Learning to Play Pac-Xon with Q-Learning and Two Double Q-Learning Variants

Jits Schilperoort
Artificial Intelligence
University of Groningen
Groningen, The Netherlands
jitsschilperoort@gmail.com

Ivar Mak
Artificial Intelligence
University of Groningen
Groningen, The Netherlands
ivarmak@hotmail.com

Madalina M. Drugan
ITLearns.Online
Utrecht, The Netherlands
madalina.drugan@gmail.com

Marco A. Wiering
Artificial Intelligence
University of Groningen
Groningen, The Netherlands
m.a.wiering@rug.nl

***Abstract*—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. In this paper it is investigated whether a reinforcement learning (RL) agent can successfully learn to play this game. The RL agent consists of a multi-layer perceptron (MLP) that uses a feature representation of the game state through input variables and gives Q-values for each possible action as output. For training the agent, the use of Q-learning is compared to two double Q-learning variants, the original algorithm and a novel variant. Furthermore, we have set up an alternative reward function which presents higher rewards towards the end of a level to try to increase the performance of the algorithms. The results show that all algorithms can be used to successfully learn to play Pac-Xon. Furthermore both double Q-learning variants obtain significantly higher performances than Q-learning and the progressive reward function does not yield better results than the regular reward function.**

***Index Terms*—Reinforcement Learning, Q-Learning, Double Q-Learning, Multi-layer Perceptron, Games**

I. Introduction

Learning by trial and error is the foundation of reinforcement learning (RL) [1]. Games provide suitable environments to model an agent that is trained using reinforcement learning to learn to distinguish desirable from undesirable actions [2]. Board games like backgammon [3] and more graphical intensive video games such as Ms. Pac-Man [4], regular Pac-Man [5] and multiple Atari games [6] have provided challenging environments for reinforcement learning numerous times. In the previously named video games a variant of reinforcement learning, Q-learning [7], has been widely applied. In the Ms. Pac-Man and Pac-Man papers a multi-layer perceptron (MLP) is used, while the Atari games were trained using Deep Q-Networks (DQN). The experiments in [4] and [5] 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 [6].

The Atari games agents are trained using raw pixel input as a state representation [6]. An advantage in using the raw pixel input is a better distinction between states since every individual game state presented to the algorithm is unique. The Ms. Pac-Man agent was trained using higher-order inputs [4], 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 [5]. 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. A variant to Q-learning called double Q-learning [8] decreases the optimistic bias in Q-learning and obtained a performance improvement for many Atari games [9].

We present our research on learning in the video game Pac-Xon using Q-learning and two double Q-learning variants: the original algorithm [8] and a novel variant, called double-B Q-learning. Pac-Xon is a difficult game with a large game-state space and many different levels. Each new level is a little bit more difficult than the previous one, thus 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 II. Using the previous research in Ms. Pac-Man and Atari games, we investigate the advantage of the double Q-learning algorithms in combination with higher-order inputs and an MLP. 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. Furthermore, we study if the novel double-B Q-learning variant performs just as well or even better than double 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. When starting a level, it is quite easy to obtain points. As the agent progresses, the risk that has to be taken to obtain the same amount of points becomes increasingly higher. Two different reward functions are tested. One reward function is proportional to the obtained score, whereas the second reward function takes level progress into

account by rewarding more in a later stage of the level.

Section II describes the framework we constructed to simulate the game and the extraction of the game-state features. Section III discusses the theory behind the reinforcement learning algorithms combined with an MLP. Section IV describes the experiments that were performed and the parameters we used. Section V shows and discusses the results that were obtained, and in Section VI we present our conclusions and give suggestions for future work.

II. Pac-Xon

Pac-Xon is a computer game implementing parts of the game-play of Xonix, which is derived from Qix (released in 1981 by Taito Corp.), combined with some game-play and graphics of Pac-Man. The game starts in an empty rectangle with drawn edges. Each level is a grid space with 34×23 tiles, with a player initiated on the edge and a number of enemies initiated in the level space. Figure 1 depicts a screen shot of the game, 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 it is not moving over empty space.

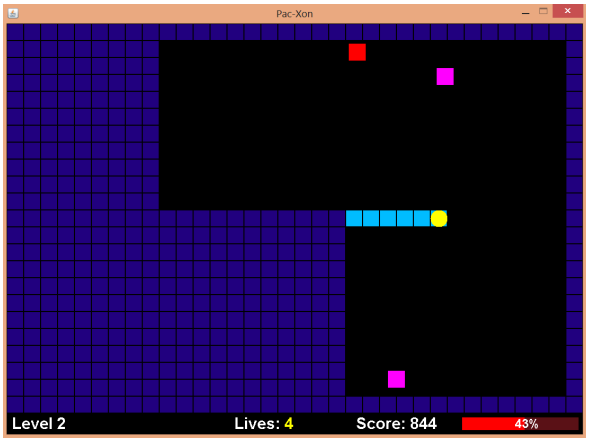

Fig. 1. Screenshot of the game, in which the following objects can be viewed: player (yellow), tail (cyan), normal enemies (pink), eater enemy (red), solid tiles and edges (dark blue).

A. Game-play

The main goal for the player is to claim empty spaces, while avoiding the enemies. A player drags temporary tiles, also addressed with *tail*, from edge to edge. A temporary tile becomes a solid tile after reaching another solid tile. In Figure 1, the tail is depicted in cyan. When enclosing an empty space (i.e. there are no enemies apparent), the empty space and the tail convert to solid tiles. The agent scores a point for each tile. If the tail did not enclose empty spaces, only the tail itself switches to solid tiles. A level is completed when the agent has conquered 80 percent of the total area, after which the player advances to the next level.

There are several 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. 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.

B. 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.

C. 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.

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.

D. 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, which contains the information that can be viewed on the screen. A total of 42 variables are computed, 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 is a number of features that produce multiple inputs (for example one for every direction). We have divided the features into three categories: Tile Vision, Danger and Miscellaneous.

Tile Vision (17 features)

1) **unFilledTiles**: We compute the normalized inverted distance towards the closest unfilled tile in each of the four directions. If there is no tile in the neighbourhood, this value is set to 0.
2) **tailDir**: Four binary values determined by whether there is an active tail in the four directions and within 4 blocks of the player. Set to 1 if there is, set to 0 if not.
3) **distToFilledTile**: In each of the four directions, we compute the normalized inverted distance towards the closest filled (solid) tile.
4) **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.
5) **unfilledTiles on row/column**: In the four directions, it calculates the normalized inverted distance towards the closest row or column with unfilled tiles.

Danger (17 features)

1) **enemyDir**: It determines the direction of the closest enemy. This value is either 0 = not moving towards player, 0.5 = moving towards player in one direction (x or y) or 1 = moving towards player in two directions (x and y).
2) **enemyDist**: The normalized inverted distance towards the closest enemy is computed in each direction. Set to zero if there is none.
3) **distToClosEnemy**: The Euclidean distance towards the closest enemy is inverted and normalized through a division by 20.
4) **distToCreeper**: The Euclidean distance towards the creeper (green enemy) is inverted and normalized through a division by 20.
5) **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.
6) **tailThreat**: The inverted Euclidean distance from the active tail towards the closest enemy is normalized through a division by 20. This value is set to zero if there is no active tail.
7) **tailHasBeenHit**: Binary value determines 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.
8) **creeperLocation**: Inverted normalized distance towards creeper (green enemy) is computed in the four directions. If there is no creeper in that direction, the value is set to zero.

Miscellaneous (8 features)

1) **playerDir**: Five binary values that determine the direction in which the player is moving. No direction (standing still) is the fifth option.
2) **numberOfFields (/enemies)**: It is based on the ratio between empty spaces and the number of enemies: 0 when there is only one field, and 1 when all enemies are separated. In Figure 1 this variable takes the value 0, but the value changes to 0.5 when and if the player successfully encloses the enemy.
3) **percentage**: It represents the normalized percentage of the level passed (filled with tiles).
4) **tailSize**: Normalized length of the tail is set to 1 if is larger than 40 blocks and to zero if there is none.

Figure 2 depicts some examples of the game-state features and how they relate to the on-screen information. Note that in Figure 1 the player is about to enclose an enemy. When this happens, the numberOfFields variable increases.

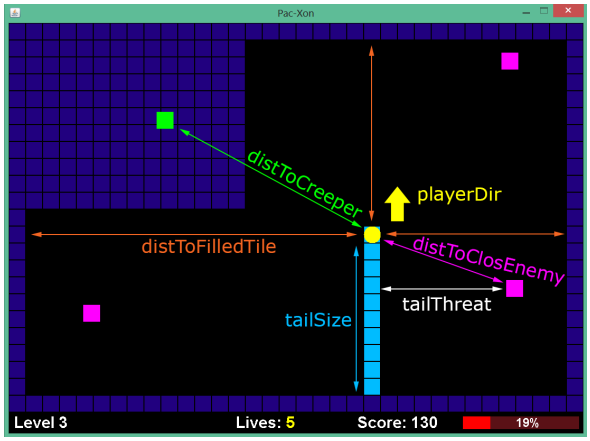

Fig. 2. State representation example, with the state variables depicted in arrows with their corresponding labels. The following objects can be viewed: player (yellow), tail (cyan), normal enemies (pink), creeper enemy (green), solid tiles and edges (dark blue).

III. Reinforcement Learning

A Markov Decision Process (MDP) [10] is the process of sequentially making decisions based on observations and being rewarded for reaching particular states. The dynamic environment of the game to be solved with reinforcement learning can be divided into five main elements: the learning agent, the environment, a policy, a reward function, and a value function [11]. The policy is a mapping from states to actions and generates a sequence of state-action pairs representing the global behavior of a learning agent. At time t, the policy selects and executes the 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 selects actions at random. As time passes, the agent learns

about its environment and obtains a policy that is based on rewards it has encountered in previous situations.

A. Q-Learning

The Q-learning algorithm [7] is a form of temporal difference learning predicting a quantity that depends on future values of a given signal. Each time step, the agent is in a state s_t, selects an action a_t and transits to a next state s_{t+1}, while obtaining a reward r_t. The goal is to maximize the overall intake of reward and Q-learning does this by using the experiences (s_t, a_t, r_t, s_{t+1}) to learn a state-action value function, $Q(s, a)$. The quantity $Q(s, a)$ is a measure of the total amount of discounted rewards expected over the future when the agent selects action a in state s and follows its policy afterwards. The Q-learning update rule in its general form is:

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

There are two constants that affect this equation. The constant α denoting the learning rate is a value between 0 and 1, and influences the extent to which the Q-values are altered following an action. The constant γ, called the discount factor, is also a value between 0 and 1, and determines how much influence the future rewards have on the updates of the Q-value. A discount factor close to 0 means the agent focuses on rewards in the near future, whereas using a discount factor closer to 1 means rewards obtained in the more distant future are weighed more heavily.

The tabular Q-learning algorithm uses a lookup table with state-action pairs to store and use information. Each state-action pair is associated with a Q-value that expresses the quality of the decision. The algorithm calculates these Q-values based on the reward received after choosing an action plus the maximal expected future reward. Since our state-action space is too large to store all Q-values in a table, we use a different approach that combines Q-learning with a function approximator explained in detail in Section III-C.

B. Reward Function

The reward function represents the desirability of choosing an action transitioning to some state. In a game, the reward values refer to specific events. The reward value is either 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 are compared 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 I. 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.

TABLE I
REGULAR (STATIC) REWARD FUNCTION WITH a = NUMBER OF TILES CAPTURED

Event	Reward
Level passed	50
Tiles captured	((a / 20) + 1)
Movement in empty space	2
Died	-50
No movement	-2
Movement in direction opposite from previous	-2
Movement over solid tiles	-1

The second approach is a progressive reward function that takes the level progress into account in the reward that is generated for capturing tiles. This level progress is expressed in the percentage of the level space that is filled with tiles. Since the player has to fill 80 percent of the level in order to pass it, this is the denominator used in calculating the level progress. This alternative reward function is shown in Table II. 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 II
PROGRESSIVE REWARD FUNCTION WITH a = NUMBER OF TILES CAPTURED AND b = PERCENTAGE OF THE LEVEL PASSED

Event	Reward
Level passed	50
Tiles captured	$(\sqrt{a \times b/0.8} + 1)$
Movement in empty space	2
Died	-50
No movement	-2
Movement in direction opposite from previous	-2
Movement over solid tiles	-1

C. Multi-Layer Perceptron (MLP)

Tabular Q-learning is not an alternative for games as Pac-Xon because of its huge state space given by too many different game states to keep track of. As a solution, we combine Q-learning with an MLP. An MLP is an artificial neural network which acts as a function approximator, such that 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 1.

We made an estimation of the number of game states in the first level. Any tile, but only one tile, can contain the player. There are two normal enemies that can exist on any tile that is not the border, which are 32×21 possible tiles. Any tile that is not the border can either be empty, conquered or contains the tail of the player. Therefore, we estimates the maximum to be $(34 \times 23) \times (32 \times 21)^2 \times 3^{(32 \times 21)} \approx 10^{329}$ possible

Algorithm 1 Q-Learning algorithm. The exploration algorithm is explained in section IV

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

game states in the first level. Note that this estimation gives a very optimistic idea of the scale since it contains many game states that are in practice not possible. However, this estimation considers only the first level; the computational complexity increases with each level because there are more and different kinds of enemies.

The implemented MLP consists of an input layer, a hidden layer and an output layer. Between layers a matrix with weights exists.

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

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 is used.

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 [12] but we decided to leave that for future research.

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. 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 IV) or the action with the highest Q-value from the network is chosen.

4) Backpropagation: The network makes use of data that is acquired dynamically while playing the game. Therefore it is considered online learning [13]. Updating the network is done by backpropagation [14]. The error required for backpropagation is the difference between the target Q-value (Equation 1) and the current Q-value for the selected action. Whenever a terminal state is reached, i.e. the agent either died or passed a level, Equation 2 is used since there are no future values to take into account.

Algorithm 2 Double Q-Learning algorithm

initialize s, Q_A and Q_B
repeat
  $pickrandom(A, B)$
  if A **then**
    if $explore()$ **then**
      $a \leftarrow randomMove()$
    else
      $a \leftarrow \arg\max_a Q_A(s, a)$
    end if
    $s^* \leftarrow newState(s, a)$
    $r \leftarrow reward(s, a, s^*)$
    $a^* \leftarrow argmax_a Q_A(s^*, a)$
    $Q_A^{target}(s, a) \leftarrow r + \gamma Q_B(s^*, a^*)$
    $update(Q_A(s, a), Q_A^{target}(s, a))$
  else if B **then**
    if $explore()$ **then**
      $a \leftarrow randomMove()$
    else
      $a \leftarrow \arg\max_a Q_B(s, a)$
    end if
    $s^* \leftarrow newState(s, a)$
    $r \leftarrow reward(s, a, s^*)$
    $a^* \leftarrow argmax_a Q_B(s^*, a)$
    $Q_B^{target}(s, a) \leftarrow r + \gamma Q_A(s^*, a^*)$
    $update(Q_B(s, a), Q_B^{target}(s, a))$
  end if
until end

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

$$Q^{target}(s_t, a_t) \leftarrow r_t \tag{2}$$

D. Double Q-Learning Variants

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 [8] 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 a Q-function is updated it computes the best action in the next state, but uses the Q-value associated to this best action of the other Q-function. This should reduce the optimistic bias since chances are small that both Q-functions overestimate on exactly the same states and actions. The target Q-value is computed using Equations 3 and 4. 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 2.

$$a^* \leftarrow argmax_a Q_A(s_{t+1}, a) \tag{3}$$

$$Q_A^{target}(s_t, a_t) \leftarrow r_t + \gamma Q_B(s_{t+1}, a^*) \tag{4}$$

TABLE III
PARAMETERS USED FOR THE EXPERIMENTS

Discount factor	0.98
Learning rate	0.005
Number of hidden layers	1
Nodes in hidden layer	50

We made a variant of double Q-learning, in which a Q-function is updated using the maximal Q-value of the other Q-function in the next state. Here, if network A selected the previous action, network B is used to compute both the best action in the next state and the associated Q-value. Because network B is now used two times, we call this alternative algorithm: double-B Q-learning. The target Q-value for network A that selected the previous action is now computed using Equations 5 and 6. The pseudo-code of the algorithm is very similar to the one shown in Algorithm 2.

$$a^* \leftarrow argmax_a Q_B(s_{t+1}, a) \tag{5}$$

$$Q_A^{target}(s_t, a_t) \leftarrow r_t + \gamma Q_B(s_{t+1}, a^*) \tag{6}$$

IV. Experimental Setup

A. Training the Agent

At initialization, an agent has absolutely no knowledge of playing 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.

1) Exploration: Each time-step, the agent chooses to pick an action from the network or performs a random move for exploration. The exploration method used is a decreasing ϵ-greedy approach [15]. 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 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 7.

$$\epsilon(E) = \begin{cases} 1 - \frac{0.9 * E}{50{,}000} & \text{if } 0 \leq \text{E} < 50{,}000 \\ \frac{750{,}000 - E}{700{,}000 * 10} & \text{if } 50{,}000 \leq \text{E} < 750{,}000 \\ 0 & \text{if } 750{,}000 \leq \text{E} < 1{,}000{,}000 \end{cases} \tag{7}$$

Whenever an agent has failed to obtain any positive rewards in 100 subsequent moves, a random move is performed as well. 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.

A total of 60 agents are trained. The agents are divided into six groups. Each group of 10 agents is trained using a unique combination of one of the reward functions and double Q-learning, double-B Q-learning or regular Q-learning. After a search through parameter space, we selected the hyper-parameters as shown in Table III.

B. 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 in which the agent died or got stuck are stored.

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

V. Results

In Figures 3 and 4 the training stage of the agents is plotted. Figure 3 shows the average level the agents reached with the reward function that is not related to the level progress. Figure 4 shows the average level that the agents reached when using the progressive reward function. In both graphs three lines are plotted representing Q-learning, double Q-learning and double-B 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. The learning curves show that the different algorithms learn with almost the same speeds, but that finally double-B Q-learning reaches the highest average performance.

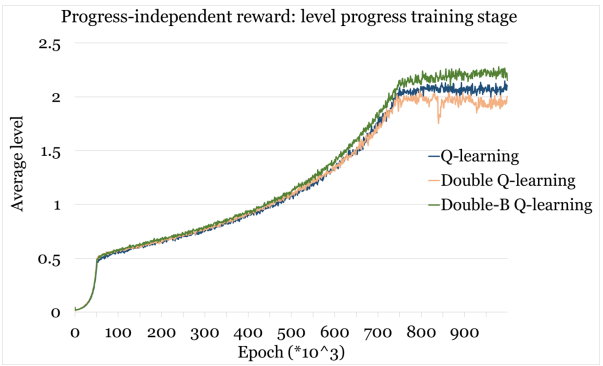

Fig. 3. Graphs of the training stage using the regular reward function. Each line is averaged over 10 trials.

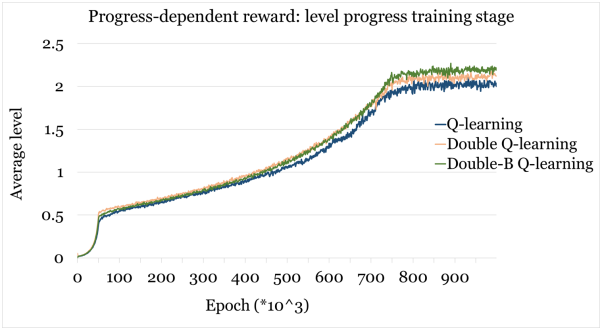

Fig. 4. Graphs of the training stage using the progressive reward function. Each line is averaged over 10 trials.

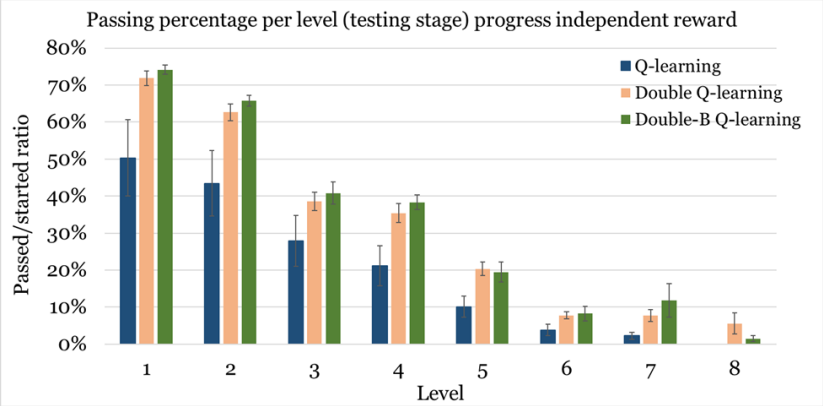

Fig. 5. Graphs of the testing stage with the progress independent reward function. Each bar shows the average passed/started ratio of 10 trials. The error bars show the standard error.

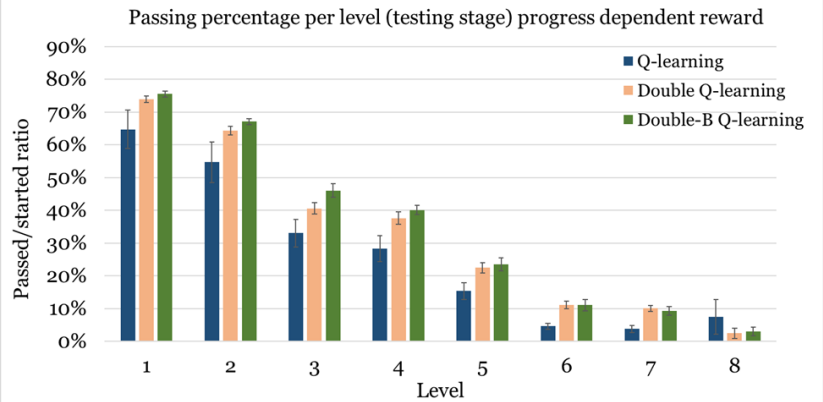

Fig. 6. Graphs of the testing stage with the progress dependent reward function. Each bar shows the average passed/started ratio of 10 trials. The error bars show the standard error.

TABLE IV
MEAN SCORES AND STANDARD DEVIATIONS OF THE EXPERIMENTS. ALGORITHM: Q = Q-LEARNING, DQ = DOUBLE Q-LEARNING, DBQ = DOUBLE-B Q-LEARNING. PROGRESSIVE: WHETHER THE REWARD IS DEPENDENT ON THE LEVEL PROGRESS. N: NUMBER OF TRAINING AGENTS. SCORE: MEAN SCORE OF THE TRAINED AGENTS IN THE TESTING STAGE. σ: STANDARD DEVIATION OF THE SCORES, SE: STANDARD ERROR.

						95% confidence interval			
Algorithm	Reward	N	Mean	σ	SE	Lower Bound	Upper Bound	Min	Max
Q	Regular	10	814	424	134	551	1076	0	1227
Q	Progressive	10	968	261	82	806	1129	427	1282
DQ	Regular	10	1091	146	46	1000	1181	811	1274
DQ	Progressive	10	1144	81	26	1093	1194	1021	1274
DBQ	Regular	10	1156	109	34	1088	1223	994	1335
DBQ	Progressive	10	1190	86	27	1137	1243	1080	1339

The results of the testing phase are shown in Figure 5 and Figure 6. Note that in the testing phase there is no exploration and learning, so all decisions made by the agents are chosen from the neural network and the network is not updated anymore. The figures show that the double Q-learning variants are able to pass more levels than Q-learning. The results also show that the novel double-B Q-learning algorithms performs slightly better than standard double Q-learning.

A more elaborate summary of the results can be found in Table IV. This table also shows that the highest scores are obtained with the double-B Q-learning algorithm. Furthermore, it shows that the progressive reward function results in more stable outcomes, as the standard deviation of the scores is lower given the same RL algorithm.

We performed a Tukey HSD post-hoc test to compare the results of the different RL algorithms to each other for which we used both reward functions. The results of this test are shown in Table V. The tests show that Q-learning is significantly ($p < 0.01$) outperformed by both double Q-learning variants. However, no significant difference between the results obtained by double Q-learning and double-B Q-learning is found.

If we again examine Table IV, we can observe that Q-learning sometimes obtains very low average scores. This shows that the double Q-learning variants are more robust in obtaining good performances in multiple experiments.

TABLE V
TUKEY HSD POST-HOC TEST

	diff	lower	upper	p-value
Q - DQ	-226.2	-395.1	-57.4	<0.01
DBQ - DQ	55.9	-113.0	224.8	0.71
DBQ - Q	282.1	113.2	451.0	<0.01

VI. CONCLUSIONS

In this paper we have examined two existing temporal difference learning algorithms, Q-learning and Double Q-learning, and introduced a variant of double Q-learning called double-B Q-learning. These algorithms have been combined with a multi-layer perceptron and extracted game-state features to learn to play the game Pac-Xon. We have examined different implementations of the reward function. We compare the 'standard' reward function using fixed values, and a progressive reward function, increasing the reward pursuant to level progress. Based on the outcomes of our experiments we conclude that all RL algorithms learn to play the game well with both reward functions. Our results also show that both Double Q-learning variants reach a significantly higher performance than Q-learning. The algorithms were able to reach a good performance in passing the first levels, and reaching levels that have revealed to be difficult for human players.

In future work, we want to compare the used RL algorithms to other algorithms such as actor-critic algorithms. Furthermore, we want to develop a learning method that allows the agent to predict the dynamics of the environment over multiple time steps. Such a model could then be used to improve the behavior of the RL agents even further. Finally, it would interesting to examine if double-B Q-learning also performs well on other problems and if it has advantages or disadvantages compared to standard double Q-learning.

REFERENCES

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

[2] J. Laird and M. VanLent, "Human-level AI's killer application: Interactive computer games," *AI magazine*, vol. 22, no. 2, p. 15, 2001.

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

[4] 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)*, April 2013, pp. 156–163.

[5] M. Gallagher and M. Ledwich, "Evolving Pac-Man players: Can we learn from raw input?" in *2007 IEEE Symposium on Computational Intelligence and Games*, April 2007, pp. 282–287.

[6] 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.

[7] C. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[8] H. van Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems 23*. Curran Associates, Inc., 2010, pp. 2613–2621.

[9] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning." in *AAAI*, vol. 16, 2016, pp. 2094–2100.

[10] R. Bellman, "A markovian decision process," *Indiana Univ. Math. J.*, vol. 6, pp. 679–684, 1957.

[11] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[12] 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*, 2018, pp. 29–40.

[13] L. Bottou, "Online algorithms and stochastic approximations," in *Online Learning and Neural Networks*. Cambridge, UK: Cambridge University Press, 1998, revised, oct 2012.

[14] D. Rumelhart, G. Hinton, and R. Williams, "Neurocomputing: Foundations of research," J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699.

[15] 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*, 2018, pp. 355–362.