Informatica — Universiteit van Amsterdam

Bachelor Informatica

University of Amsterdam

Deep Reinforcement Learning in Pac-man

Tycho van der Ouderaa

June 10, 2016

Supervisor(s):

Dr. Efstratios Gavves
Matthias Reisser

Abstract

In recent years, major advances in Artificial Intelligence have been made. Developments in the field of parallelism and distributed computing have led to significant results in artificial intelligence. At the same time, progress in Reinforcement Learning has brought Artificial Intelligence a step closer to general intelligence.
In this paper, a model capable of learning to play the game Pac-man is proposed. The technique of Deep Q-learning is applied, whereby Reinforcement Learning methods are combined with Deep Learning. The model finds an action-selection policy, on the basis of observations of the game grid and game score, without additional knowledge of the inner workings of the game. The study strengthens the idea that Deep Reinforcement Learning is a valuable method for finding an action-selection policy for dynamic stochastic systems.

Contents

Introduction

The past years have seen increasingly rapid advantages in Artificial Intelligence. This year, the AlphaGo algorithm [21] has won against Lee Sedol during a 5-game tournament in the game Go [2], which is seen as a landmark in artificial intelligence research. The victory followed an algorithm capable of learning to play a range of Atari games on human level performance solely by looking at the pixel values of the game screen and the score values.
What these achievement have in common is that they all use Reinforcement Learning. In this study, a state-of-the-art Reinforcement Learning technique called Deep Q-learning is evaluated by applying it to the game Pac-man.
The algorithm which learned to play a multiple of Atari games was created by Google [14] [13] in 2015 and used Deep Reinforcement Learning. However, the algorithm failed to successfully learn to play the game Pac-man (Ms. Pac-man Atari 2600). Creating such self-learning model which can play Pac-man is yet an unsolved problem.
Pac-man is one of the most iconic arcade video games, which was originally developed by Namco in 1980 [15]. Since its introduction, countless remakes were released. When playing the game one controls a dot-eating character called Pac-man through a maze. The maze is filled with dots (representing food), which reward the Pac-man with a positive score when eaten. Collision with one of the player-chasing ghosts results in a loss and consequently the end-of-game. After eating special capsules the player has the opportunity to also eat ghosts (resulting in a positive reward) for a short period of time. These capsules are generally found in the corners of the game grid. Finally, the game is won by eating all dots.
The original version of Pac-man behaves in a strict deterministic manner. However, in some later versions of the game, including the implementation used in this study, ghost movement is stochastic. Also, the Pac-man implementation used in this study does not include multiple lives.
An agent situated in this environment performs actions in discrete time steps. Every time step, the environment undergoes a stochastic transition of state depending on the action chosen by the agent. Every state transition results in a reward.
A completed Pac-man game forms a sequence of state transitions, actions and rewards. The goal of the agent is to find an action-selection policy maximising the cumulative reward over this sequence.
This study evaluates the effectiveness of Deep Q-learning used to find an optimal policy to win the game Pac-man and maximise its game score. Moreover, the findings in this paper can be applied to solve a wide range of problems such as autonomous inverted helicopter flights [16] and learning locomotive skills on dynamic terrain [18]. The method described focuses on Reinforcement Learning, whereby the context of the Pac-man game is just a vehicle to investigate the effectiveness of Deep Q-learning and show its results in a more concrete way.

CHAPTER 1

Theoretical background

1.1 Reinforcement Learning

There is a strong relationship between Reinforcement learning (RL) [22] in computer science and the concept of conditioning in behavioural psychology. Different theories in literature [24] suggest that animal behaviour can partially be explained by associative processes in the animal mind resulting from experiences in association with pleasure [24]. These associative processes in combination with reinforcement may lead to behavioural change. Positive and negative reinforcement respectively are synonyms for reward and punishment.
In computational learning theory, the concept of reinforcement is applied to computational pattern-recognising. RL can be used in situations where the probabilities and rewards are unknown.
RL differs from supervised learning where patterns are learned on the basis of labeled examples. In RL there is interaction between the algorithm or agent and the system to be learned. The cycle between a RL agent and an environment is shown in the *Figure 1.1*. The RL agent acts in an environment from which it receives observations and numerical rewards (scalars). Based on the acquired information (observations with corresponding rewards) the agent updates its policy function used to assess the best action.
One of the main challenges in RL is to find the optimal action-selection policy. Q-learning, among other techniques (*Section 1.2*), can be used to effectuate such a policy in a RL setting.

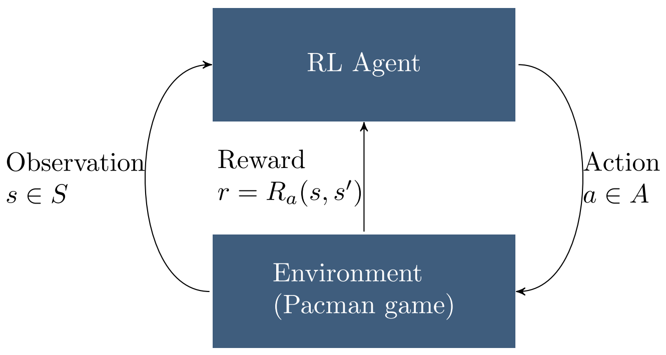


Figure 1.1: Reinforcement learning cycle

1.2 Markov Decision Processes

Markov Decision Processes (MDPs) can mathematically formalise decision making problems. It is useful to describe a decision making problem as a MDP as it reduces a problem to its essential

aspects and thereby helps to understand the problem and to formulate a solution. MDPs are commonly used and make up a general theory to describe problems in Reinforcement Learning. A MDP is a discrete time stochastic control process and forms a model consisting of a set of all possible states S, all possible actions A and a reward function $R_a(s, s')$ for each possible state transition. States are 'memory-less' and contain all relevant information about the past (also known as the Markov Property) [22].
MDPs can be both deterministic and stochastic. In a stochastic MDP, as is the case in this study, a probability distribution exists over next possible states given a state s and an action a. A finite sequence of states, actions and corresponding rewards together form an episode in the MDP. The total reward of an episode is the sum of all rewards gained every state transition.

Because rewards in the future are less certain than close-by rewards, the expected future reward should be discounted by a γ factor. The discount factor γ satisfies $0 \leq \gamma < 1$. If the discount rate equals $\gamma = 0$, only the immediate rewards are taken into account, whereas the more γ approaches one rewards further in the future (long-term) will gradually weigh more. The total discounted future reward from the beginning of an episode ($t = 0$) to end ($t = n$) becomes

$$\sum_{t=0}^{n} = \gamma^t R_a(s_t, s_{t+1})$$

A solution for a MDP can be defined as a policy function π mapping each state $s \in S$ to an action $a \in A$ and maximising the cumulative reward of an episode. Many techniques have been proposed [22] to find an optimal policy for MDPs, such as Monte Carlo search, Value Iteration, Policy Iteration and Q-learning.

1.3 Q-learning

In Q-learning [27] the function $Q_\theta(s, a)$, parametrised by θ, is defined as the expected discounted future reward in a state s given an action a. The function returns the Q-value of a certain state-action pair. The Q-value can be calculated with the function $Q_\theta(s, a)$ and represents the "quality" of the action a from state s. It was proven that for a finite MDP the Q-value always converges [26]. However, this theoretical proof of convergence does not mean convergence is guaranteed in practical settings, as there typically are constraints on training time.
The Q-function is updated to match the expected discounted future reward using the function

$$Q_\theta(s_t, a_t) \leftarrow Q_{theta}(s_t, a_t) + \alpha * (r_{t+1} + \gamma * \max_a Q_{\theta-}(s_{t+1}, a) - Q_\theta(s_t, a_t))$$

.

where $Q_\theta(s_t, a_t)$ is the Q function, α is a learning rate, s and a are respectively the state and the action and r_{t+1} is the reward corresponding with the current state transition.
An action-selection policy π for a state s can be formed by selecting the action a maximising Q for all actions possible from s by

$$\pi(s_t) = \arg\max_a Q(s_t, a_t)$$

.

1.4 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a variation of a regular feed-forward neural network. The architecture of a CNN is designed to exploit the local-connectivity typically found in high dimensional data such as images. CNNs have been widely applied in computer vision and image recognition [11] [23]. CThe discoveries in visual mechanisms of living organisms found in biological research inspired CNNs. Research performed by dr. H. Hubel and T.N. Wiesel in 1968 [10] gave insights into the behaviour of cells in the visual cortex of advanced living organisms. The study showed that cells in parts of the monkey brain that are responsible for visual processing are sensitive to sub-regions of the visual field, called receptive fields.
In contrast to the fully-connected layers in regular neural networks, a layer in a CNN uses convolutions on its input when computing its output. Each layer has a range of convolutional kernels or filters, which together effectively classify features. The weights of these kernels are updated by stochastic gradient descent.

1.5 Deep Q-learning

Originally Q-learning uses a data table containing the Q-values of state-action pairs. However, due to the enormous size of the state space in Pac-man, the required amount of memory and the number of iterations needed for convergence make it practically impossible to set-up such table. The advantage of using Deep Q-learning (DQN) is that the Q-values for state-action pairs can be interpolated by a convolutional neural network instead of a data table.
A state transition from state s to s' s a result of action a and the obtained reward r together form an experience tuple $\langle s, a, r, s' \rangle$. At the beginning of the learning progress, the Q-network returns an arbitrary number. After a number of experiences, the algorithm updates Q so it converges to the expected discounted future reward.

An intuitive approach (*Figure 1.2*) to calculated the Q-value for a state-action pair (s,a) would be to feed (s,a) as an input to a neural network. This structure can be improved by only feeding the neural network a state s instead of (s, a) and defining the output of the network as a vector of Q functions for all possible actions from s, as shown in *Figure 1.3*.

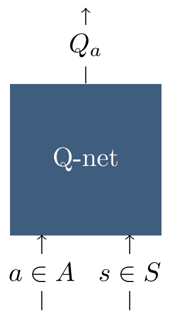


Figure 1.2: Naive architecture

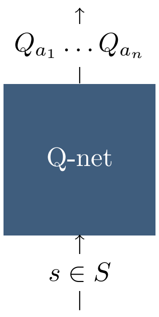


Figure 1.3: Used architecture

The policy π of a RL agent using the latter improved method is to select the action with the same index as the index of the highest value in the vector outputted by a feed-forward pass of the current state through the neural network.

1.5.1 TargetNet and Q-net

In Deep Q-learning two neural networks are used: the *Q-network* and the *target network*. Every training iterations, the Q-network is updated by back-propagation and the earlier described (*Section 1.5*) update function. The target network is used when Q-values are calculated, and

is an earlier copy of the Q-network. Once every set amount of iterations the target network is substituted by an up-to-date version of the Q-network.
It is proven [13] that this technique, in which the most up-to-date Q-network is not used to calculate Q-values, improves the performance of the Deep Q-learning algorithm significantly.

1.6 Experience Replay

Updating the Q-function with subsequent state transitions violates the assumption the states are independent and identically distributed (i.i.d.). This will reduce the general predictive capacity of the learning algorithm. Experience Replay (ER) [1] was proposed to address this issue. It has been demonstrated that the use of Experience Replay can improve the effectiveness of algorithms for learning optimal control policies such as Q-learning.
In ER experience tuples $\langle s, a, r, s' \rangle$ are stored in a database called 'replay memory'. A mini-batch of experience tuples are sampled from replay memory, once every set number of iterations.
This re-use of previous state-transitions can be compared with the replay of experiences found in the hippocampus in animal brains. Hippocampal replay is the phenomenon found in a range of species where sequences of neural activations are observed to occur repeatedly [7].

1.7 Exploration vs Exploitation

Intrinsically to RL is the *exploitation vs exploration* dilemma. The optimal policy for an agent is to always select the action found to be most effective based on previously encountered situations. To improve or learn a policy the agent must encounter situations it has never observed before. Thus, a trade-off arises between performing the action that leads to the highest expected future reward (exploiting) and trying actions that lead to new situations with the goal of acquiring new knowledge (exploring).
There are different ways to handle the exploration vs exploitation dilemma. The most simple method is *greedy* selection in which the algorithm always selects the action found most effective (always exploit). When applied to Q-learning this selection strategy will result in the selection of the action

$$a_t = \pi(s_t) = \arg\max_{a \in A} Q_t(Q, a)$$

An extension of this greedy selection is the ϵ-*greedy* strategy. The ϵ-greedy method defines an ϵ value, satisfying $0 <= \epsilon < 1$, to balance exploration and exploitation. In ϵ number of the times the algorithm selects a random action within the action space to explore new situations. Consequently, in $1 - \epsilon$ of time the algorithm exploits by selecting the action with the highest Q value. The ϵ-greedy strategy can be defined as follows:

$$\begin{aligned} With\ probability\ 1 - \epsilon :\ pick\ random\ action \\ With\ probability\ \epsilon :\ pick\ best\ action\ a_t = \pi(s_t) = \arg\max_{a \in A} Q_t(Q, a) \end{aligned}$$

CHAPTER 2

Method

2.1 System Architecture

The set-up used in this study [17] consists of different components. The Pac-man implementation forms the environment as described in *Section 1.1.* The DQN algorithm implemented in Python and represents the RL agent. The game sends game states - observations - and rewards to this algorithm. The observations of state are carefully designed to match a state representation close to one of a human. By ensuring that no hand-made features (e.g. coordinates of key-components or path-finding information) exist in state-representation the generality and thereby relevance of the method is maintained in a broad context.
The CNNs (Q-net and targetnet) are implemented in Tensorflow 0.8 [9]. Updated models of the Q network are regularly stored in *checkpoint* (back-up) files.

2.2 Pac-man Implementation

As described in *Section 1.1*, information about the state, rewards and actions are exchanged between the environment and an agent in a RL setting.
An implementation of the Pac-man game is needed which enables relative easy exchange of states, actions and rewards between the game and the algorithm. In addition, to effectively learn a policy using DQN a vast amount of iterations will be needed. Hence, it is helpful during training if simulations on the Pac-man game can run faster than real-time.
The used Pac-man implementation created by UC Berkeley meets both requirements and is therefore specially suited for RL research [6][8].

2.2.1 Levels

The Pac-man implementation by UC Berkely has native support for different maps or levels. This paper focuses on the medium-map (*Figure 2.1*) which is a game grid of 20×11 tiles. The game grid in the original version of Pac-man has a size of 27×28.

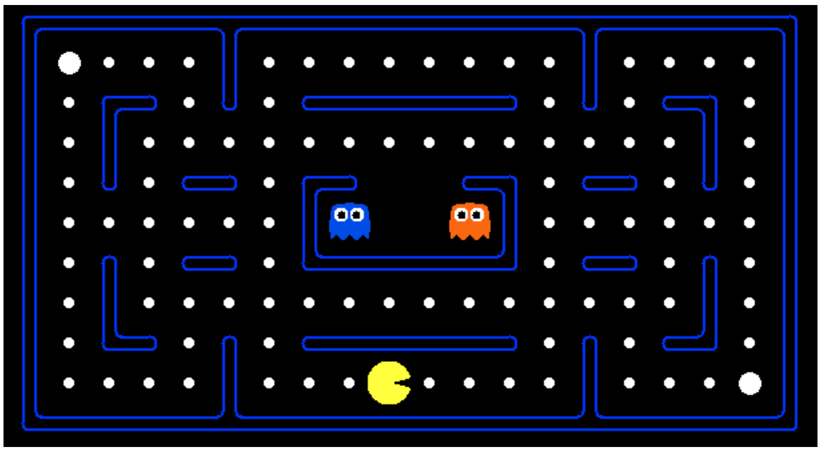


Figure 2.1: Medium map

2.3 Algorithm Implementation

The DQN learning algorithm is implemented as a modified version of the *deepQN_tensorflow* [12] program. This program was originally built to learn to play ATARI games using solely the screen pixel values and rewards. It uses in Python 2.7, Tensorflow and OpenCV in combination with the Arcade Learning Environment (ALE) [3]. The code dependent on OpenCV and responsible to interact with ALE was removed. Furthermore, the implementation was rewritten to function in combination with the latest versions of Python (3.5) and Tensorflow (0.8r) and the chosen Pac-man implementation.

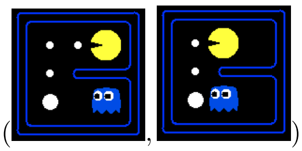
2.3.1 Experience Replay Implementation

In order to improve the learning progress experiences are randomly sampled using Experience Replay.
The replay memory in the original *DeepQN_tensorflow* implementation was reimplemented as a queue, using the native *deque* class in Python, which enables for easy sampling and straightforward freeing of memory. Every iteration the algorithm stores an experience tuple $\langle s, a, r, s' \rangle$ in this *deque*. In order to prevent that the specified maximum of used replay memory is exceeded, experience tuples are popped from this queue when more than *mem_size* experiences are added.

2.4 State Representation

The goal of modeling is to insert as little as possible 'oracle' knowledge. This means that the algorithm should learn the environment on its own without explicitly being told how the environment works. Therefore, the model and its input (game states) does not contain explicit information such as coordinates and information especially useful for pathfinding. The observations of game states presented to the learning algorithm should ideally match observations made by a human when playing the game. This study does not use raw pixel values, as previously done, but defines states as a tensor representation of the game grid. By doing this the dimensionality of the input data decreases significantly, but very little 'oracle' knowledge is told to the algorithm.
As stated earlier (*Section 1.2*) the Markov property in MDPs defines that transitions of state only depend on the present state and that prior history is not relevant. Applied to the Pac-man game this means that no memory is needed to store all past game frames. However, all ghosts in the Pac-man game move in a certain direction, but lack any acceleration. Therefore, the state is defined as the two last game frames as shown in *Figure 2.2*. This way a state includes both positional and directional information.
Each game frame consists of a grid or matrix containing all 6 key-elements: Pac-man, walls, dot,

capsules, ghosts and scared-ghosts.
It is helpful to add a new dimension to the state in which the locations of elements are represented by independent matrices. In each matrix a 0 or 1 respectively express the existence or absence of the element on its corresponding matrix.
As a consequence, each frame contains the locations of all game-elements represented in a $W \times H \times 6$ tensor, where W and H are the respective width and height of the game grid. Conclusively a state is represented by a tuple of two of these tensors together representing the last two frames, resulting in an input dimension of $W \times H \times 6 \times 2$.



$$(\quad , \quad) \rightarrow$$

$$((\underset{walls}{\begin{pmatrix} 1&1&1&1&1 \\ 1&0&0&0&1 \\ 1&0&1&1&1 \\ 1&0&0&0&1 \\ 1&1&1&1&1 \end{pmatrix}}, \underset{Pac-man}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{dot}{\begin{pmatrix} 0&0&0&0&0 \\ 0&1&1&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{capsules}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{ghosts}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{scared\,ghosts}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}),$$

$$(\underset{walls}{\begin{pmatrix} 1&1&1&1&1 \\ 1&0&0&0&1 \\ 1&0&1&1&1 \\ 1&0&0&0&1 \\ 1&1&1&1&1 \end{pmatrix}}, \underset{Pac-man}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{dot}{\begin{pmatrix} 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{capsules}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{ghosts}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{scared\,ghosts}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}))$$

Figure 2.2: State representation

2.5 Rewards

The obtained reward by the RL agent corresponds with the difference in game score. In this set-up Pac-man is rewarded with a small positive reward for eating dots and a high positive reward when eating *scared* ghosts or winning. A loss (eaten by ghosts) results in a high negative reward. A small negative reward is given over time, in order to advocate quick solutions.

2.6 Network architecture

The Q-network and the target network consist of two convolutional layers followed by two fully connected layers. The convolutional layers are valuable because the game states in the Pac-man game imply the existence of local-connectivity. As a result, CNNs can be extremely useful to improve the classification of features in the game grid.
The first layers use the Rectified Linear Unit (ReLu) non-linearity as activation function. The ReLu is defined as $f(x) = \max(0, x)$. Convolutional networks with ReLu activation functions (*Figure 2.3*) have proven to train several times faster than networks with other activation functions such as *tanh* [11]. The final layer is fully-connected and uses a linear activation function.

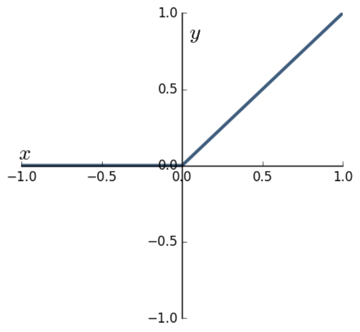


Figure 2.3: Rectifier Linear Unit (ReLu)

Pooling layers are not required due to the fact that the resolution of the input is not excessively large. Moreover, pooling layers would lead to a loss of locational information which presumably is essential for the directional action-selection in Pac-man.
Finally, RMSprop is used as an optimization of gradient descent. In RMSprop gradients are divided by the mean squared error of the widths , in order to train more efficiently. [25].

CHAPTER 3

Experiments and Results

3.1 Parameters

The applied parameters can be found in *Table 3.1.* In this section the chosen values for the more interesting (hyper-)parameters are described. Due to limitations, such as time constraints, not all parameter values could be chosen based on experiments. Furthermore, no experiments were carried out on parameters that were expected to have no or very little impact on the performance of the algorithm.

Replay memory size	10000
RMS decay	0.99
RMS episodes	10^{-6}
Batch size	32
Discount rate	0.95
Learning rate	0.0002
Training start	10000
ϵ start	1.0
ϵ steps	100000
ϵ final	0.1
Input dimensions	$W \times H \times 6 \times 2$

Figure 3.1: Used parameters

3.1.1 Reward parameters

The exact reward values are reflected in *Figure 3.2*.

Eat ghost	50
Eat dot	10
Loss (Collision with a ghost)	-100
Time punishment (Every time-step)	-1
Won (All dots eaten)	100

Figure 3.2: Rewards

3.1.2 Training parameters

The convolutional layers use small filter sizes (3×3) without any stride (stride $= 1$) because the resolution in the state representation is relatively small compared to, for example, images. Bigger filter sizes and larger strides are typically used in image classification tasks using deeper convolutional networks to classify relatively high-dimensional input. A few experiments with larger filter sizes and deeper networks did not immediately show improvements in the learning process.
The discount parameter γ was set to 0.95. Experiments were carried out with discount rates of 0.90 and 0.99. Some of these experiments showed no improvements and in some cases the altered discount rate significantly delayed the learning process.

In all experiments a learning rate of 0.0002, a RMSprop decay_rate of 0.99 and an RMSprop epsilon value of 0.99 were used.

The underlying table (*Table 3.1*) gives an overview of the used network architecture.

Layer	**Function**	**Input**	**Output**	**Filters**	**Filter size**	**Stride**
Convolutional	ReLu	$W \times H \times 6$	$W-2 \times H-2 \times 16$	16	3×3	1
Convolutional	ReLu	$W-2 \times H-2 \times 16$	$W-4 \times H-4 \times 32$	32	3×3	1
Fully-connected	ReLu	$W-4 \times W-4 \times 32$	256	256		
Fully-connected	Linear	256	4	4		

Table 3.1: Network Architecture

3.1.3 Exploration/Exploitation Parameters

In this study the ϵ-greedy strategy was applied. At the start of the learning process the ϵ value was set at one to explore all the time. During the game the ϵ value descents linearly to enable exploitation of the agent's improving policy. After a predefined amount of iterations ϵ settles at a rate of 0.1.
While testing a trained model the ϵ-value was set to zero to always exploit and fully apply a learned policy.

3.1.4 Experience Replay Parameters

As described before (*Section 1.6*), random mini-batches of experience tuples $\langle s, a, r, s' \rangle$ are sampled from replay memory. The replay memory has a maximum memory of 10000 experience tuples to limit the memory usage. To ensure this replay memory is filled before training, the first Q-function update occurs after the first 10000 iterations. Every training iteration a mini-batch, consisting of 32 experience tuples, is sampled. Some experiments were carried out to evaluate the effectiveness of using a larger mini-batch sizes (64 and 128 experience tuples). These experiments showed no improvements on the learning progress.

3.2 Evaluation of Method

The described method applied to Pac-man shows impressive results. On a medium map (10×20 game grid) a trained model was capable of winning 95 % of the games. A video of a successfully played game by the model can be found online together with source code of the algorithm [17].
All experiments were performed on a server with an *Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz* CPU, a *Nvidia GeForce GTX TITAN X* GPU and 65 GB of internal memory.
Figure 3.3 shows the percentage of won games against a few hours of training iterations. The agent starts winning after approximately 2000 episodes of training. Continueing to train the model after 2000 episodes slowly increases the win rate.

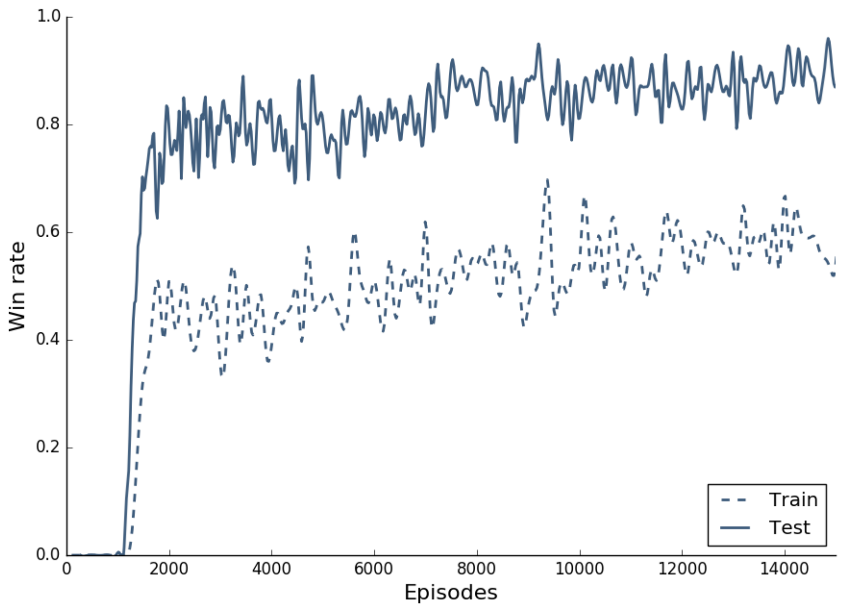


Figure 3.3: Win rate plotted against training episodes

3.2.1 Evaluation of Episode

The graph below (*Figure 3.4*) shows the maximum Q-values out of the possible actions are plotted against time steps in an episode. The average of these maximum Q-values are taken over 1000 games. The graph shows Q-values in episodes start high and decline throughout the game. It can be concluded that the algorithm learns a sensible Q-function as the number of possible rewards decline throughout a game. The Q-function, therefore, matches the expected future reward.

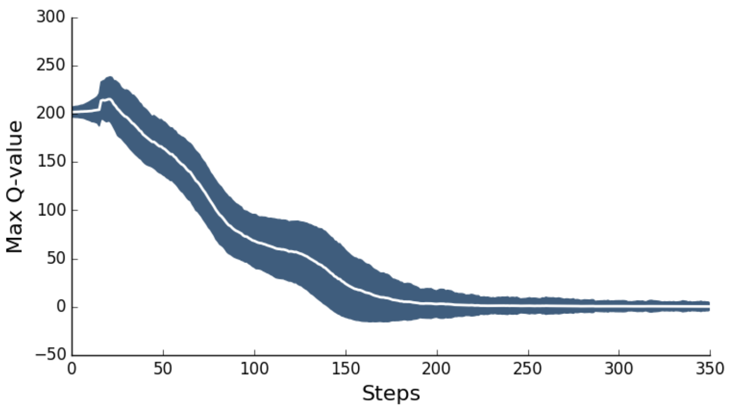


Figure 3.4: Maximum Q-values in an episode averaged over 1000 games

Another particularly interesting statistic is the peaks that arise in the Q-values in an episode after a capsule is eaten visible in *Figure 3.5*. This rise of Q-values is followed by a sudden drop in Q-value every time a ghost was eaten or after the possibility of eating (scared) ghosts is over. This is expected behaviour, as the capsule makes it possible to gain extra reward by eating a ghost. This possibility increases the expected future reward and thereby the Q-value.

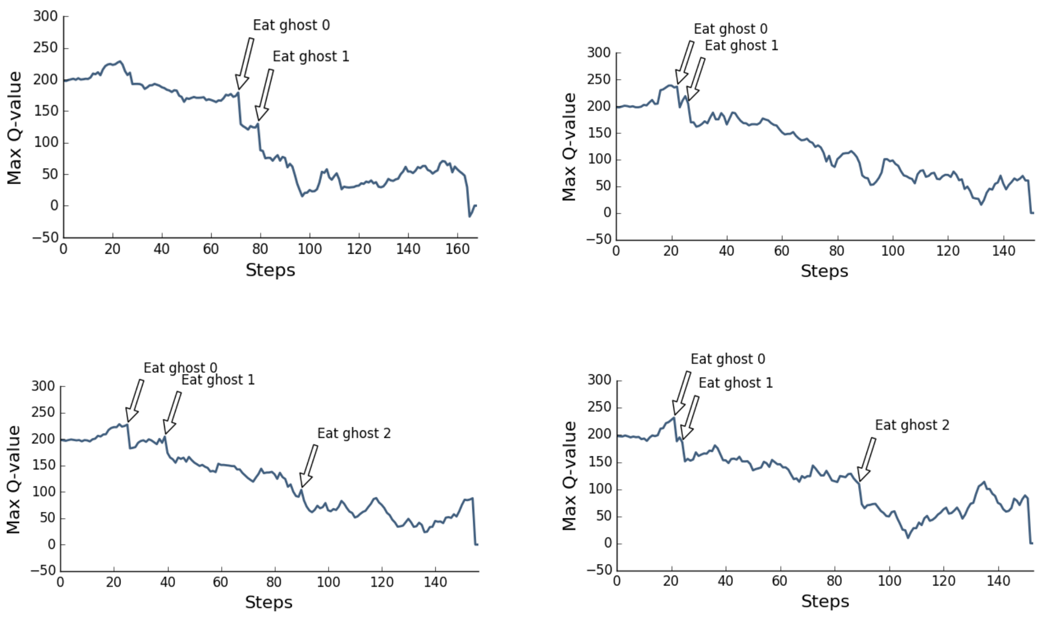


Figure 3.5: Maximum Q values in four different episodes. Annotated at the moments ghosts were eaten

The decrease in Q-value goes along with the total of earned rewards within an episode. The total of earned rewards in one episode illustrated in the figure below (*Figure 3.6*).

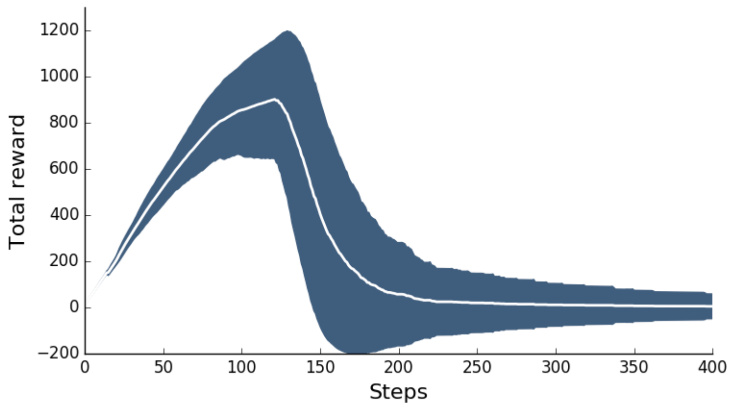


Figure 3.6: Total reward in an episode averaged over 1000 games

3.2.2 Unseen Environments

Experiments were carried out to evaluate the performance of the model when placed in situations it did not encounter during training. It was not expected that the model would generalise well over certain static aspects of the game, such as the number of ghosts and different levels.

More Ghosts

Adding more ghosts makes the problem harder, because a higher number of ghosts increases the number of actions that can possibly result in a loss. However, it would be interesting to investigate to what extent a trained model can generalize over this kind of situations.
In the underlying graph (*Figure 3.7*) the percentage of won episodes by a model is plotted against

the number of ghosts in the map. The model was trained on a medium map (*Section 2.2.1*) with two ghosts. The model performs notably less when presented with a higher number of ghosts. This is expected behaviour as the difficulty of the problem increases when more ghosts are added. Despite these lower win rates, the graph shows that it can generalise over situations with more ghosts by still winning a significant amount of games.

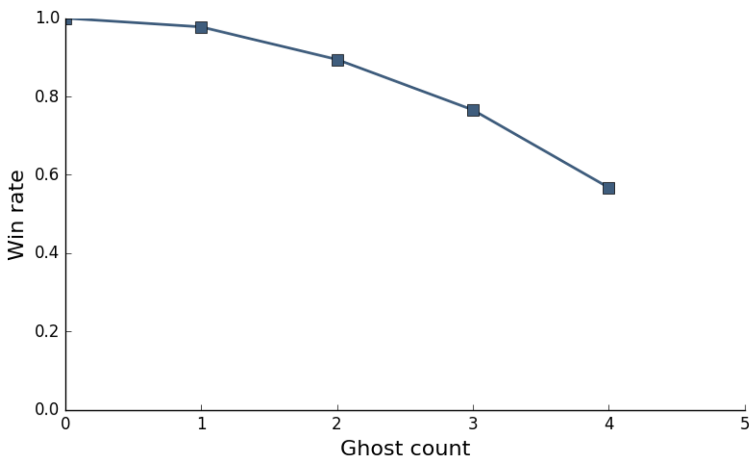


Figure 3.7: Win-rate plotted against the amount of ghosts in model trained on two ghosts

Different Levels

A model trained on one map can not generalise over maps it has never seen. When testing a model on new maps the actions performed by the model are close to random. As a result the Pac-man fails to obtain rewards and thereby fails to win. In further research it would be interesting to investigate if a model can generalise over multiple levels by training on a subset of these levels.

3.3 End-game Problem

After thousands of iterations the AI performs reasonably well on small- and medium-sized maps. However, there is a notable difference between the behaviour of a trained model at the beginning of the game and the behaviour in the end-game. The model especially performs better at the beginning of a game. Strikingly, the lack in performance in the end-game also applies to tasks occurring both at the beginning of the game as in the end-game, such as avoiding collision with ghosts.
To address the problem of the decline in performance throughout a game, three hypotheses have been tested to understand this (depressed) behaviour. Important to note is that the main characteristic of an end-game scenario is that the amount of dots in the map is sparse. Firstly the problem could be a result of the reward function. Secondly, the sparsity in the matrix could be an issue. And/or lastly, the relatively small amount of observed end-game states could have a negative impact on end-game performance.

3.3.1 End-game Problem: Rewards

The model may predict that moving into a ghost and lose (suicide) results in a higher cumulative reward than finding the last dot in the map, due to the punishment-over-time. If this is the case, a possible solution might be to change the reward function, such as variations to the amount of punishment-over-time or increasing the reward for winning a game.
In order to test this hypothesis a trained model was placed in multiple end-game scenarios (sparse

dot layer) with different distances between the Pac-man and a ghost. The Pac-man performed notably less (movement seems more random) in the end-game situations (sparse dot layer) as compared to situations in the beginning of the game. However, there was no significant evidence of 'deliberately' moving into a ghost. The few cases in which the Pac-man did move into a ghost are no significant evidence that the model preferences a early loss over the long-term reward of the last dots. These movements and are most likely the result of random movement.

3.3.2 End-game Problem: Too Many Zeros

Another explanation for the decreased performance in the end-game may be the sparsity in the matrix resembling the locations of dots. The increasing number of zero values might eliminate variations in the neural network necessary for good decision making. To investigate this hypothesis all zero values in the state representation were substituted by -1.
After this change no increase in performance was observed making the explanation less likely.

3.3.3 End-game Problem: Lack of End-game Observations

The lack of observed end-game scenarios may have made it harder to make decisions in the network. This theory could also apply to tasks which were performed well at the beginning of the game and bad at the end of the game because the environment has changed in the end-game. For example, the rate in which the model succeeds in avoiding a ghost may decline throughout an episode because the model has seen more ghosts in the context of dots than it has seen outside this context.
To test if this is the case, the training time was increased significantly. More training showed improvement in end-game scenarios. This result is in line with the stated hypothesis as more training ensures more end-game scenarios are observed by the algorithm.
A possible solutions may be longer training to incorporate more end-game scenarios in the model or using techniques as Prioritized Reinforcement Learning [19]. In Prioritized Reinforcement Learning the impact of experience tuples, when used for training, is dependent on the error in temporal difference. This means that, in theory, experience tuples in less commonly seen states such as end-game scenarios have more impact on Q-value updates using this technique.
In addition, changes in the state representation may solve the decline in performance of certain tasks throughout a game, such as avoiding ghosts. A suggested change would be to ensure that there is no overlap between certain elements in the state representation. By preventing overlap in the locational information of ghosts and dots, the extent to which the model is able to avoid ghosts will presumably be more independent on the sparsity of the dot layer.

CHAPTER 4

Conclusion / Discussion

This study evaluates the effectiveness of Deep Q-learning applied to Pac-man.
The paper includes background theory of Reinforcement Learning techniques and describes how these techniques were combined to create a model capable of learning to play Pac-man. Experiments were carried out to evaluate the strengths and weaknesses of the used algorithm.
The model wins 95% of played games on a medium map (*Section 2.2.1*). Therefore, it could be concluded that Deep Q-learning is able to successfully play the game of Pac-man.
A further study could assess the performance of the model on larger maps, including the map size of the original game. An additional weakness is the fact that, for now, the states are represented in a general but hand-made manner. Ultimately the observations carried out by the agent could be, the even more general, pixel values of the game screen.
Considerably more work will need to be done to solve the (yet unsolved) problem of playing Pac-man on the original map size and using pixel values of the game screen.
The generalisability of these results is subject to certain limitations. The method fails to generalise over scenarios that it did not encounter while training. It would be interesting to investigate to which extent a model trained on multiple maps can generalise over new, unseen, maps.
Notwithstanding these limitations, the present study provides additional evidence with respect to the general effectiveness of Deep Q-learning in dynamic stochastic systems. The methods used may be applied to a wide range of other systems and problems.

Bibliography

[1] S. Adam, L. Buşoniu, R. Babuška, *Systems Man and Cybernetics Part C: Applications and Reviews IEEE Transactions on* **2012**, *42*, 201–212.

[2] **2016**.

[3] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, *Journal of Artificial Intelligence Research* **2012**.

[4] R. Bellman, *Indiana Univ. Math. J.* **1957**, *6*, 679–684.

[5] R. E. Bellman, S. E. Dreyfus, *Applied dynamic programming*, Princeton university press, **2015**.

[6] Berkeley Pacman project, `http://ai.berkeley.edu/project_overview.html`.

[7] T. J. Davidson, F. Kloosterman, M. A. Wilson, *Neuron* **2009**, *63*, 497–507.

[8] J. DeNero, D. Klein in Proceedings of the Symposium on Educational Advances in Artificial Intelligence, **2010**.

[9] Google, Tensorflow, `https://www.tensorflow.org/versions/r0.8/get_started/index.html`, **2016**.

[10] D. H. Hubel, T. N. Wiesel, *The Journal of physiology* **1968**, *195*, 215–243.

[11] A. Krizhevsky, I. Sutskever, G. E. Hinton in Advances in neural information processing systems, **2012**, pp. 1097–1105.

[12] T. Kulkarni, Deep Q Learning in Tensorflow for ATARI, `https://github.com/mrkulk/deepQN_tensorflow`, **2012**.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., *Nature* **2015**, *518*, 529–533.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, *arXiv preprint arXiv:1312.5602* **2013**.

[15] N. Montfort, I. Bogost, *Racing the beam: The Atari video computer system*, Mit Press, **2009**.

[16] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, E. Liang in *Experimental Robotics IX*, Springer, **2006**, pp. 363–372.

[17] T. v. Ouderaa, Pacman Deep Qlearning, `https://github.com/tychovdo/PacmanDQN`, **2016**.

[18] X. B. Peng, G. Berseth, Michiel van de Panne, *ACM Transactions on Graphics (Proc. SIGGRAPH 2016)* **2016**, *35*.

[19] T. Schaul, J. Quan, I. Antonoglou, D. Silver, *arXiv preprint arXiv:1511.05952* **2015**.

[20] D. Silver, RL Course by David Silver, `https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PL5X3mDkKaJrL42i_jhE4N-p6E2Ol62Ofa`.

[21] D. Silver, A. Huang, C. J. Maddison, A. Guez, G. Sifre, Laurent and van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., *Nature* **2016**, *529*, 484–489.

[22] R. S. Sutton, A. G. Barto, *Reinforcement learning: An introduction*, MIT press, **1998**.

[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, **2015**, pp. 1–9.

[24] E. L. Thorndike, *The Psychological Review: Monograph Supplements* **1898**, *2*, i.

[25] T. Tieleman, G. Hinton, *COURSERA: Neural Networks for Machine Learning* **2012**, *4*, 2.

[26] C. J. Watkins, P. Dayan, *Machine learning* **1992**, *8*, 279–292.

[27] C. J. C. H. Watkins, PhD thesis, University of Cambridge England, **1989**.