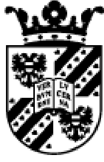

Learning to Play Pac-Xon Using Different Kinds of Q-Learning

Bachelor's Project Thesis

Jits Schilperoort, s2788659, j.j.schilperoort@student.rug.nl,
Ivar Mak, s2506580, s.i.mak@student.rug.nl,
Supervisor: dr. M.A. Wiering

Abstract: When reinforcement learning (RL) is applied in games, it is usually implemented with Q-learning. However, it has been shown that Q-learning has its flaws. A simple addition to Q-learning exists in the form of double Q-learning, which has shown promising results. In this study, it is investigated whether the advantage double Q-learning has shown in other studies also holds when combined with a multilayer perceptron (MLP) that uses a feature representation of the game state (higher order inputs). Furthermore we have set up an alternative reward function which is compared to a conventional reward function, to see whether presenting higher rewards towards the end of a level increases the performance of the algorithms. For the experiments, a game called Pac-Xon is used. Pac-Xon is an arcade video game in which the player tries to fill a level space by conquering blocks while being threatened by enemies. We found that both variants of the Q-learning algorithms can be used to successfully learn to play Pac-Xon. Furthermore double Q-learning obtains higher performances than Q-learning and the progressive reward function does not yield significantly better results than the regular reward function.

Keywords: Reinforcement Learning, Q-Learning, Double Q-Learning, Multi-layer Perceptron, Pac-Xon

1 Introduction

"Pure logical thinking cannot yield us any knowledge of the empirical world; all knowledge of reality starts from experience and ends in it."

- Albert Einstein.

When humans are born, they have very little knowledge about the world. As they grow older, they have endeavoured many different actions that have led to many different situations. Whenever a new situation occurs, a good action can be approximated based on experiences in similar previous situations. This can be seen as learning by trial and error. This way of learning is the foundation of reinforcement learning (Wiering and Van Otterlo, 2012), an artificial computational approach of learning by trial and error. Games can provide suitable environments to model an agent that is trained using reinforcement learning (Laird and VanLent, 2001). In this approach, the agent plays the game in order to learn to distinguish desirable from undesirable actions.

1.1 Previous Work

Because of their nature, games have provided environments for reinforcement learning numerous times. This has been done in both board games like backgammon (Tesauro, 1995) and more graphical intensive video games such as Ms. Pac-Man (Bom et al., 2013), regular Pac-Man (Gallagher and Ledwich, 2007) and multiple Atari games (Mnih et al., 2013).

In the previously named video games the widely used variant of reinforcement learning, Q-learning (Watkins and Dayan, 1992) was applied. In the classical way, Q-learning makes use of a table to store its state-action pairs. Watkins and Dayan have proven that this method of reinforcement learning converges to optimal action-values under

the condition that all actions are repeatedly sampled in all states. However, when Q-learning is applied in video games with a huge state space, this method becomes unfeasible since it would take too long to repeatedly sample over all individual state-action pairs. This is why the previously named games were trained using an artificial neural network to approximate Q-values for state-action pairs rather than a lookup table that stores these Q-values. In the Ms. Pac-Man and Pac-Man papers a multilayer perceptron (MLP) is used, while the Atari games were trained using Deep Q-Networks (DQN). With these methods, less states are required to be presented to the learning agent, and less data is needed to be stored in order to obtain good playing behavior. The experiments in (Bom et al., 2013) and (Gallagher and Ledwich, 2007) have shown promising results, often leading to playing behavior as good as that of average human players. In the case of the Atari games, in three of the seven games included in the experiments it even surpasses the playing behavior of human expert players (Mnih et al., 2013).

The learning agents in the Atari games were trained using raw pixel input as a state representation that is presented to the agent. The Ms. Pac-Man agent was trained using higher order inputs, requiring only 22 input values as a representation of its game state. The regular Pac-Man agent uses several grids as input which were converted from raw pixel data. Using the raw pixel input has as an advantage that a better distinction between states can be made since every individual game state presented to the algorithm is unique. When using higher order input variables, less unique states can be represented, so it can happen that different states appear similar to each other to the algorithm. The drawback of using raw pixel input data is that it requires enormous amounts of computational power while an average personal computer should be able to train an algorithm that makes use of higher order inputs.

Even though Q-learning has shown great successes, it does have its flaws. Because of its optimistic nature, it sometimes overestimates action-values. As an alternative to Q-learning van Hasselt proposed a variant to Q-learning called double Q-learning (van Hasselt, 2010). Because of the decrease of the optimistic bias in double Q-learning, this variant has shown improvements in many Atari games (van Hasselt et al., 2016).

1.2 Contributions

In this research the video game Pac-Xon is used as an environment. This game has a large game state space and many different levels. Each new level that is encountered by the agent is a little more difficult than the previous one, giving the game high scalability. This provides a situation in which the agent has to constantly adapt to new situations. Whenever the agent has obtained a policy with which it is capable of completing a certain level, it encounters a new, more complex level. The full explanation of the game can be found in Section 2.

We present a research in which we use techniques used in the previously named researches on Ms. Pac-Man and Atari games. It is investigated whether the advantage of double Q-learning, as observed in the Atari games, also holds in combination with higher order inputs and an MLP as applied in the Ms. Pac-Man paper. The double Q-learning algorithm has not been implemented before with the use of higher order inputs (a game state feature space), which makes it interesting to see whether its performance differs from regular Q-learning.

Another aspect that makes the performed experiments an interesting new challenge, is the fact that the nature of this particular game requires the agent to take more risk as it progresses through a level. At the start of a level it is quite easy to obtain points, but as the agent progresses, the risk that has to be taken to obtain the same amount of points becomes increasingly higher. To deal with this, two different reward functions are tested. One of them gives a reward proportional to the obtained score, the other one also takes level progress into account, giving a higher reward to points obtained in a later stage of the level.

Outline

Section 2 describes the framework we constructed to simulate the game and the extraction of the game state features. Section 3 discusses the theory behind the reinforcement learning algorithms combined with an MLP. Section 4 describes the experiments that were performed and the parameters we used. Section 5 describes and discusses the results

that were acquired, and in Section 6 we present our conclusions and give suggestions for future work.

2 Pac-Xon

2.1 Game Setup

Pac-Xon is a computer game, of which the name is a contraction of Pac-Man and Xonix. The game itself implements parts of the gameplay of Xonix, which is derived from Qix (released in 1981 by Taito Corp.), combined with some gameplay and graphics of Pac-Man.

The game starts in an empty rectangle with drawn edges. The level can be viewed as a grid space with 34 × 23 tiles, with a player initiated on the edge and a number of enemies initiated in the level space. In Figure 2.1, a screen shot of the game is depicted, in which the player is moving to the right while dragging a tail. The player can move in four directions (north, east, south, west), and can stand still, as long as is it not moving over empty space.

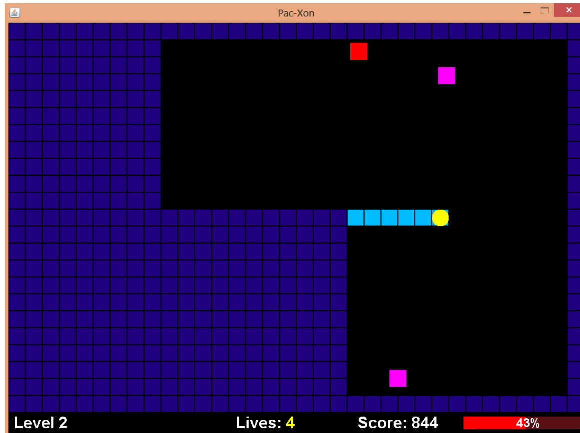

Figure 2.1: Screenshot of the game, in which the following objects can be viewed: player (yellow), tail (cyan), normal enemies (pink), eater enemy (red), solid tiles and edges (dark blue).

2.2 Gameplay

The main goal for the player is to claim empty space, while avoiding the enemies. This is done by dragging temporary tiles, from now on addressed with *tail*, from edge to edge, which become solid tiles after reaching a solid tile. In Figure 2.1 the tail is depicted in cyan. When enclosing an empty space (i.e. there are no enemies apparent), the space and the tail will convert to solid tiles, scoring a point for each tile. If the tail did not enclose empty space, solely the tail will convert to solid tiles. After claiming 80 percent of the total area, the player completes the level and advances to the next.

There are two options for failing the level, which are related to the enemies moving around in the level space. When the player collides with an enemy, the player dies. Each time the player dies, the game ends in a loss and restarts. This as opposed to the original game, in which the player would lose a life and restart on the edge as long as it has lives left. We chose this implementation to simplify the game. In order to achieve better training results using our MLP, we eliminate the extra variable for lives which would influence decision making.

The second option for a fail is when the enemy collides with the tail of the player. The tail will break down with a speed of $1\frac{1}{2}$ that of the player's speed. This means the player will have a certain amount of time to reach a solid tile in order to complete its tail and survive. If the player does not reach a solid tile in due time (i.e. gets caught up by the breaking tail), the player dies.

There is a third way in which the player can fail the level, which is not related to the enemies. When the player runs into its tail, the player dies.

2.3 Enemy Description

We have implemented three different types of enemies, which we named according to their specific behavior:

- **Normal**: Depicted in pink, moving around in the empty space at a constant speed, bouncing away from solid tiles and the player's tail when they hit them.

- **Eater**: Depicted in red, moving around in the empty space at a constant speed which is half that of the normal enemies. These enemies will clear away the solid tile they hit before bouncing away, the solid edges of the level and the player's tail excluded.

- **Creeper**: Depicted in green, moving around on the solid tiles of the level at a constant speed the same as the normal enemy, bouncing away from the edges of the frame and the empty space. This enemy is initiated after the player fills its first block of solid tiles, to avoid it getting stuck on an edge.

2.4 Enemy Distribution

The amount of enemies is dependent on the game progress. Starting off with two enemies in level one, after passing, the number of enemies increases by one for each subsequent level. The enemies are distributed differently over the levels according to the following set of rules:

- If the level number is even: add $\frac{levelnumber}{2}$ of eater enemies.
- If the level number can be divided by three (and returns a remainder of zero): add a creeper enemy, to a maximum of one.
- Add the remaining number of normal enemies.

This means there will always be one enemy more than the number of the current level. Each level will contain at least two normal enemies, and there will never be more than one creeper enemy.

2.5 State Representation

We want to supply the MLP with information about the game, since the agent needs to link its feedback in the form of future rewards to a situation in the game. We have constructed a representation of the environment which is called the game state, that contains the information that can be viewed on the screen. We calculate a total of 42 variables, which are stored in a vector. In order to use them as input for the MLP, we normalize these values between zero and one. There are a number of features that produce multiple inputs (for example one for every direction). This means we have a total of 17 features, which are described in Table 2.1. We have divided the features into three categories: Tile Vision, Danger and Other. Furthermore, Figure 2.2 depicts some examples of the game state features and how they relate to the on-screen information.

Note that in Figure 2.1 the player is about to enclose an enemy. When this happens, the numberOfFields variable (explained in Table 2.1) will increase.

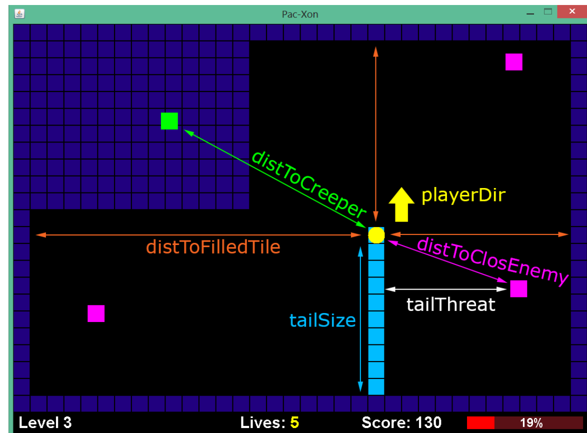

Figure 2.2: State representation example, with the state variables depicted in arrows with their corresponding labels. The following objects can be viewed: player (yellow), tail (cyan), normal enemies (pink), creeper enemy (green), solid tiles and edges (dark blue).

Table 2.1: State representation features

Number	Feature	Description
Tile Vision		
1-4	unFilledTiles	In four directions, the normalized inverted distance towards the closest unfilled tile. If there is none, this value is set to 0.
5-8	tailDir	In four directions, a binary value determined by whether there is an active tail within 4 blocks of the player. Set to 1 if there is, set to 0 if not.
9-12	distToFilledTile	In four directions, the normalized inverted distance towards the closest filled (solid) tile.
13	safeTile	A binary value determined by the current position of the player. If it is on a filled (safe) tile, this value is set to 1. If not, it is set to 0.
14-17	unfilledTiles on row/column	In four directions, the normalized inverted distance towards the closest row or column with unfilled tiles.
Danger		
18	enemyDir	Determined by the direction of the closest enemy. This value is either 0= not moving towards player, 0.5=moving towards player in 1 direction (x or y) or 1=moving towards player in two directions (x and y).
19-22	enemyDist	In four directions, the normalized inverted distance towards the closest enemy in that direction. Set to zero if there is none.
23	distToClosEnemy	The euclidean distance towards the closest enemy, inverted, and normalized through a division by 20.
24	distToCreeper	Euclidean distance towards the creeper (green enemy), inverted, and normalized through a division by 20.
25-28	enemyLoc	In four directions, the player has vision in that direction with a width of 7 blocks. If there is an enemy apparent, the inverted distance gets normalized such that one enemy can amount up to 0.5. This means the player can detect up to two enemies in each direction, and adjust its threat level accordingly.
29	tailThreat	The inverted euclidean distance from the active tail towards the closest enemy, normalized through a division by 20. This value is set to zero if there is no active tail.
30	tailHasBeenHit	Binary value determined by whether the active tail has been hit by an enemy. If there is no active tail, or it has not been hit, this value is set to zero.
31-34	creeperLocation	Inverted normalized distance towards creeper (green enemy) in four directions. If there is no creeper in that direction, the value is set to zero.
Other		
35-39	playerDir	Binary value determined by the direction in which the player is moving, with no direction (standing still) as a fifth option.
40	numberOfFields (/enemies)	Based on the ratio between between empty spaces and the number of enemies. (0 when only one field, 1 when all enemies separated). In Figure 2.1 this variable will have value 0, but will become 0.5 when the player successfully encloses the enemy.
41	percentage	The normalized percentage of the level passed (filled with tiles).
42	tailSize	Normalized length of the tail, set to 1 if is larger than 40 blocks. Set to zero if there is none.

3 Reinforcement Learning

The process of making decisions based on observations can be seen as a Markov Decision Process (MDP) (Bellman, 1957). To deal with the dynamic environment of the game, reinforcement learning was applied. Such a system can be divided into five main elements: the learning agent, the environment, a policy, a reward function, and a value function (Sutton and Barto, 1998).

The policy can be seen as the global behavior of the learning agent. It represents the function of, at time t, deciding on an action (a_t) based on a certain state (s_t).

When the process starts, the learning agent runs through the training stage. Initially the learning agent has no information of its environment and therefore its policy will be random. As time passes the agent will learn about its environment and obtain a policy that is based on rewards it has encountered in previous situations.

3.1 Q-Learning

The Q-learning algorithm (Watkins and Dayan, 1992) is used to tackle this MDP. This is a form of temporal difference learning, which implies that we try to predict a quantity that depends on future values of a given signal. The quantity in this case is a measure of the total amount of discounted rewards expected over the future. Equation 3.1 states the Q-learning update rule in its general form.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{3.1}$$

There are two constants that affect the equation. The constant α represents the learning rate, which influences the extent to which the Q-values are altered following an action, this is a value between 0 and 1. The constant γ represents the discount factor, which is also a value initiated between 0 and 1. This value determines how much influence the future rewards have on the updates of the Q-value. The future rewards get discounted according to this value. This means that a discount factor close to 0 means the agent focuses on rewards in the near future. Using a discount factor closer to 1, rewards in the more distant future will have a larger influence. s_{t+1} represents the new state after having taken action a_t in state s_t. r_t represents the observed reward at time t.

The algorithm uses state-action pairs to store and use information, these consist of a state representation explained in the previous section and a game action (i.e. move) that can be chosen. Each state-action pair is associated with a Q-value that expresses the quality of the decision. The algorithm calculates these values, based on the reward that is received for choosing that action plus the maximal expected future reward.

This equation is applied for updating Q-values stored in a table. Our state-action space is too large to store all Q-values in a table, so we implemented a different approach by combining Q-learning with a function approximator, which is explained in detail in Section 3.3.

3.2 Reward Function

The reward function is used to represent the desirability of choosing an action (i.e. transitioning to some state following that particular move). This is done by using values that refer to specific events in the game. This value could either be positive, or negative, the latter of which can be seen as a penalty for an action that is undesirable. The rewards that are yielded after a state transition are a short term feedback, but due to the fact that the algorithm includes future rewards in its action evaluation, it could be the case that an action is linked to a high Q-value while returning a low immediate reward.

We have implemented two types of reward functions, which we are going to compare in terms of performance. The first is a 'regular' (fixed) reward function, which yields rewards that are static in terms of the level progress. This function is described in Table 3.1. The reward for capturing tiles is dependent on the number of tiles that are captured, and this number is scaled to the other rewards by dividing it by 20. We have implemented a positive reward for movement over empty space (i.e. unfilled tiles) in order to encourage the agent to move there. Furthermore, a number of negative rewards have been implemented to ensure the agent does not get stuck in a local maximum, moving over tiles that do not increase the progress and score.

The second approach is a progressive reward function that takes the level progress into account in the reward that is generated for capturing tiles.

Table 3.1: Regular (static) reward function with *a* = number of tiles captured.

Event	Reward
Level Passed	100
Tiles Captured	$((a\ /\ 20) + 1)$
Movement in empty space	2
Died	-100
No movement	-2
Movement in direction opposite from previous	-2
Movement over solid tiles	-1

This level progress is expressed in the percentage of the level space that is filled with tiles. Since the player has to fill only 80 percent in order to pass the level, this is the denominator used in calculating the level progress. This alternative reward function is shown in Table 3.2.

The aim of this reward function is to incline the agent to take more risk, the further it progresses through the level, by returning a higher reward for capturing tiles. The rewards yielded for other events are equal to the regular reward function.

Table 3.2: Progressive reward function with
***a* = number of tiles captured**
***b* = percentage of the level passed**

Event	Reward
Level Passed	100
Tiles Captured	$(\sqrt{a \times b/0.8} + 1)$
Movement in empty space	2
Died	-100
No movement	-2
Movement in direction opposite from previous	-2
Movement over solid tiles	-1

3.3 Multi-Layer Perceptron

To show the relevance of using an MLP rather than a table with Q-values, an estimation of the number of game states in the first level is made. Any tile can contain the player (but only one). Furthermore there are two normal enemies that can exist on any tile that is not the border (32×21 possible tiles) and any tile that is not the border can either be empty, conquered or contain the tail of the player. This results in an estimation of $(34 \times 23) \times (32 \times 21)^2 \times 3^{(32 \times 21)} \approx 10^{329}$ possible game states in the first level.

Note that this estimation also contains some game states that are in practice not possible but it gives a general idea of the scale. Furthermore this estimation is of the first level and each level the complexity increases because there are more enemies and different kinds of enemies.

Using a table to store the Q-values of all state-action pairs is not feasible in games with a huge state space such as Pac-Xon, because there exist too many different game states to keep track of. This is why a combination of Q-learning with an MLP is applied. An MLP is an artificial neural network which acts as a function approximator. Because of this, less data is required to be stored since not all individual state-action pairs have to be stored. A pseudo code version of the algorithm used for implementing Q-learning combined with the MLP is shown in Algorithm 3.1.

Algorithm 3.1 Q-Learning algorithm. The exploration algorithm is fully explained in section 4

initialize s and Q
repeat
  if $explore()$ **then**
    $a \leftarrow randomMove()$
  else
    $a \leftarrow \text{argmax}_a\, Q(s,a)$
  end if
  $s^* \leftarrow newState(s,a)$
  $Q^{target}(s,a) \leftarrow r_t + \gamma \max_a Q(s^*,a)$
  $update(Q(s,a), Q^{target}(s,a))$
until *end*

The used MLP consists of an input layer, a hidden layer and an output layer. Between the layers a set of weights exists. A simplified version of the network is shown in Figure 3.1.

3.3.1 Input

As its input, the MLP gets all state representation variables which are explained in Section 2.5. They add up to a total of 42 values which are all normalized between 0 and 1. These values are multiplied by the input weights and sent to the hidden layer.

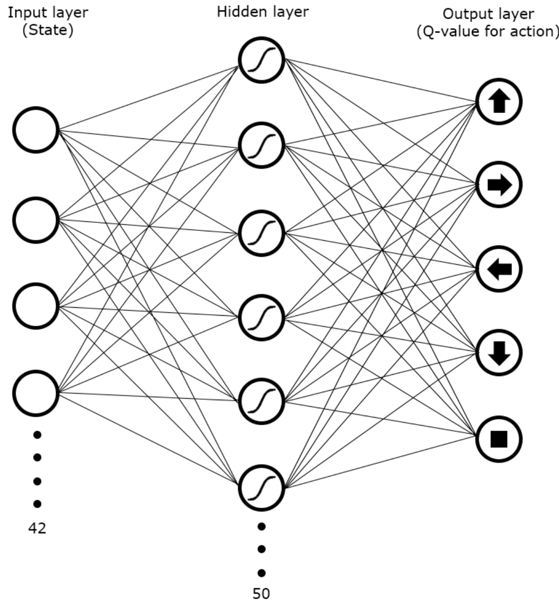

Figure 3.1: Multilayer Perceptron

3.3.2 Hidden Layer

The hidden layer consists of an arbitrary number of nodes. Each node receives the input of all input nodes multiplied by their weights. These weighted inputs are added together with a bias value and sent through an activation function, for which a sigmoid function (Equation 3.2) is used.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

This results in values in the hidden nodes between 0 and 1. It was also considered to apply a variant of an exponential linear unit (ELU) rather than a sigmoid activation function, which was shown to result in better performances in (Knegt et al., 2018) but we decided to leave that for future research.

3.3.3 Output Layer

The output layer consists of five nodes and receives its input from the hidden layer multiplied by the corresponding weights. The activation function used in this layer is linear ($f(x) = x$). Each node in this layer represents a Q-value for a specific action. Whenever an action is picked, it is either random (exploration, explained in Section 4) or the action with the highest Q-value from the network is chosen.

3.3.4 Backpropagation

The network makes use of data that is acquired dynamically while playing the game. Therefore it is considered online learning (Bottou, 1998). Updating the network is done by backpropagation (Rumelhart et al., 1988). The error required for backpropagation is the difference between the target Q-value (Equation 3.3) and the perceived Q-value. Whenever a terminal state is reached, i.e. the agent either died or passed a level, Equation 3.4 is used since there are no future values to take into account.

$$Q^{target}(s_t, a_t) \leftarrow r_t + \gamma \max_a Q(s_{t+1}, a) \tag{3.3}$$

$$Q^{target}(s_t, a_t) \leftarrow r_t \tag{3.4}$$

3.4 Double Q-Learning

When calculating target Q-values, Q-learning always uses the maximal expected future values. This can result in Q-learning overestimating its Q-values. Double Q-Learning was proposed in (van Hasselt, 2010) because of this possible overestimation. The difference with regular Q-Learning lies in the fact that two agents (MLPs) are trained rather than one. Whenever an agent is updated it uses the maximal Q-value of the other agent. This should reduce the optimistic bias since chances are small that both agents overestimate on exactly the same values. The new equation for the target Q-value is shown in Equation 3.5. Note that the equation contains both a Q_A and a Q_B. These represent the individual trained networks. A pseudo code version of the algorithm is shown in Algorithm 3.2.

$$Q_A^{target}(s_t, a_t) \leftarrow r_t + \gamma \max_a Q_B(s_{t+1}, a) \tag{3.5}$$

Algorithm 3.2 Double Q-Learning algorithm.

```
initialize s, Q_a and Q_b
repeat
  pickrandom(A, B)
  if A then
    if explore() then
      a ← randomMove()
    else
      a ← argmax_a Q_A(s, a)
    end if
    s* ← newState(s, a)
    Q_A^target(s, a) ← r_t + γ max_a Q_B(s*, a)
    update(Q_A(s, a), Q_A^target(s, a))
  else if B then
    if explore() then
      a ← randomMove()
    else
      a ← argmax_a Q_B(s, a)
    end if
    s* ← newState(s, a)
    Q_B^target(s, a) ← r_t + γ max_a Q_A(s*, a)
    update(Q_B(s, a), Q_B^target(s, a))
  end if
until end
```

4 Experiments

4.1 Training the Agent

When an agent is initialized in the environment it has absolutely no knowledge of the game. The weights of the MLP are all randomly set between -0.5 and 0.5. In total each agent is trained for 10^6 epochs. One epoch consists of the agent playing the game until it reaches a terminal state, which in the training stage means that it either dies or passes a level.

4.1.1 Exploration

Each training epoch, the agent chooses to pick an action from the network or perform a random move for exploration. The exploration method used is a decreasing ϵ-greedy approach (Groot Kormelink et al., 2018). This means that the agent has a probability of ϵ to perform a random action and a probability of $1-\epsilon$ to choose the action from the network which is expected to be the best. The value of ϵ decreases over the epochs. The algorithm for exploration is shown in Algorithm 4.1. The value of ϵ is initialized at 1, but decreases as the training progresses. The value of ϵ is determined by a function over the epochs. The formula $\epsilon(E)$, with E as the current training epoch is shown in Equation 4.1.

$$\epsilon(E) = \begin{cases} 1 - \frac{0.9*E}{50,000} & \text{if } 0 \leq \text{E} < 50,000 \\ \frac{750,000-E}{700,000*10} & \text{if } 50,000 \leq \text{E} < 750,000 \\ 0 & \text{if } 750,000 \leq \text{E} < 1,000,000 \end{cases} \tag{4.1}$$

Whenever an agent has failed to obtain any points in 100 moves, also a random move is performed. This is done in order to try to speed up the training as the agent cannot endlessly wander around the level or just stay in a corner without obtaining points.

Algorithm 4.1 ϵ-greedy exploration

```
r ⇐ randomDouble(0, 1)
if r > ε then
  move ⇐ bestExpectedMove()
else
  move ⇐ randomMove()
end if
performMove(move)
```

A total of 40 agents are trained. The agents are divided into four groups. Each group of 10 agents is trained using a unique combination of one of the reward functions and either double Q-learning or regular Q-learning. After a search through parameter space, we decided to use the hyperparameters as described in Table 4.1.

Table 4.1: Parameters used for the experiments

Parameter	Value
Discount factor	0.98
Learning Rate	0.005
Number of hidden layers	1
Nodes per hidden layer	50

4.2 Testing the Agent

The testing stage consists of 10^5 epochs. In every epoch each trained agent plays the game until it either dies or gets stuck (performs 100 actions without obtaining points). Both the total score of each

epoch as the level the agent died or got stuck in are stored.

With this data, both algorithms and reward functions will be compared to each other.

5 Results

In Figure 5.1 and 5.2 the training stage of the agents is plotted. Figure 5.1 shows the average level the agents reached with the reward function that is not related to the level progress. Figure 5.2 shows the average level that the agents that made use of the progressive reward function reached. In both graphs two lines are plotted representing Q-learning and double Q-learning. The shapes of the graphs are related to the exploration variable ϵ of the agents. The value of this variable decreases from 1 to 0.1 in the first 50,000 epochs, and then gradually decreases until it reaches 0 in epoch 750,000.

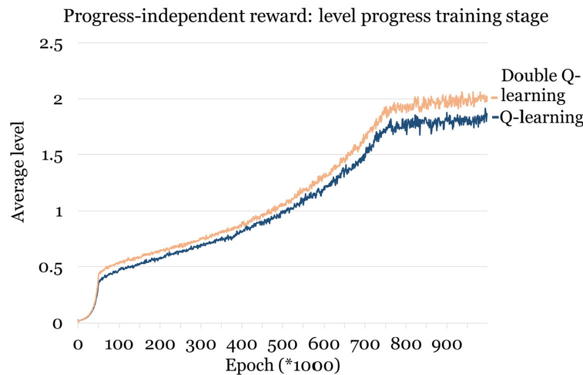

Figure 5.1: Graphs of the training stage using the regular reward function. Both lines are averaged over 10 trials.

The results of the testing phase are shown in Figure 5.3. Note that in the testing phase there is no exploration and learning, so all decisions made by the agents are chosen from the network and the network is not updated anymore.

To compare the algorithms and reward functions t-tests were performed. The values used for these tests are shown in Table 5.1 and 5.2. A more elaborate summary of the results can be found in Table 5.3. The performed t-test for the comparison between the scores of the algorithms with a difference in mean of 177 in favor of double Q-learning yielded a p-value of 0.04. This indicates that, according to

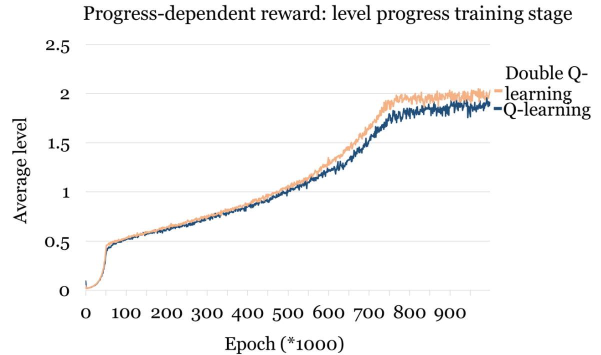

Figure 5.2: Graphs of the training stage using the progressive reward function. Both lines are averaged over 10 trials.

conventional criteria ($\alpha = 0.05$) a significant difference in means exists.

The difference in mean of the reward functions is 39 in favor of the regular reward function with a p-value of 0.66, which is higher than the conventional α, indicating that this difference in means is not significant.

It is important to note that each of the four trials consists of individually trained instances. Because of this, they have to be seen as four unique populations. Therefore a repeated measures analysis of variance (ANOVA) statistical test would not be valid with our comparison. So instead of this, additionally to the t-tests, a one-way ANOVA was performed to ascertain that the differences in performance are not related to the different variations of the algorithms. The outcomes of this test can be seen in Table 5.4 and 5.5. The smallest p-value that came out of the test is 0.28, which is greater than the conventional value of α. This means that there is no significant difference in means of the individual combinations of all four populations.

Table 5.1: Reward mean comparison

	Regular reward			Progressive reward			
	N	Mean	SD	N	Mean	SD	*p-value*
Score	20	944	261	20	905	298	0.66

Table 5.2: Algorithm mean comparison

	Q-learning			Double Q-learning			
	N	Mean	SD	N	Mean	SD	*p-value*
Score	20	836	339	20	1013	162	0.04

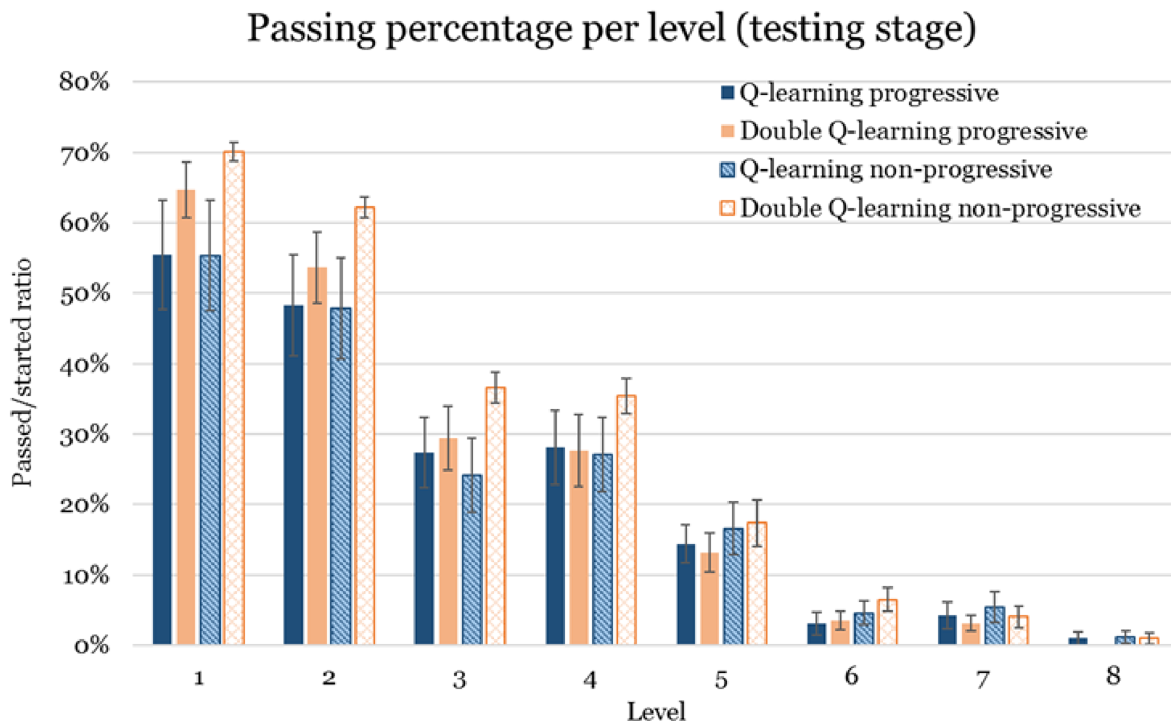

Figure 5.3: Graphs of the testing stage. Each bar shows the average passed/started ratio of 10 trials. The error bars show the standard error.

Table 5.4: ANOVA table

	Sum of squares	d.f.	Variance	F	p
Between groups	347794	3	115931	1.57	0.21
Within groups	2650201	36	73617		
Total	2997994	39			

Table 5.5: Tukey HSD Post-hoc Test

		95% conf interval		
	Diff	From	To	p
Q prog vs Q reg:	-4	-331	323	1.02
Q prog vs DQ prog:	134	-192	461	0.69
Q prog vs DQ reg:	216	-110	543	0.30
Q reg vs DQ prog:	138	-189	465	0.67
Q reg vs DQ reg:	220	-107	547	0.28
DQ prog vs DQ reg:	82	-245	409	0.91

5.1 Discussion

In this research we have examined two different temporal difference learning algorithms, Q-learning (Watkins and Dayan, 1992) and Double Q-learning (van Hasselt, 2010). These algorithms have been trained using an input vector of state variables, similar to an approach that has been used in previous research on games (Bom et al., 2013).

We compared these algorithms in terms of performance in playing the game Pac-Xon. From the results, described in Table 5.2, we can conclude that Double Q-learning reaches a significantly higher performance than Q-learning. This suggests that there is some form of overestimation in the Q-learning algorithm that is averted by training two separate MLPs.

Both algorithms were able to reach a satisfactory performance in passing the first levels, and reaching levels that have revealed to be difficult for human standards.

Furthermore, we have examined different implementations for the reward function, which is a vital part in these reinforcement learning approaches (Sutton and Barto, 1998). We made a comparison between a 'standard' reward function using fixed values, and a progressive reward function, increas-

Table 5.3: Summary of the results

						95% confidence interval			
Algorithm	Reward	N	Mean	σ	SE	Lower bound	Upper bound	Min	Max
Q	Progressive	10	838	370	117	608	1067	8	1300
Q	Regular	10	834	326	103	632	1036	250	1305
DQ	Progressive	10	972	203	64	846	1098	620	1187
DQ	Regular	10	1054	102	32	991	1117	927	1220

ing the reward pursuant to level progress. Based on the outcomes of our experiments we can conclude that Q-learning does not depend on the reward function in order to cope with the increasing complexity the game poses as the player progresses through the individual levels.

6 Conclusion and Future Work

We have found that the advantage double Q-learning showed with the DQNs in Atari games (Mnih et al., 2013) also holds when applied with higher order inputs and an MLP. This suggests that the advantage of double Q-learning is generalizable to different applications. The double Q-learning algorithm is a relatively simple addition to regular Q-learning, making it easy to implement while obtaining better results. Q-learning has been the standard in the field of reinforcement learning for quite a while, but the recent findings combined with this research make a strong case to suggest that double Q-learning should replace Q-learning in this position.

The comparison between the reward functions did not yield a significant difference, which can be seen as an indication that these reinforcement learning algorithms are already adaptive in such a way that they recognize situations that are more dangerous as closer towards the completion of a level, giving a very high reward.

This research provides suggestions for future research in a couple of ways. First off, we used quite a large number of input variables in our state representation. It might be possible to decrease the size of this, by eliminating variables that add marginally to the overall performance. Secondly, in our testing stage, both by the gathered data and visualizing the playing performance, we came across agents that have flaws. Occasionally there is a combination of factors that constrains the agent in such a way that it stays idle, and does not move anymore. This could be improved, to reach a higher performance in playing the game. Furthermore, an interesting opportunity for further research might be another variation of the reward function, in which there is no large reward for completing the level. This way, the agent will solely train on the positive reward achieved by claiming tiles. It could also be tested whether experiments using different activation functions, apart from the sigmoid function that we have been using, influence the performance of double Q-learning relative to Q-learning. Finally, more studies should be performed on the comparison between Q-learning and double Q-learning in as many different environments as possible, enabling to confidently state that double Q-learning performs better than Q-learning.

References

R. Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518.

L. Bom, R. Henken, and M. Wiering. Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 156–163, April 2013.

L. Bottou. Online algorithms and stochastic approximations. In *Online Learning and Neural Networks.* Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012.

M. Gallagher and M. Ledwich. Evolving Pac-Man players: Can we learn from raw input? In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 282–287, April 2007.

J. Groot Kormelink, M. Drugan, and M. Wiering. Exploration methods for connectionist Q-learning in Bomberman. In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence, ICAART 2018, Volume 2, Funchal, Madeira, Portugal*, pages 355–362, 2018.

S. Knegt, M. Drugan, and M. Wiering. Opponent modelling in the game of Tron using reinforcement learning. In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence, ICAART 2018, Volume 2, Funchal, Madeira, Portugal*, pages 29–40, 2018.

J. Laird and M. VanLent. Human-level AI's killer application: Interactive computer games. *AI magazine*, 22(2):15, 2001.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*, 2013.

D. Rumelhart, G. Hinton, and R. Williams. Neurocomputing: Foundations of research. pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* A Bradford book. 1998.

G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3): 58–68, 1995.

H. van Hasselt. Double Q-learning. In *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double Q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

C. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

M. Wiering and M. Van Otterlo. *Reinforcement Learning: State of the Art.* Springer, 2012.

A Appendix

A.1 Full test results

Table A.1: All members of the populations

Q-learning				Double Q-learning			
Regular reward		Progressive reward		Regular reward		Progressive reward	
Mean score	High score	Mean score	High score	Mean score	High score	Mean score	High score
1305	5166	8	3918	962	4837	713	4767
406	3222	666	4740	1161	4803	996	4757
986	4805	1122	4650	987	4691	851	**5180**
884	4802	732	4995	1064	4733	1031	4738
994	4741	581	4791	1001	4746	**1187**	4687
250	4454	999	3952	1171	**5114**	1166	4808
1109	**5739**	**1300**	4741	**1220**	4773	1154	5053
988	4760	914	4677	927	4744	853	5021
596	3631	889	4773	961	4798	1149	4783
820	4146	1163	**5052**	1085	4777	620	4774