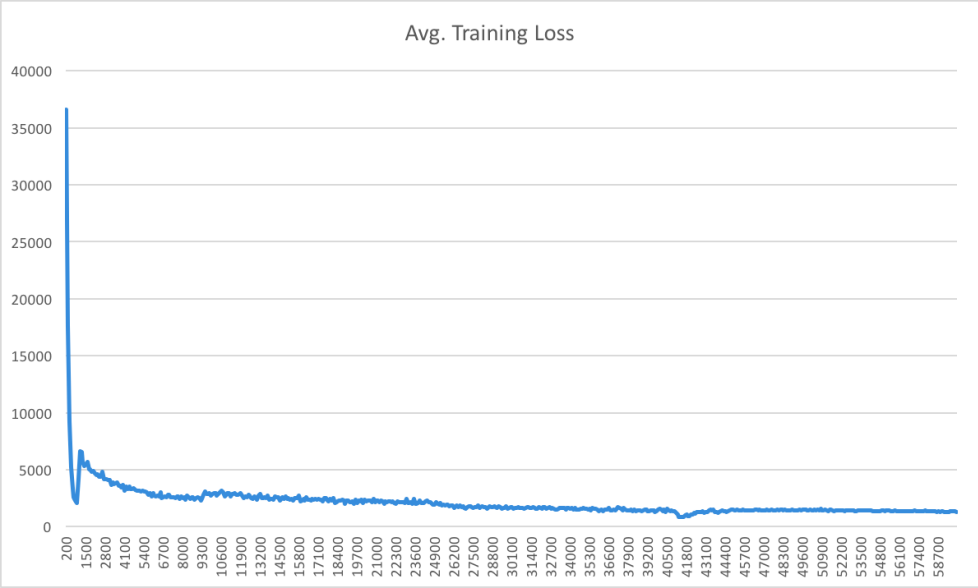

Figure 4.1.3: Training loss against iteration results for the *Distances* representation without replay file.

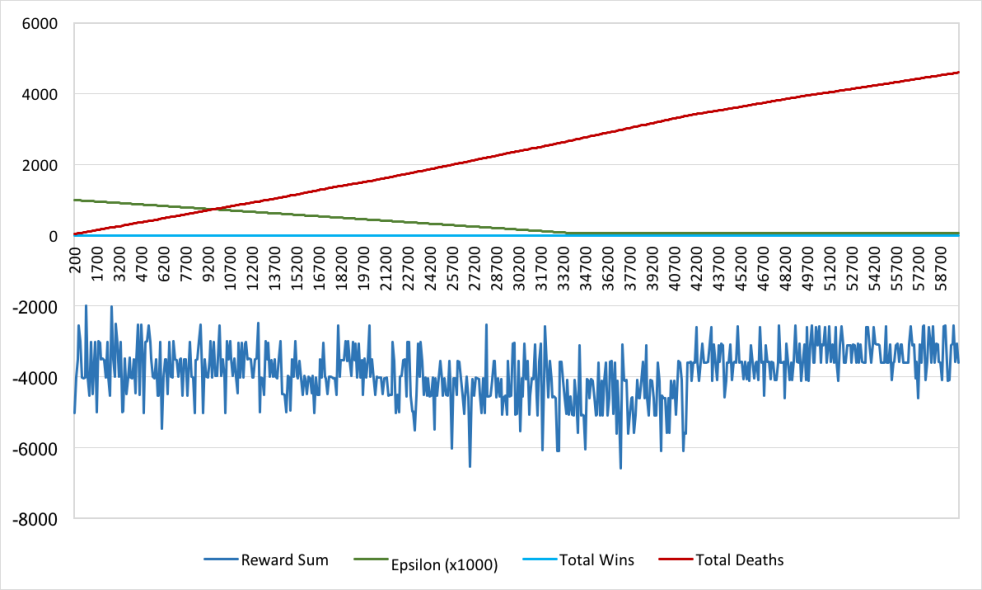

Figure 2: Reward Sum, Epsilon, and Total Wins against iteration.

Behavior notes: PacMan did seem to try to escape the ghost at first but when the ghost went near him he was not able to escape.

4.1.3 Inverse Distances State Representation

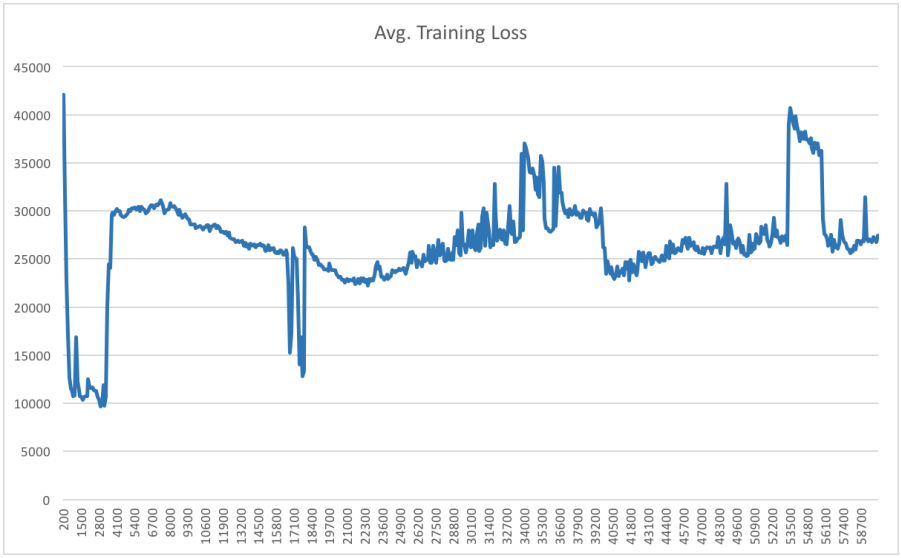

Figure 4.1.5: Training loss against iteration results for the *Inverse Distances* representation without replay file.

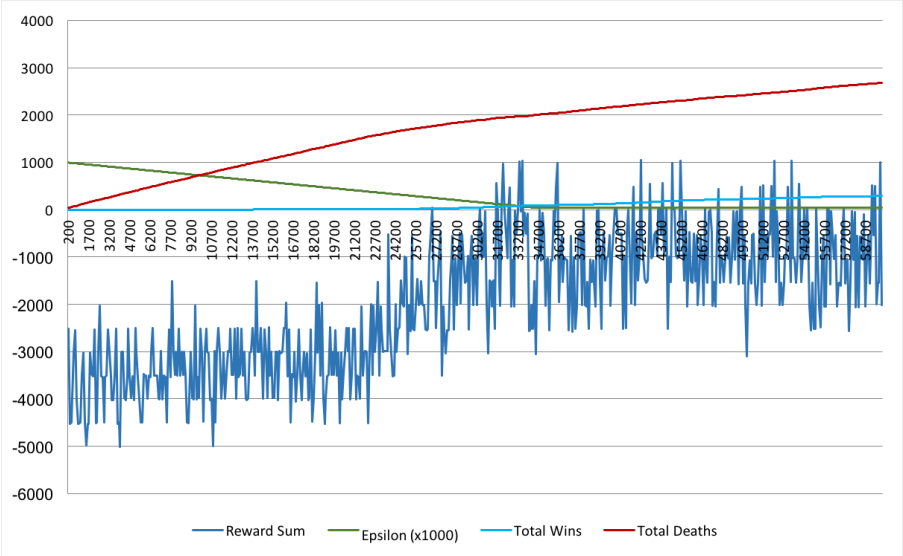

Figure 2: Reward Sum, Epsilon, and Total Wins against iteration.

Behavior notes: PacMan did learn to escape from the ghost but did not seem to seek food. He would consistently die, however, if being chased in a certain area of the map.

4.1.4 Positions State Representation

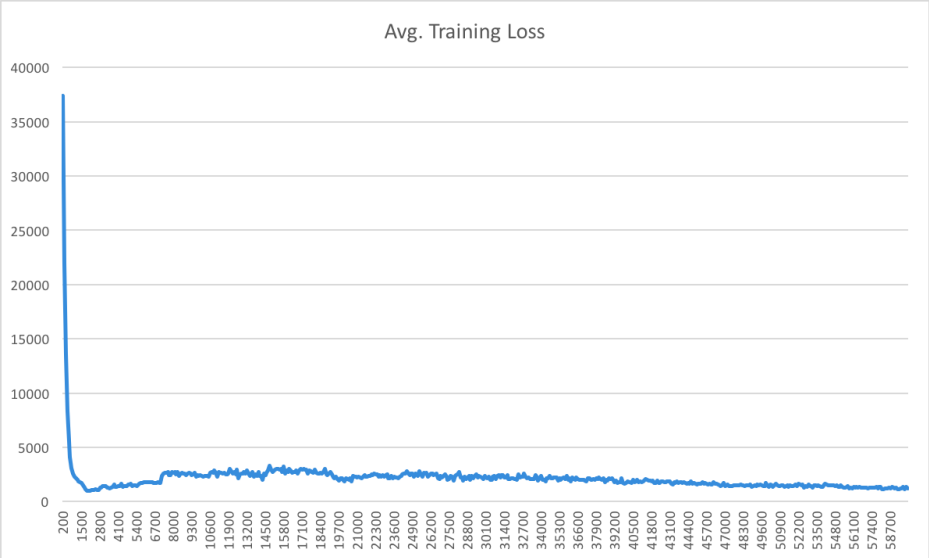

Figure 4.1.7: Training loss against iteration results for the *Positions* representation without replay file.

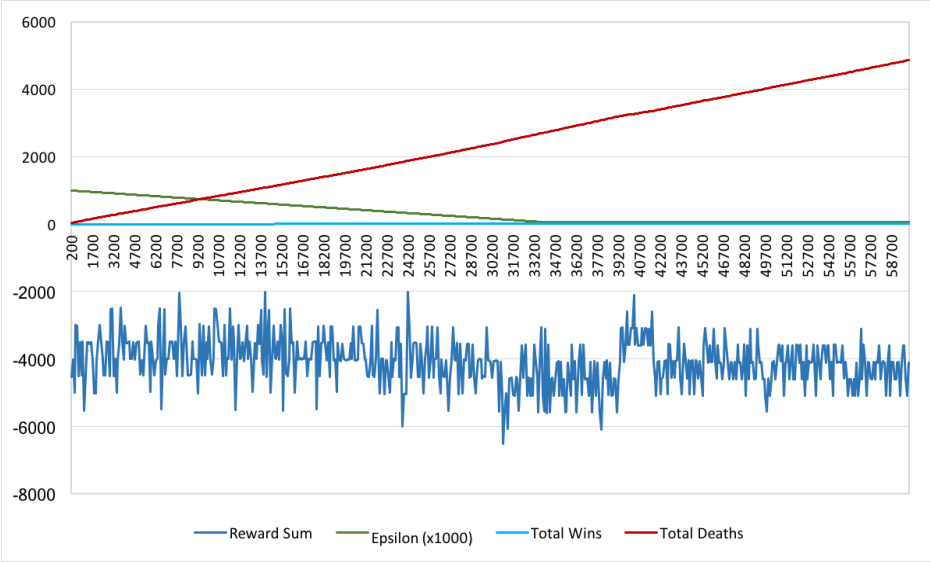

Figure 2: Reward Sum, Epsilon, and Total Wins against iteration.

Behavior notes: PacMan would just go up and down repeatedly until the ghost reached him and ate him.

4.2 With Replay File

4.2.1 Binary State Representation

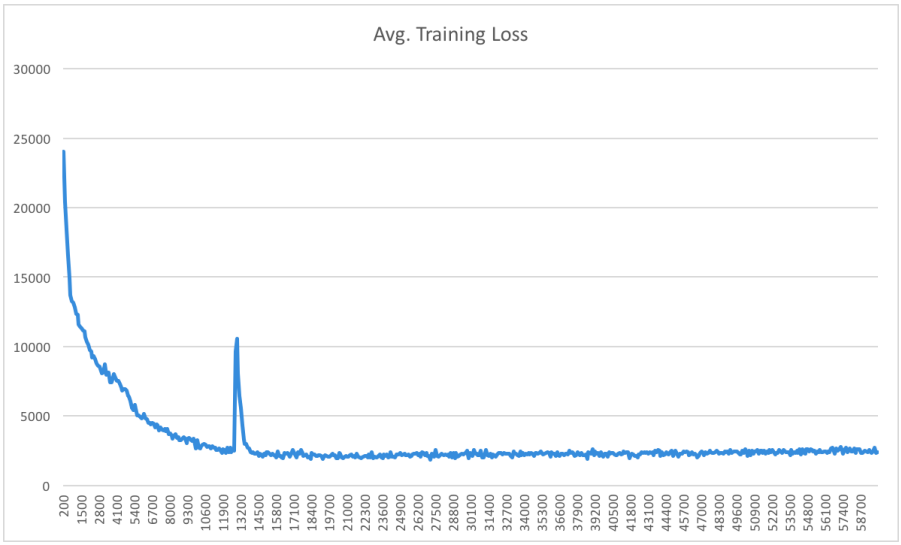

Figure 4.2.1: Training loss against iteration results for the *Binary* representation with replay file.

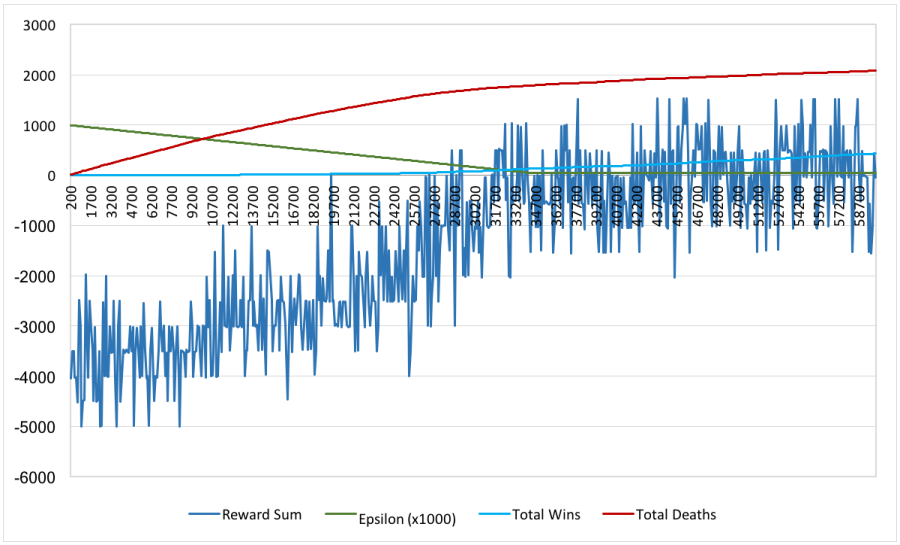

Figure 2: Reward Sum, Epsilon, and Total Wins against iteration.

Behavior notes: PacMan never died but he tended to go in circular moves, wasting time and lowering the score.

4.2.2 Distances State Representation

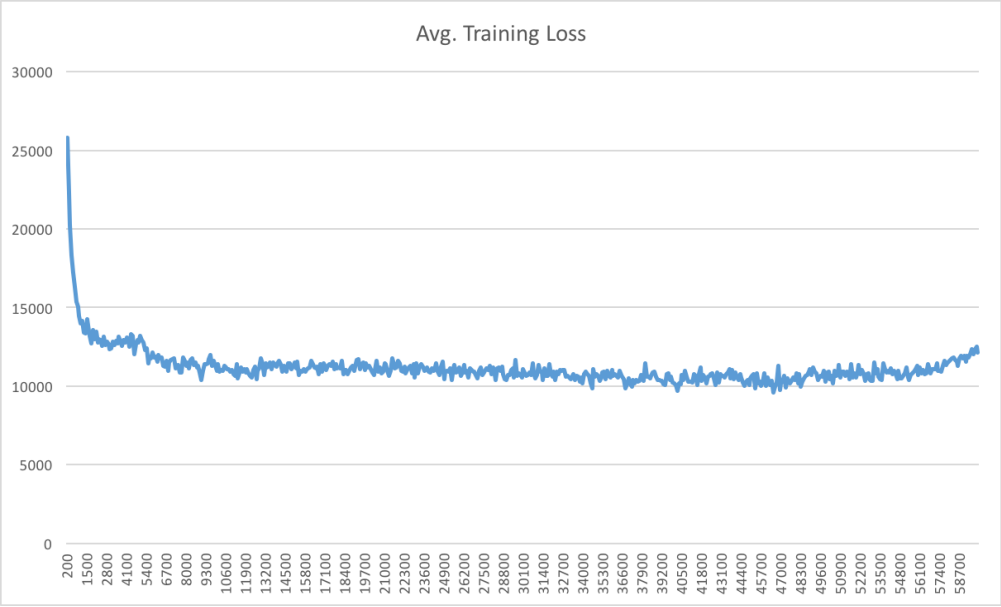

Figure 4.2.3: Training loss against iteration results for the *Distances* representation with replay file.

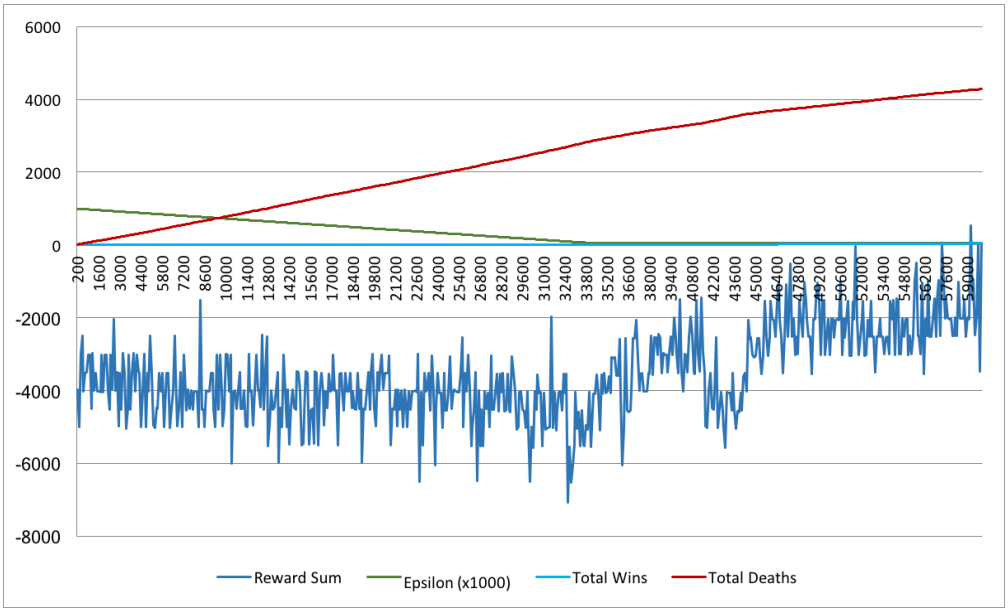

Figure 2: Reward Sum, Epsilon, and Total Wins against iteration.

Behavior notes: PacMan escaped from the ghost but only if it was one step away. While being chased PacMan also seemed to sometimes prefer to choose an action that made him eat a food particle that was one step away. However, PacMan did not seek food which was more than one step away. In some cases he would make the wrong decision and take an action which led him to death.

4.2.3 Inverse Distances State Representation

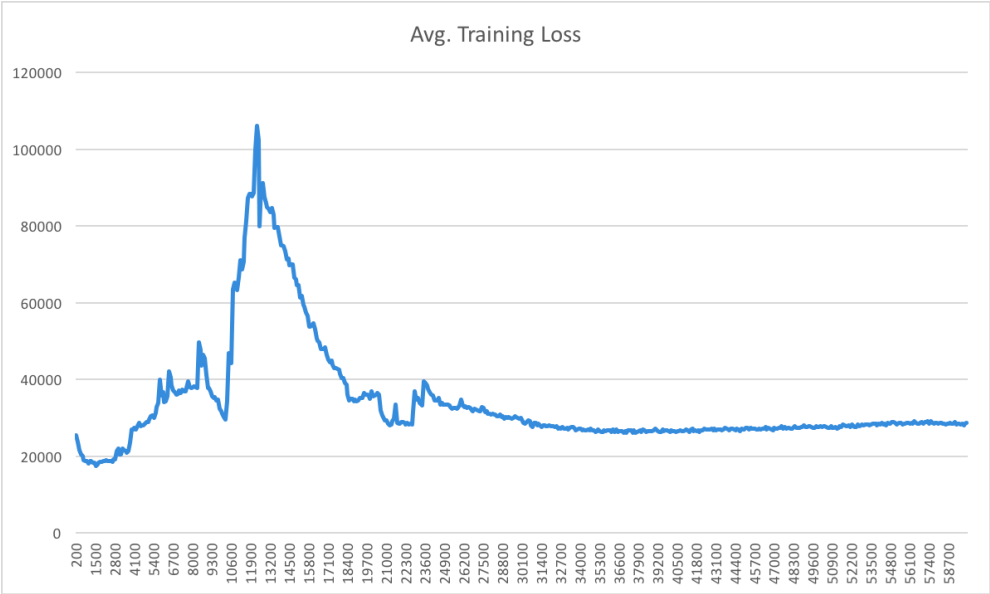

Figure 4.2.5: Training loss against iteration results for the *Inverse Distances* representation with replay file.

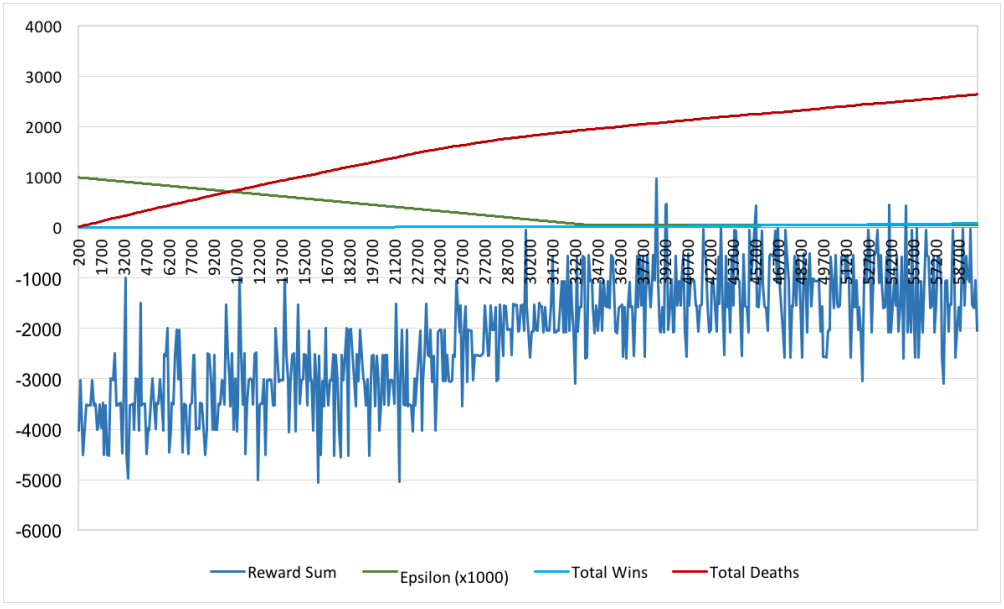

Figure 2: Reward Sum, Epsilon, and Total Wins against iteration.

Behavior notes: PacMan did not seem to look for food, but rather he only seemed to avoid the ghost when the ghost was right next to him. One particular detail was that the only (and certain) way PacMan would die was when PacMan was located in the top-right corner and the ghost right below him.

4.2.4 Positions State Representation

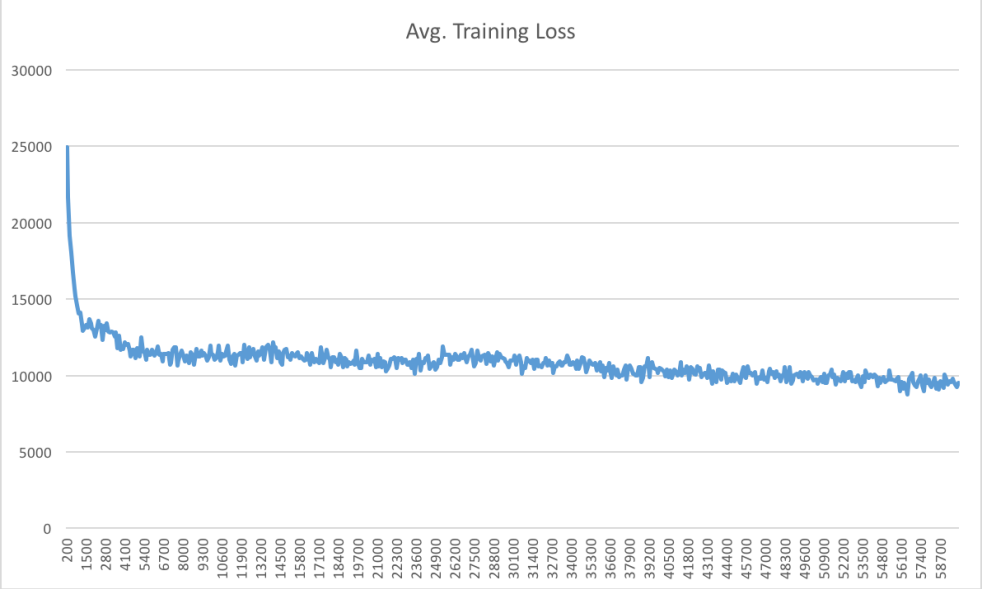

Figure 4.2.7: Training loss against iteration results for the *Positions* representation with replay file.

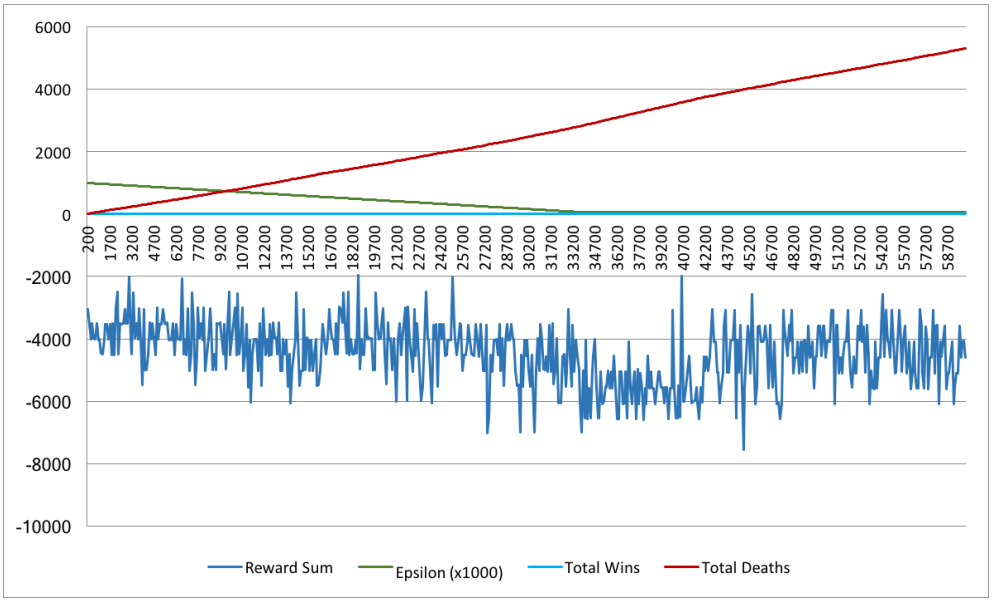

Figure 1: Training loss against iteration.

Figure 2: Reward Sum, Epsilon, and Total Wins against iteration.

Behavior notes: PacMan would just go up and down repeatedly until the ghost reached him and ate him.

4.3 State Representation Comparison

Training loss and reward sum were grouped into a single graph so comparison could be easier. The data were smoothed out this time using exponential smoothing to facilitate visualization.

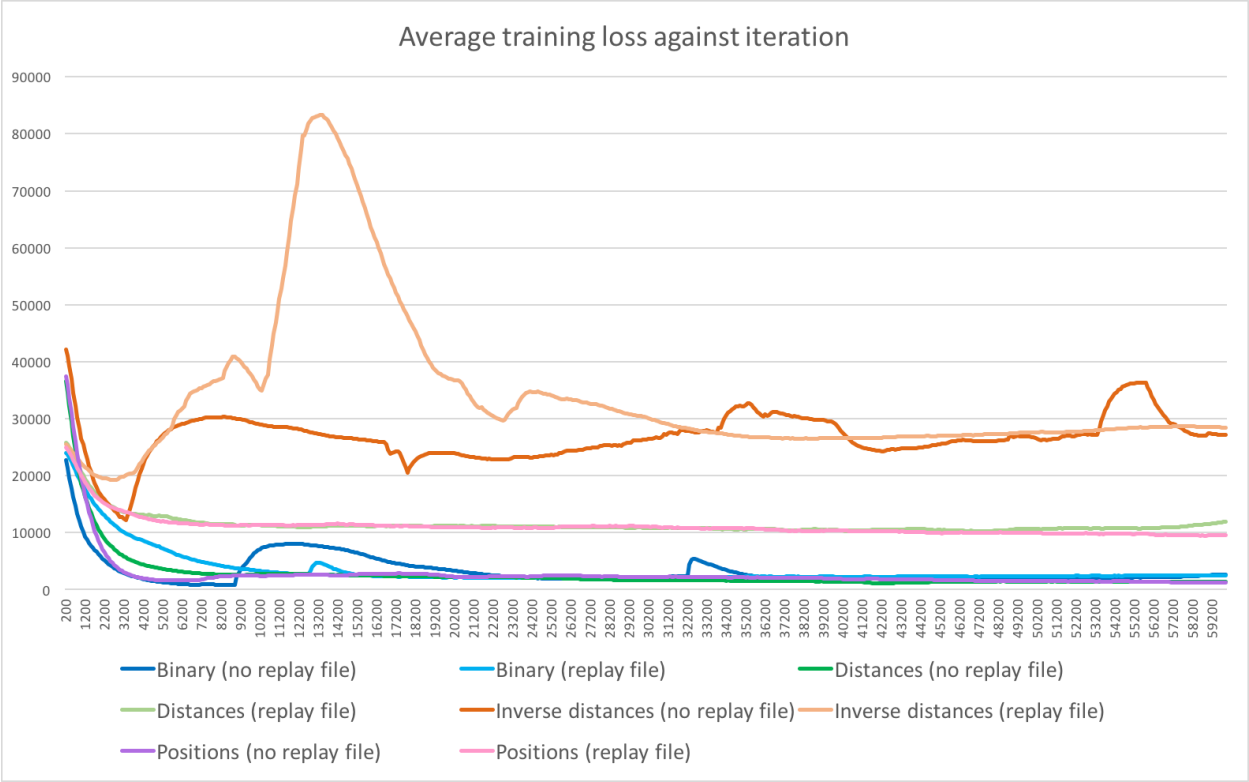

Figure 4.3.1: Training loss against iteration comparison between the different state representations.

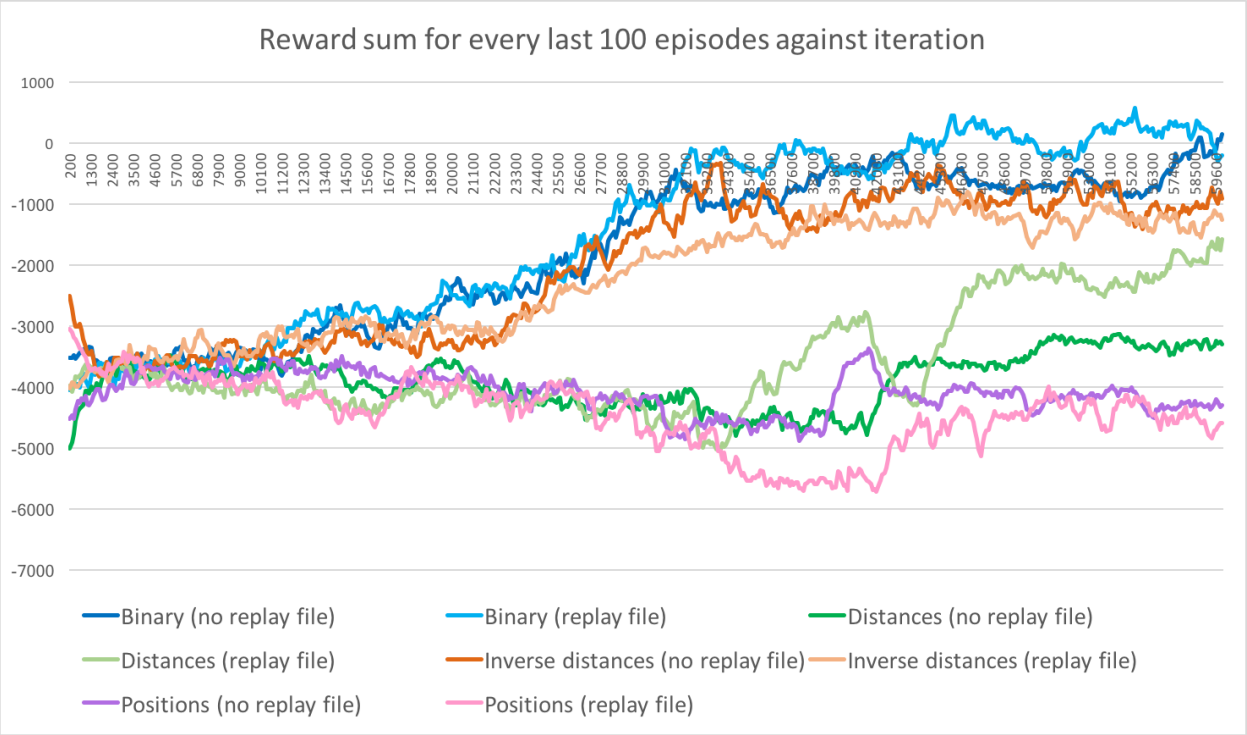

Figure 4.3.2: Rewards sum against iteration comparison between the different state representations.

4.4 Network Topology Comparison

In order to study how and if the topology of the network affected the training result significantly two training sessions were compared, one of them using the first neural network described in chapter 3.9 (one hidden layer) and the other using the second. The results are shown below.

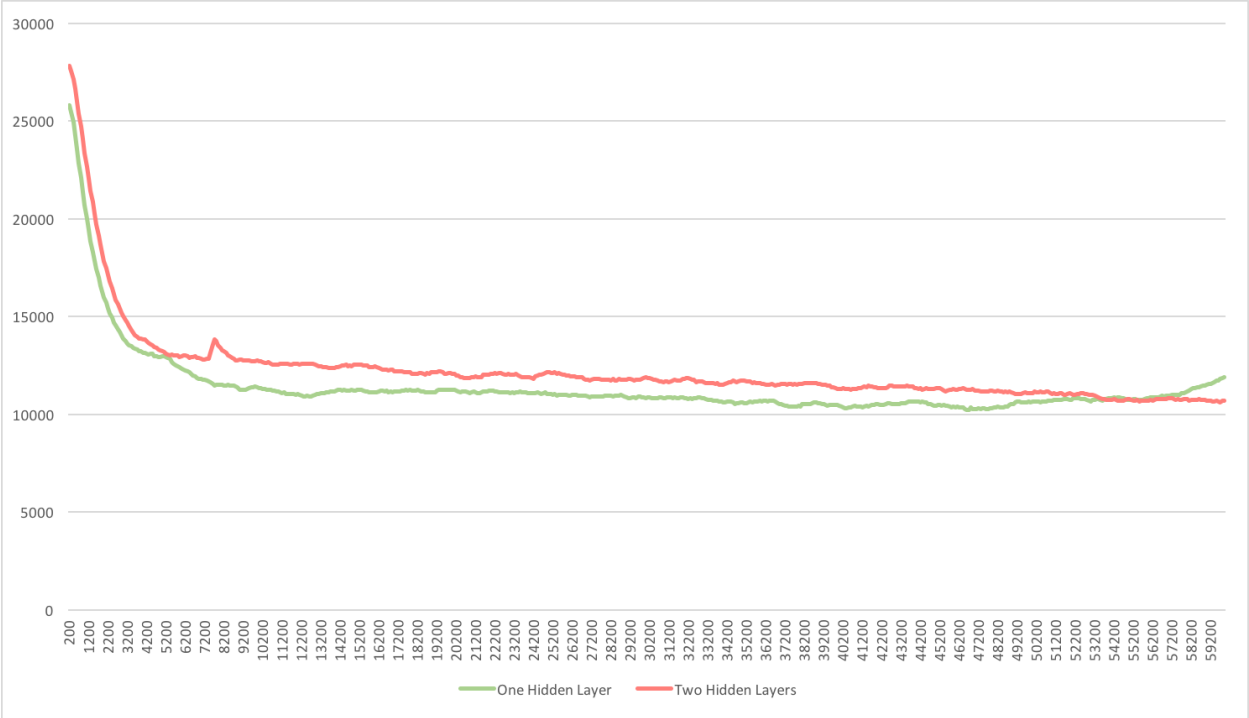

Figure 4.4.1: Training loss against iteration comparison between a network containing one hidden layer and a network with two hidden layers.

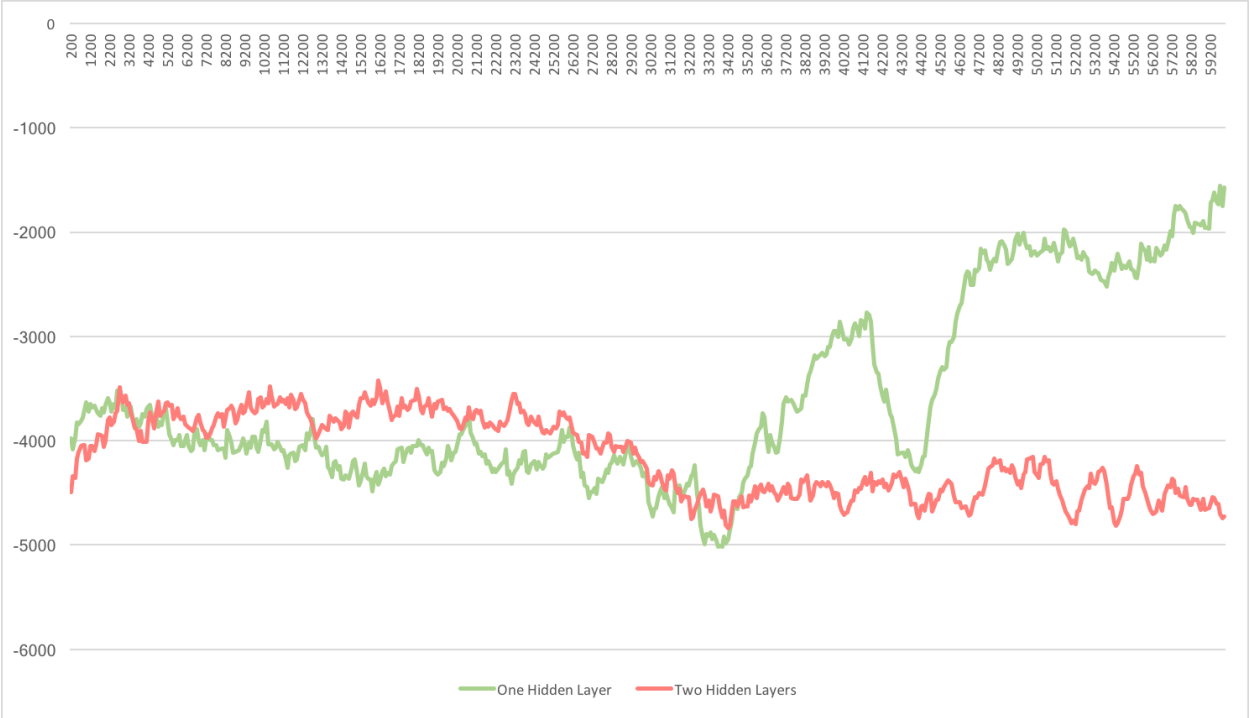

Figure 4.4.2: Rewards sum against iteration comparison between a network containing one hidden layer and a network with two hidden layers.

Figure 4.4.3: Total deaths against iteration comparison between a network containing one hidden layer and a network with two hidden layers.

4.5 Score Achievement

Persisted replay file	Network topology	State representation	Average score after 5000 games[4]	Percentage of games won
No	One hidden layer	Binary	490.62	100
		Distances	-515.45	0
		Inverse Distances	-76.091	43
		Positions	-511.79	0
Yes	One hidden layer	Binary	469.16	100
		Distances	-202.79	29.06
		Inverse Distances	-291.02	23.82
		Positions	-512.27	0
	Two hidden layers	Distances	-510.58	0

For comparison, the score achieved by the handwritten implementation after 5000 rounds was 472.39, winning 100% of the times.

[4] Games run with epsilon = 0

Chapter 5

Discussion and Conclusions

5.1 Deep Q-learning PacMan Agent and State Representations General Conclusions

Firstly, the objective of building an AI agent which used Q-learning with neural networks to play Pac-Man without any prior knowledge of the the rules of the game was achieved. The module was built on top of Berkeley's PacMan engine and successfully learned to play under restricted scenarios and without any previous knowledge of the game. It also successfully used a neural network to approximate the Q-function.

On the other hand, it could be seen that there was a clear and substantial difference between training sessions using different state representations. This leads to the conclusion that the information that an AI agent receives about the world affects the way it can train in a significant way. This was not necessarily obvious at first, since it could have been hypothesized that assuming all state representations are complete — meaning that they represent the world in its entirety for a particular task (e.g. avoiding ghosts and eating food directly around PacMan) — there would be no considerable difference in training as neural networks are very powerful function approximators. Nonetheless, now it can be seen that although neural network are indeed very good function approximators, the information that is fed as input to them during

Q-learning affects considerably both training time and the end result in terms of loss with respect to the objective function.

5.2 Trivial Handwritten Implementation

As shown in chapter 4.5 the handwritten trivial implementation did not perform better than then best score achieved by the deep Q-learning sessions. This shows that in this case the algorithm did take advantage of the available information in the simplest of the state representations (the *Binary* representation), in contrast to just choosing an action which did not lead PacMan to losing.

5.3 Iterations against training loss

One of the most relevant results visible in the graphs was that average training loss was not a continually decreasing variable, but rather had an often unpredictable behavior at some points during training. As an example, figure 4.2.5 shows that training loss actually increased during around 12000 iterations and only decreased to its initial values around 10000 iterations later. This kind of behavior was also visible with the *Binary* representation in figure 4.1.1. In fact, the lowest loss values were achieved in these cases during the first 2000 iterations but rewards were not good since *epsilon* was still very high. In theory, rewards could have also been good at that point if test games were run with an *epsilon* of 0.

It could be thought then, that training for less iterations was more convenient in these cases. This could hold true in the short term, however, and unless the neural network got stuck in a local minimum, training

loss should continue to decrease with enough iterations to a value less than or equal to the aforementioned values seen during the initial 2000 episodes of training. Another hypothesis could be that such a behavior was pure chance caused by a convenient weights initialization. To test this, two training sessions were run with the *Binary* state representation, no replay file, an initial and final epsilon of 1 and 0.98 respectively, and 2000 iterations. Neither of the two were able to perform better or even close to the session run in chapter 4.1.1, leading PacMan to losing in the first movements and getting average scores of around -504. This leads to think that, indeed, the behavior mentioned before was caused by chance. However, more test sessions would be needed to prove this correctly.

5.4 Replay File against No Replay File

It could be seen that having a pre-calculated and persisted replay memory did improve training time until one point for the *Binary* representation. Training loss decreased faster (and more consistently) for the training session that used the training file, but after episode 22000 both remained the same. Moreover, while at about iteration 40000 the session using the replay file started to have a consistent positive sum of rewards, the training session that did not use such file did not achieve this until the very last iterations. However, one striking result was that after being almost 2000 more than the session with no replay file the reward sum dropped significantly in the last 4000 iterations for the session using replay file, ending up even lower than the other session (figure 4.3.2). This supports the detail mentioned before: that more training episodes actually could make results worse in the short term. Another important aspect was that in the end the session that was not using the replay file scored very good after 5000 games using *epsilon*=0 (491.58), while the one using the replay file performed slightly worse with a score of 469.16.

On the other hand, for the *Distances* state representation having a replay file actually made training loss higher during the whole training, but in the end the session using the session using the persisted replay file got better reward sums and a much higher average reward after 5000 games using *epsilon*=0 (-219.70 against -515.34). This shows that training loss does not necessarily correlate with the rewards. This makes sense since one network could learn to predict better the more popular non-meaningful transitions and perform badly at predicting meaningful ones, while another could predict well important transitions and perform badly in non-meaningful ones, but still scoring higher as a consequence. If the last statement is the cause of the issue it could probably be improved by prioritizing the replay of transitions to the most meaningful ones, a problem which is addressed by DeepMind in their "Prioritized Experience Replay" paper [13].

For the *Inverse Distances* representation the effect was the opposite than with the *Distances* representation. Training with the replay file seemed to make the training results considerably worse. The scores after 5000 games were -76.09 with a win ratio of 42% and -291.02 with a win ratio of 23.82% for the sessions run without the persisted replay file and with it, respectively. The behavior during training, however, seemed to be very similar.

Finally, for the *Positions* representation the replay file seemed to make no difference.

In conclusion, it could be said that the only representation for which having the persisted replay file improved training clearly was for the *Distances* representation, while for the others the final score was actually worse and performance during training did not have considerable differences. So, in general, it could be said that results were not sufficiently conclusive to establish a clear verdict. However, it was seen that the replay file did have an impact in the results. For example, for the *Distances* representation,

the session which used the replay file caused PacMan to go in circles. In fact, in the first execution it ended up in a loop where he would not ever win or lose, so the session was run a second time. The second time PacMan did exhibit a similar behavior but he would eventually be able to get out of the loop and finish the game. The explanation to this seems to be that the replay file actually biased the results toward the transitions that it contained. The replay file was very large (362,802 transitions), but in each training episode only one of these transitions was being replaced with a new one. This was causing that even in episode 60,000 there was only a 17% chance of sampling a random transition and getting one of the transitions generated during the training session, which meant that the new transitions were taken very little into account.

So, does this mean a replay file like the one used should be avoided? Probably not, but it is important to take into account the relevance that is given to new transitions during training. In other words, that transitions from the replay file do not overwhelm new transitions. This can be achieved by limiting the amount of experiences that the replay file has in proportion to the number of episodes. It is also important to guarantee that transitions are unique, so as not to give false priority to some arbitrary transitions. It could be said then, that a persisted replay file could probably be used to save some time during training instead of waiting for the replay memory to fill up to a minimum of experiences before sampling batches. However, the file should be sufficiently small so that new transitions generated during training make the majority of the experiences the algorithm sees. This is something that would remain to be tested in a future occasion.

5.5 State Representations

In general, it is notable that simpler states performed better than the complex states, indicating that the state representation does have a considerable impact on the training result. Specifically, the *Binary* state representation was by far the best to perform as seen in chapters 4.3 and 4.5. In fact, it was the only state to receive a positive average score after 5000 games, and a relatively close-to-optimal result. On the other hand, the *Distances* representation was the runner up with an average score of -76.09 (in its best case), and its inverse version was next with a score of about 130 points less. Finally, the *Positions* representation had a score of -511.79 in its best case. The*, Binary, Distances,* and *Inverse Distances* representations all were able to win the game in at least some of the games, but the Positions representation did not win any game.

As mentioned earlier, this shows that even though neural networks are very good function approximators they are sensitive to the input and therefore the definition of the Q-learning state should not be taken lightly. However, all of this does not mean that a simple state representation should always be selected. Rather, the conclusion would be that the representation to select should be the simplest possible as long as it is complete. Being complete in this case would mean that the state needs to have all the information that would be needed in order for the neural network to draw the desired conclusions.

As an example, if it is desired that the neural network finds a conclusion like "it is better that PacMan does not enter the tunnel in front of him because there is a ghost behind him and a ghost entering at the other end", the only representation that would work would be the *Positions* representation. This is because to know where the tunnels are it would be needed, either to know the absolute position of PacMan and the ghosts on the map and sufficient iterations for PacMan to "infer" the locations of the tunnels implicitly, or to include tunnel information explicitly in the state representation. Given that case, it could be inferred that it would be easier to train the network using the latter option, however, the tradeoff being that the

network would not learn things that the developer did not take into account from the beginning. This is the reason why DeepMind does not handcraft the state representation, but rather uses the pixels from the game. Using the pixels gives the network all the information that it could possibly need to learn any conceivable technique that would lead to maximizing the score. However, it also means that a lot more training and optimizations would be needed for the algorithm to actually learn something useful. In these cases, convolutional neural networks are used to sort this issue out while taking advantage of the graphical nature of the game.

A decision has to be made then: how much information should be included in the neural network's input (state representation in this case)? It could be concluded that it depends. If it is desired to get the maximum score / classification accuracy or the minimum loss (i.e. if a regression is the case) then it is definitely useful to include all information the neural network could eventually take advantage of. However, if the goal is to get fast training speeds and a previously established acceptable accuracy / score / loss then handcrafting a simpler state that targets the specific needs could be enough.

5.6 Inverting State Numbers

One of the experiments was to run training with both a regular distance (i.e. lower distance means PacMan is closer to a ghost) and an inverted version (i.e. higher numbers mean PacMan is closer to a ghost). This was done with the objective of finding out whether it is better to avoid numbers close to 0, as they cancel out the weights of the network. After running the experiments interesting results were found. While the *Inverted Distances* representation definitely achieved higher scores faster than its *Distances* counterpart (see chapter 4.3), final scores after 5000 games using epsilon=0 were not consistent when

using the persisted replay file and not. However, when not using the replay file (which resembles the traditional approach to Q-learning) scores were significantly different (-515.45 against -76.09 for the *Distances* and *Inverse Distances* representations respectively). It would seem, then, that the inverse representation does improve the training results. Nonetheless, it would be probably valuable to test this with longer training sessions and to also run multiple of them in order to mitigate any variations given by the random initialization of the weights and the non deterministic nature of the game itself.

5.7 Network Topology

Finally, experiments were also run with different network topologies to see if it would signify a considerable difference in the training results. The results showed that it absolutely affected the training results notably. Namely, the larger version of the network performed far worse than its smaller counterpart. While the network with only one hidden layer was able to learn in time to be able to win in about 30% of the occasions, the network with two hidden layers did not perform any better than an agent picking random actions. Does this mean that it is always preferable to choose smaller networks to larger ones? The answer would be no. Larger networks allow for approximation of more complex functions, however, using a large network for a simple function would be very wasteful in terms of resources, so a decision to add another layer to the network certainly adds another level of abstraction but also another layer of complexity, which is another set of weights that must be trained. In conclusion, adding one layer to the network does represent a non-negligible difference in training time.

Bibliography

[1] D. Silver *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[2] S. Higginbotham, “Inside Facebook’s Biggest Artificial Intelligence Project Ever,” *Fortune*. .

[3] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[4] “Google Self-Driving Car Project,” *Google Self-Driving Car Project*. [Online]. Available: http://www.google.com/selfdrivingcar. [Accessed: 28-Aug-2016].

[5] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[6] T. Matiisen, “Guest Post (Part I): Demystifying Deep Reinforcement Learning,” *Nervana*, 21-Dec-2015. .

[7] T. van der Ouderaa, “Deep Reinforcement Learning in Pac-man,” University of Amsterdam, 2016.

[8] F. S. Melo, “Convergence of Q-learning: a simple proof.”

[9] M. A. Nielsen, *Neural Networks and Deep Learning*. 2015.

[10] V. Mnih *et al.*, “Playing Atari with Deep Reinforcement Learning,” *ArXiv13125602 Cs*, Dec. 2013.

[11] L.-J. Lin, “Reinforcement Learning for Robots Using Neural Networks,” Carnegie Mellon University, 1993.

[12] “Berkeley AI Materials.” [Online]. Available: http://ai.berkeley.edu/home.html. [Accessed: 05-Dec-2016].

[13] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay,” *ArXiv151105952 Cs*, Nov. 2015.