Connectionist Machine Learning Techniques in a 2D Arcade Game

Bachelor's Project Thesis

Henry Maathuis, s2589540, h.maathuis@student.rug.nl,
Jeroen Langhorst, s2534657, j.langhorst@student.rug.nl,
Supervisor: Dr. M.A. Wiering

Abstract: Implementing intelligent behaviour in an agent for a computer game is often established by the usage of simple rules. Our research investigates the performance of Learning from Demonstration in combination with several techniques (Q-learning, SARSA and Supervised Learning) when learning policies for an agent in a 2D arcade game. For our experiment we constructed a 2D platforming arcade game using Java, in which the agent is controlled by Neural Networks. Experience Replay is explored to train the agent in an efficient manner. The goal was to see if the algorithms were capable of learning the desired policies. We also included several activation functions of the neural networks in our research: the traditional sigmoid, ReLU and a linear approach. We could not find any significant difference in success rate between implementations in the continuous environment. Also, we found that in a discretized environment Q-learning and SARSA in combination with the sigmoid should be avoided as it performs significantly worse than all other implementations.

1 Introduction

The first video-game was introduced back in 1958. *Tennis for Two*, it was called, and it was a simple tennis game on an analogue computer. In the game you played tennis against each other and the goal was to score as many points as possible. Since *Tennis for Two*, many other games were developed such as Pong, Pac-Man, and Mario. Conventionally opponents in video games are created using hard-coded rule sets, though Artificial Intelligence (AI) and Machine Learning algorithms (ML) are being investigated to control agents (adversaries and other non-player characters) in arbitrary video game environments. Previous research has shown success in learning an agent to play video-games such as Ms Pac-Man, Mario and other games using Reinforcement Learning (RL) techniques (Wiering and Van Otterlo, 2012; Bom, Henken, and Wiering, 2013).

1.1 Neural Networks

Artificial Neural Networks (NNs) are inspired by biological neural networks. Their use in ML is function estimation or approximation, which allows for it to be used as a technique in tasks such as handwriting or speech recognition. NNs are often used in systems to replace the logical or rule-based programming paradigms. Making correct inferences from scratch is a hard thing to do and is often needed to complete a plethora of tasks. This is one of the cases where NNs come in. Neural Networks are used by providing the network certain inputs, which is generally a set of numbers (called features as they describe certain aspects of the environment important to the network). For every feature vector the network returns an output value. The network can be constructed in such a way that the input, for example, is the state of the agent and the output of the network can represent how appropriate it is to take a particular action given the input state. A network is trained by propagating an error back through the network. Backpropagating the error updates the weights connecting the artificial neurons of the network.

1.2 Learning From Demonstration with Different RL Techniques

The algorithms that are discussed in this thesis are *Q-Learning* (Watkins and Dayan, 1992), *SARSA* (Rummery and Niranjan, 1994) and a basic *Supervised Learning* method. Learning from Demonstration uses these techniques in combination with Neural Networks to learn policies. Since the RL techniques are used in combination with Learning From Demonstration, pure RL is not used.

1.2.1 Q-Learning

Q-Learning is a very popular RL technique (Watkins, 1989; Watkins and Dayan, 1992; Sutton and Barto, 1998). Q-Learning uses the observation of the environment to estimate a value function (the Q-value function). This value function can be used to construct policies. Actions are selected by choosing the action with the highest corresponding Q-value in a given state. Several proofs exist that Q-learning eventually converges to an optimal value function given certain conditions (Tsitsiklis, 1994; Watkins and Dayan, 1992; Borkar and Meyn, 2000). The conditions that are important to find an optimal value function are the learning rate and the method of exploration.

1.2.2 SARSA

SARSA is a Reinforcement Learning technique similar to Q-Learning. Instead of updating state-action pairs by looking at the Q-value given a particular resulting state s_{t+1}, the algorithm looks at the resulting state s_{t+1} and action a_{t+1} for the Q-value function target. SARSA, which is an on-policy variant of Q-Learning, thus estimates the Q-value of the action at time step t by looking at the state-action pair valuations at time $t+1$ (Rummery and Niranjan, 1994).

1.2.3 Supervised Learning

Where RL does not use correct input-output pairs it can learn from, Supervised Learning (SL) does. It simply looks at the presented pair and reinforces on those values as the target values when re-evaluating the weights of the NN. Like RL, SL has been studied extensively (Sutton and Barto, 1998; Kotsiantis, Zaharakis, and Pintelas, 2007).

1.3 Research

As stated before, most game AI is simple and deterministic. An example might be: if an enemy detects a bullet within a certain range of him then the enemy should check for cover. In our research we investigate another possible implementation: the possibility of Machine Learning techniques to adapt an agent to play a 2D arcade game.

In our research we compare Learning from Demonstration using several techniques: Q-Learning, SARSA and Supervised Learning in its basic form. These are used in combination with Experience Replay and Neural Networks.

1.3.1 Research Question

Which algorithm can outperform the others in terms of training an agent to play a 2D arcade game when looking at the algorithms Q-learning, SARSA and Supervised Learning?

1.4 Overview

In the next section we will discuss and explain the environment as well as the agent of the game used in this research. The section after that will discuss the algorithms and the algorithmic extension in more detail after which a section called "Multi-layer Perceptrons" will give a brief overview of NNs as used in our research. Next, we describe how we got the results and follow up with a section which presents these results. Lastly, an in-depth discussion section and conclusion end this thesis, in which we will cover several key aspects of our research and conclude our findings.

2 The Game

In this section, a brief overview is given of the environment and the agent.

2.1 Environment

There are three different goals in the environment that we use to test the performance of the agent. The environment used is static, it does not change over time, and is of size 896x640 pixels. In the environment walls and platforms are present. Each goal is a certain position in the environment that an agent has to reach. The level of difficulty varies per goal. The three different goal states are referred to as *"Level 1"*, *"Level 2"*, or *"Level 3"*. Level 1 is the easiest goal to achieve whereas Level 3 is the most difficult one. In figure 2.1 a visualization of the environment is given.

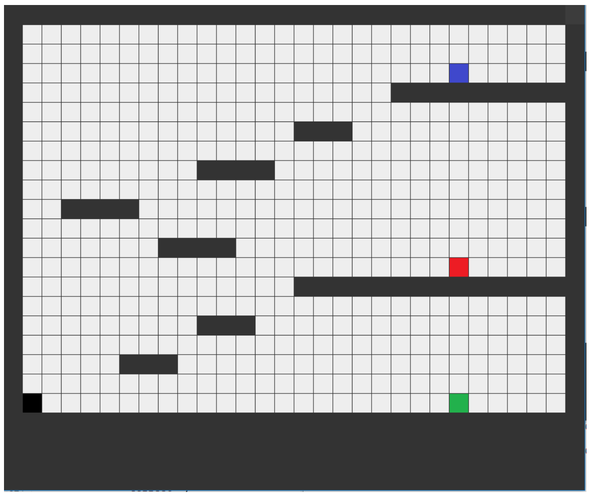

Figure 2.1: Level 1 is represented by the green square, Level 2 by the red square and Level 3 by the blue square. The agent's starting location is the black square in the lower left corner.

2.2 Agent

The goal of the algorithms is to train the agent in such a way that it is able to reach one of the three goal positions. The agent performs actions in the environment to update its state (which in our case is its discretized position in a grid of 28 X 20 cells as seen in figure 2.1. Each cell is 32 x 32 pixels). The action taken by the agent keeps being performed until the agent reaches a new state or it has been performing the action for 100ms.

The actions are all implemented to alter the X and Y velocities of the agent. Every game tick (1/60th second) the agent is updated with the X and Y velocities at that game tick. The following actions can be performed:

- The agent has the ability to move to the left. To do so it increases its X velocity by -0.5 pixels per tick (negative velocities will move the agent to the left).
- The agent has the ability to move to the right. To do so it increases its X velocity by 0.5 pixels per tick.
- The agent has the ability to jump. It increases its Y velocity by -30 pixels per tick (negative values make the agent move upward due to conventions in the implementation language). Per tick a gravitational constant of 10 is added to simulate falling. The agent falls down if there is no platform below it. During jumping and falling a directional key can be pressed so the agent gets both X and Y velocities and the agent can jump into a specified direction. When an agent collides with a platform its relevant velocity (for example Y when falling onto a platform) is set to 0 in order to stop it from moving through the platform.

Given the actions mentioned above, the agent has means to reach its goal position given an arbitrary position in the environment.

2.3 Variations

The standard version, used in almost all our testing, uses a continuous environment. This means the agent moves around the environment and keeps its momentum when stopping to perform an action using the following formulae where t is in seconds.

$$velocity_x = velocity_x * 0.92^t \tag{2.1}$$

$$velocity_y = velocity_y * 0.92^t \tag{2.2}$$

The parameter t denotes one game tick and 60 game ticks are executed per second. When a velocity lies within the -0.5, 0.5 range it is set to 0.

However, as will be discussed later on, a discretized version was needed for a few tests we did additionally. This discretized version keeps the same environment, but the agent now moves discretely through the state space. Meaning that performing the "move left" action will cause the agent to step one state to the left immediately. This eliminates any momentum there was in the standard version. Falling still works along the same principle, but now moves as the other actions: one state at a time, with the possibility to influence the direction of the fall by pressing a directional action key.

3 Algorithms

In this section a description of the algorithms used is presented. All implemented algorithms follow the same schematic as can be seen in figure 3.1. Q-learning and SARSA use the same discount factor γ of 0.99.

3.1 Learning from Demonstration

Learning from Demonstration (LfD) is a technique that can be used to train a Neural Network by generating the input-output patterns for the NNs, which in our case are state-action pairs. This means that we can guide the agent through the environment to obtain specific experiences. These experiences can be used to train the agent. Someone who has played tennis or table tennis might have experienced LfD from their trainer. Whenever someone makes a bad swing with their racket or bat, the trainer often takes your arm and guides your arm such that you get the correct swing. There are several reasons to implement LfD in combination with Machine Learning to learn a policy for an agent (Atkeson and Schaal, 1997a,b).

- When learning a value function we can obtain a more favourable initial policy by using the demonstration data. This allows us to prime the value function.
- When dealing with a large state space, it might be near impossible to reach a goal state with the use of default exploration techniques (ϵ-greedy etc).
- Some applications have shown that Learning from Demonstration entails greater reliability. In (Atkeson and Schaal, 1997a) they describe that a robot has learned to balance a pole given only one demonstration with great reliability.

3.2 Q-Learning

Q-learning (like other RL techniques) needs some kind of method to see which state-action pairs are favourable and which ones are not. In order to do so it uses NNs which approximate a Q-value function per action. The functions return a Q-value for each state-action pair (s,a), where the value of it denotes how good a given action is in a state. The higher a Q-value is, the better a state-action pair is valuated and the best action is chosen by choosing the action leading to the highest Q-value given an arbitrary state. The network corresponding to the chosen action is trained on the following calculated target Q-value:

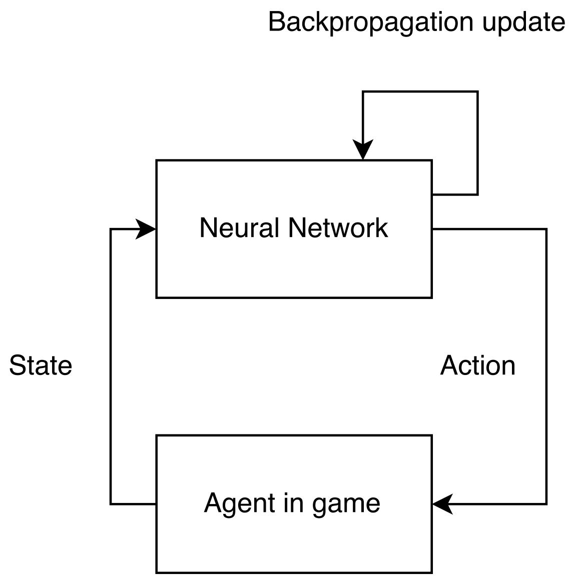

Figure 3.1: Diagram of ML algorithms used

$$Q(s_t, a_t) = r + \gamma * \max_a Q(s_{t+1}, a) \tag{3.1}$$

In this formula s_t is the state at time t with a_t being the corresponding action. s_{t+1} is the resulting state at time $t + 1$. r denotes the reward of the current state.

3.3 SARSA

Similarly to Q-learning, the agent is reinforced on the experiences. Instead of maximizing the Q-value of the resulting state, a Q-value is directly calculated using an additionally stored action, which results in the following computation of the target value for performing action a_t in state s_t:

$$Q(s_t, a_t) = r + \gamma * Q(s_{t+1}, a_{t+1}) \tag{3.2}$$

3.4 Supervised Learning

Supervised Learning (SL), unlike the previously mentioned algorithms, has no need to approximate the right decision in a given state by way of calculating a Q-value. The NNs in SL learn to return action probabilities, meaning it returns a chance of performing a certain action in the current state s_t. SL then chooses between all actions based on these action probabilities. Using the output of the NN for an action a_t, the corresponding probability is calculated as follows:

$$P_{a_t} = a_t / \sum a \tag{3.3}$$

It is simply directly trained on the presented state-action pair and discourages the use of any other action by setting the desired output for those actions to zero. For example, the desired action in a state is left. Then the network for the left action receives a target value of 1, meaning an action selection probability of 100%, to backpropagate on. The networks for right and jump receive 0 as a target value to suppress those actions in that state.

3.5 Algorithmic Extension

All techniques described above are used in combination with Experience Replay. This is done to speed up the learning process.

3.5.1 Experience Replay

Experience Replay (ER) is often used together with LfD. ER is used to train the agents more often on the experiences that are collected during LfD. We do this by reiterating over past experiences, which is done until the policy does not change for 50 epochs (an epoch is training on the whole experience set one time).

A reason to use ER is when you only have a few experiences to work with (Adam, Busoniu, and Babuska, 2012). An adaptive agent can benefit from this approach since it quickly learns a policy given only a few experiences. While LfD is performed real-time or online, ER is an offline learning technique and thus can generally be run more quickly as it does not need to update any visual components - making training an agent faster as well.

3.6 Collecting Experiences

To train the network, experiences are gathered by a human playing the game. An experience is gathered by continuously checking whether or not the player is in a new state. Once a new state has been achieved, the last performed action is extracted. If two actions (for example left and jump) were selected we only get the action taken first as in the current implementation only one action can be taken in a state. We determine the reward of the resulting state and store the experience as a *[state s_t, action a_t, state s_{t+1}, reward r_t]*-quadruple. All experiences are then saved for later use. Once the goal state has been reached, the agent is trained on those experiences using ER. Note that for SARSA an experience is of the form *[state s_t, action a_t, state s_{t+1}, reward r_t, action a_{t+1}]*.

3.7 Rewards

The agent is granted either one of two rewards. If the agent hits the goal state, the agent gets a reward of +100. In any other state the player can be in the player will obtain a reward of 0 (non-goal state).

3.8 Stop Criterion

During Experience Replay the selected action of each state is subject to change. The program terminates when there are no changes in the policy for 50 epochs (50 times of running through all the acquired experiences). Once this condition has been met, training will be finished and the networks and experiences are saved for testing. If the stop condition is not met the training is done up to 5000 epochs.

4 Multilayer Perceptrons

In our research we used NNs with a hidden layer of 60 nodes and a single output node for each of the actions. Thus we used three NNs for the agent. Each NN is trained for a different specific action (i.e. jump, left, right).

4.1 Input Vector

All ML techniques receive the same data to work with. The input vector is composed of the position of the player in the grid as seen in 2.1. The velocities of the agent are not put into the input vector.

Although the player can navigate continuously in the environment, the position is always determined to be inside one of the squares. It does this by taking the center of the agent. This means that there are 28 different x-positions and 20 different y-positions the agent can be in. Thus there are 560 different x, y coordinates. The input vector therefore holds 560 input units and every input unit either has a value of 0 or 1, representing a single square in the environment. If the input unit represents the current position of the player, the value is 1 and in any other case the value will be 0.

4.2 Different Activation Functions

All of the algorithms described are tested using three different activation functions for the hidden layers in the MLPs. The output layer uses the linear activation function. In the equations below x represents the summed hidden value. All NNs have a learning rate of 0.001, except for the tabular approach which uses a learning rate of 1.

The 'standard' sigmoid function is used which is denoted by the following equation:

$$f(x) = 1/(1 + e^{-x}) \tag{4.1}$$

Note that the sigmoid function uses bias values and updates these as well. The following activation functions as implemented by us do not use bias nodes.

The rectifier linear unit (ReLU) is also used as an activation function and can be expressed as:

$$f(x) = max(0, x) \tag{4.2}$$

Finally a linear activation function is tested as well:

$$f(x) = x \tag{4.3}$$

Lastly, we included a tabular approach which has the same activation function as the linear approach. The tabular approach only has one hidden unit instead of 60, which the other approaches have in our research. Another difference is that the weight from the hidden to output layer is not altered and always remains 1. This is similar to a lookup table as the input is directly linked to the output value.

5 Testing the Algorithms

To assess the performance of all Machine Learning techniques, several different measures are taken.

- *Number of epochs* The number of epochs ER trains on all the experiences until the stop criterion has been met or the algorithm has trained up to 5000 epochs. We measure this as it is a good way of finding out which algorithm converges fastest.
- *Training time* (measured in seconds) This is the total amount of time spent in training the algorithm using ER.
- *Testing time* (measured in seconds) For the standard, continuous, environment this measurement is preferable as the number of actions taken does not translate directly to the performance of a solution in this environment.
- *Success rate* This is the number of times the goal was reached during testing over the total number of test runs starting from the same initial position.
- *Correct actions learnt* Every experience is stored and we can check whether the action stored in the experience set is the same action as the network chose upon testing. This measurement is taken as a means to show how well an algorithm is able to follow the policy generated by the human player. As the experiences are composed of several runs there might be conflicting state-action pairs. Thus a 100% correct policy will most likely not exist.
- *Number of steps* In one of our points of discussion we also include the number of steps (i.e. actions) taken in the discretized environment. We opted to choose this as in this discretized environment the solution speed has no real value when trying to reason about the quality of a solution.

6 Results

Results are presented per level of difficulty. The data in the tables are presented in the following format: mean (standard deviation) where applicable.

Ten training runs were performed per level and for each algorithm/update-rule pair (this can be seen in the success-rate going up in steps of 10%). Each training run the NNs are reinitialized randomly, meaning the values of the weights are reinitialized randomly. The training run consists of first performing ER for up to 5000 epochs or until the policy converges (i.e. the policy stays the same for 50 epochs) and then testing the system for up to 60 seconds (meaning an agent has to reach the goal state in 60 seconds or the test will result in a failure). This is done to prevent getting stuck in an endless loop. A single epoch means performing ER on the complete set of gathered experiences once.

Failed test runs (i.e. when the agent did not reach the goal state) were omitted in the collection of these results from the *Testing time* measurement as to not give a distorted image regarding actual performance of the solutions that were found.

6.1 Continuous Environment

In tables 6.1 up to and including 6.9 we show the results as obtained in the continuous environment. The number of experiences for each level were: 24 for Level 1, 392 for Level 2 and 1032 for Level 3. SARSA has one less experience in all levels as a single experience there stores both the current and the next state-action pair, resulting in N-1 number of experiences being stored.

	Sigmoid	ReLU	Linear
Number of epochs	51 (0)	75 (6)	186 (38)
Training time (s)	17.92 (6.22)	22.45 (1.91)	50.54 (10.51)
Testing time (s)	24.30 (0.06)	24.23 (0.13)	24.50 (0.53)
Success rate	100%	100%	100%
Correct actions	24/24	24/24	24/24

Table 6.1: Level 1 - Q-Learning

	Sigmoid	ReLU	Linear
Number of epochs	51 (0)	71 (7)	173 (35)
Training time (s)	23.43 (14.40)	24.66 (7.00)	52.93 (10.45)
Testing time (s)	23.34 (0.15)	24.33 (0.13)	23.41 (0.11)
Success rate	100%	100%	100%
Correct actions	23/23	23/23	23/23

Table 6.2: Level 1 - SARSA

	Sigmoid	ReLU	Linear
Number of epochs	5000 (0)	5000 (0)	5000 (0)
Training time (s)	428.68 (8.21)	294.34 (18.38)	296.84 (5.35)
Testing time (s)	24.59 (0.42)	24.94 (0.46)	26.66 (2.94)
Success rate	100%	100%	100%
Correct actions	24/24	24/24	24/24

Table 6.3: Level 1 - Supervised Learning

	Sigmoid	ReLU	Linear
Number of epochs	259 (41)	201 (7)	182 (1)
Training time (s)	229.02 (38.55)	153.28 (5.78)	136.05 (1.28)
Testing time (s)	27.56 (0.50)	30.12 (1.02)	33.54 (9.03)
Success rate	40%	50%	100%
Correct actions	284.6/392	354.0/392	351.0/392

Table 6.4: Level 2 - Q-Learning

	Sigmoid	ReLU	Linear
Number of epochs	213 (2)	129 (7)	103 (0)
Training time (s)	193.49(4.66)	107.11 (5.00)	84.28 (1.29)
Testing time (s)	30.51 (15)	29.37 (1.74)	34.82 (11.94)
Success rate	90%	70%	50%
Correct actions	342.0/391	351.4/392	356.3/392

Table 6.5: Level 2 - SARSA

	Sigmoid	ReLU	Linear
Number of epochs	5000 (0)	5000 (0)	5000 (0)
Training time (s)	3821.02 (74.01)	3516.97 (45.14)	3519.66 (75.58)
Testing time (s)	36.08 (4.29)	35.01 (3.46)	36.56 (5.75)
Success rate	60%	90%	90%
Correct actions	363/392	363/392	363/392

Table 6.6: Level 2 - Supervised Learning

	Sigmoid	ReLU	Linear
Number of epochs	298 (24)	153 (27)	117 (0)
Training time (s)	567.30 (44.85)	257.83 (43.86)	196.64 (1.74)
Testing time (s)	-	77.47 (17.76)	91.28 (0.0)
Success rate	0%	30%	10%
Correct actions	300.6/1032	817.4/1032	824.0/1032

Table 6.7: Level 3 - Q-Learning

	Sigmoid	ReLU	Linear
Number of epochs	423 (18)	220 (19)	144 (8)
Training time (s)	798.02 (41.15)	344.77 (29.95)	254.05 (13.70)
Testing time (s)	-	76.48 (40.10)	-
Success rate	0%	60%	0%
Correct actions	440.0/1031	812.0/1031	808.0/1031

Table 6.8: Level 3 - SARSA

	Sigmoid	ReLU	Linear
Number of epochs	5000 (0)	5000 (0)	5000 (0)
Training time (s)	9938.59 (341.63)	9223.14 (159.0)	9192.86 (79.05)
Testing time (s)	449.92 (0.0)	56.46 (0.0)	321.35 (25.40)
Success rate	10%	10%	20%
Correct actions	898/1032	898/1032	898/1032

Table 6.9: Level 3 - Supervised Learning

6.2 Discretized Environment

The number of experiences here was 110 and consists of one trial - a single run of a human player generating experiences. All of this testing was done on the Level 3 environment. The results are shown in tables 6.10 - 6.15.

	Sigmoid	ReLU
Success rate	100%	70%
Correct actions	94/110	94/110
Number of steps	79.1 (14.6)	86.9 (23.3)

Table 6.10: Supervised Learning

	Linear	Tabular
Success rate	80%	100%
Correct actions	94/110	94/110
Number of steps	78.3 (24.2)	57 (0)

Table 6.11: Supervised Learning cont.

	Sigmoid	ReLU
Success rate	0%	100%
Correct actions	32.9/110	94/110
Number of steps	-	57 (0)

Table 6.12: Q-learning

	Linear	Tabular
Success rate	100%	100%
Correct actions	94/110	94/110
Number of steps	57 (0)	57 (0)

Table 6.13: Q-learning cont.

	Sigmoid	ReLU
Success rate	0%	100%
Correct actions	44.6/109	79/109
Number of steps	-	57 (0)

Table 6.14: SARSA

	Linear	Tabular
Success rate	100%	100%
Correct actions	79/109	79/109
Number of steps	57 (0)	57 (0)

Table 6.15: SARSA cont.

6.3 Results Random vs Non-Random Actions in SL

As the Supervised Learning algorithm has a different method of choosing its actions based on action probabilities instead of a valuation, we opted to test if always choosing the highest probability (the values returned now acting much like the Q-values returned by Q-learning) would result in different results. This was only done for level 3 on the linear activation function. The results of these tests can be found in table 6.16 below. More on this in the discussion section.

Number of epochs	302 (40)
Training time (s)	518.12 (180.03)
Testing time (s)	0.0 (0.0)
Success rate	0%
Correct actions	898/1032

Table 6.16: Level 3 - Linear SL without random actions

6.4 Results when Altering Experience Pool Size

In tables 6.17 - 6.20 we shown the results obtained when looking into how many trials are preferable. A trial is a single instance of the human player playing the game to generate the experiences - in other words, how many paths are present in the experience set.

Number of trials	1	20
Number of epochs	943 (64)	93 (27)
Training time (s)	275.41 (31.75)	122.26 (38.47)
Testing time (s)	30.83 (1.14)	30.76 (0.79)
Success rate	100%	90%
Correct actions	39/39	704.5/787

Table 6.17: Level 2 - QLearning (ReLU)

Number of trials	30	40
Number of epochs	83 (10)	85 (19)
Training time (s)	151.19 (18.36)	196.80 (42.53)
Testing time (s)	29.66 (85.0)	29.43 (0.91)
Success rate	70%	100%
Correct actions	1061.4/1181	1432.6/1574

Table 6.18: Level 2 - QLearning (ReLU) cont.

Number of trials	1	20
Number of epochs	1262 (184)	130 (15)
Training time (s)	490.95 (73.28)	395.44 (44.01)
Testing time (s)	-	-
Success rate	0%	0%
Correct actions	87/96	1469.2/1963

Table 6.19: Level 3 - QLearning (ReLU)

Number of trials	30	40
Number of epochs	120 (19)	117 (22)
Training time (s)	482.83 (77.41)	619.0 (118.65)
Testing time (s)	-	-
Success rate	0%	0%
Correct actions	2085.1/2883	2793.1/3791

Table 6.20: Level 3 - QLearning (ReLU) cont.

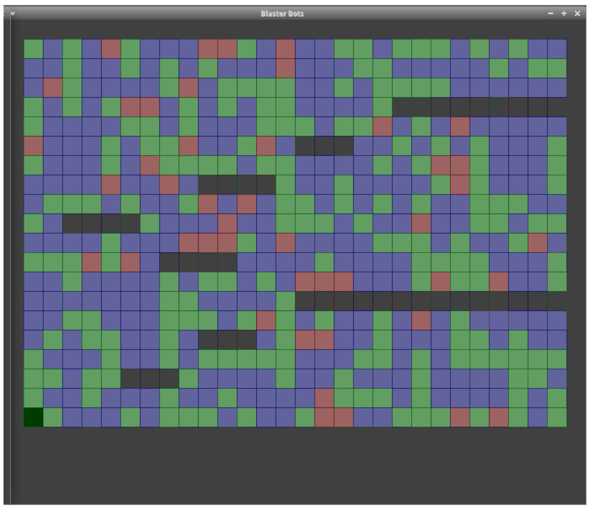

Figure 7.1: Example policy using Q-learning

7 Discussion

7.1 Analysis of Policy Images

Even though the algorithms converge to mostly the same policy following the experiences presented to it in ER, there are still a lot of differences in the final policy. Shown in figures 7.1 and 7.2 are two policies generated by the Q-learning algorithm using the ReLU update rule at the end of training. These tests were run in the Level 3 environment. The colour of the block denotes the action of the policy in that state. Red means the agent should move to the left, green move to the right and blue jump.

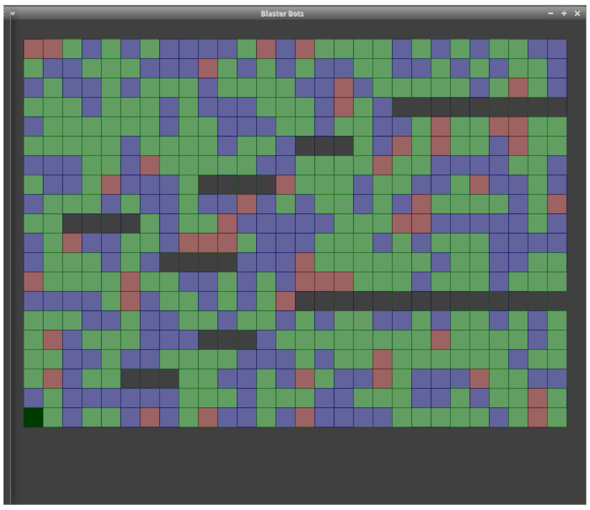

Figure 7.2: Second example policy using Q-learning

When looking at figures 7.1 and 7.2 we can first notice the similarities between the two. Along the path (going from the lower left corner to the goal state) the same key locations share the same action chosen. This, in theory, suggests that the policies made should be able to make the agent find the goal during testing. But, as our results section has shown, this is not the case.

The reason for this can be seen in the locations for which the algorithm has no experiences (for instance in the case of Level 3 the area to the lower right corner). When looking at the two figures we can see key differences in the policies of the top left area. Jumping is far more prevalent in figure 7.1 than in figure 7.2 Even in general, there seems to be more of a tendency to select the jump action in the policy of figure 7.1 than in the other policy.

This difference between the policies gives us an indication as to why even though the policy seems to be fine, the algorithm still cannot find a solution during testing. When testing, the agent is moving whilst still having momentum. So it can end up slightly sliding into an unknown state, which it will then have no good policy for. So, blindly following the policy will end up making the agent move to an undesirable location in these untrained states. More on this later in section 7.6, as we look at the random action selection used in SL.

7.2 Sigmoid

As can be seen in the results section, there is quite a difference in the performance of the sigmoid activation function. It goes from being the best in Level 1 to being the worst in the other two levels when looking at a combination of the success rate, number of epochs and training time. This made us look into this activation function a bit more. Even though it outperforms in combination with SARSA on Level 2 it performs worse in all the other Level 2 and 3 trials.

In practice, we have observed that the sigmoid first will move to a single action policy and only then slowly begin to fill in other actions and converge to the final policy. This initial stage of the policy is achieved rapidly, but it's the later learning part that takes rather long due to, amongst others, the more computationally heavy calculations made when compared to the ReLU or the linear approach.

7.3 The Difficulty of Level 3

The results show that the Level 3 environment was by far the hardest environment to master for the algorithms. The performance of the algorithms drop when going from Level 2 to Level 3. Looking into why this drop might occur we have made two key observations:

- First of all, the goal state is a lot further away from the start position. Not only that, but it is also a far more complex policy that the algorithms need to learn. More jumping and platform traversal is needed for it in order to reach the goal. This adds a lot more states in the policy where minor mistakes may be made, which can accumulate to the agent falling of or otherwise not finding the goal. These minor mistakes can be, for example, the momentum of the agent taking it to states it has not trained for or SL randomly choosing a suboptimal action.

- The second observation was that in Level 3 state ambiguity is first introduced as a major cause for the agent not finding a goal. As the state has no idea of what the previous state was it will only learn a single action per state. This results in situations as seen in figure 7.2 where around the points where the agent should turn around only a single action is available. Thus the agent may not be able to get long enough of a run up or have the momentum to make a jump. More on this in the next section.

7.4 Discretization of the Game

Due to the poor performance in the continuous environment, we discretized the movement of the agent. This means that the agent moves one square at a time and at all times is exactly in one of the squares as seen in figure 2.1. A single trial was used for ER and was made by moving in the discretized environment as can be seen in figure 2.1. The results of the discretized setting indicate an improvement over the setting in which the movement of the agent is continuous if we are looking at it in terms of success rate.

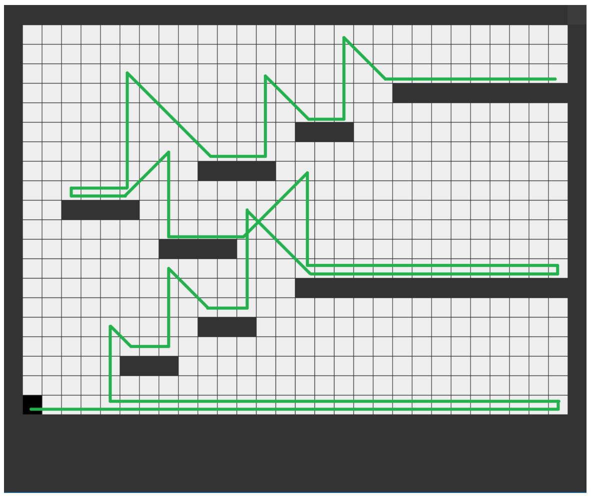

Figure 7.3: Trial for discretized environment

Using tabular learning, all the algorithms obtained a success rate of 1. Using Q-Learning and SARSA we can observe that the success rate is also 1 using the ReLU or Linear activation functions. The sigmoid however performs poorly with Q-Learning and SARSA, both obtaining a success rate of 0. Using Supervised Learning we observe that the sigmoid performs best (with a success rate of 1).

The latter surprises us since the sigmoid performs very poor in combination with Q-Learning or SARSA. A reason for this change is that SL can escape loops and unfortunate locations as SL uses action probabilities instead of a value function.

7.5 Effect of Experience Pool Size

One of the things we wanted to look into after our main research was done, was the effect of the number of trials on the performance of the algorithms. In testing this we ran Q-learning with the ReLU update function. Results for this can, like the normal results, be found in the results section. We compare the data in section 6.4 to the data in section 6.1 as section 6.1 has the data for runs of 10 trails.

For the Level 2 environment we can see that having just one trial is the best in terms of success rate. However, the training time is a lot larger.

Level 3, on the other hand, tells a completely different story. In this environment we see that 10 trials seems to be the right number of trials as this is the only number of trials for which Q-learning in combination with the ReLU activation function is able to find solutions.

7.5.1 Level 2

Table 6.17 on page 8 shows that only one trial is needed to adapt the agent to reach the goal state in Level 2.

It is also observed that the actions in the experiences are exactly followed by the learned value-function. Thus, the behaviour of the agent is identical to that of the demonstrator. This agrees with what has been found earlier (Atkeson and Schaal, 1997a), which states that one trial might be sufficient to learn a reliable policy.

Using more trials we see that the amount of correct policies and success rate decreases. This might be due to the agent learning multiple paths towards the goal, and afterwards the agent would take a mixture of the different trials which leads to undesirable sequences of actions.

However when using 40 trials, a success rate of 1 is again obtained. This is due to either adding experiences that are similar to other experiences, preferring the actions that are performed most often, or chance can be involved.

7.5.2 Level 3

In contrary to the results obtained using Level 2, we see that Q-Learning in combination with ReLU performs best using 10 epochs yielding a success rate of 0.3 as can be seen in table 6.7 which contains the data of the original continuous environment experiment with 10 trails. Since the environment is more difficult than the previously discussed level 2, we believe that only one trial is insufficient to learn a correct policy in the continuous environment for Level 3.

7.6 Supervised Learning and Random Actions

Another point of discussion is the random action selection used by SL. Instead of choosing an action based on a deterministic value like our implementation of Q-learning it selects an action based on the probabilities returned by the network. We thought this might have been affecting the performance of the algorithm, so we ran the level 3 environment again on the linear activation function version of SL. The results can be found in table 6.16 on page 8 in the results section. We will be comparing it to the data of the same algorithm-activation function pair in table 6.9 on page 8.

As we had theorized there does seem to be a difference in performance. We see that, first of all, the updated SL method does not find the goal state at all. But when we look at the number of correct actions we found that both the random and non-random versions converge to the same number of correct state-action pairs. This provides a good basis for the idea that the random action selection used in our research did affect the performance of the SL algorithm. We hypothesize that this is due to it being able to break out of locations where it normally would get stuck.

8 Future Work

8.1 Feature Vector Improvements

As the input of the NNs used was a fairly simple and discretized vector representation of the state there is bound to be found an improvement in performance by improving the input vector used. There are several improvements we suggest that mostly try to combat the ambiguity in the states currently used.

- Firstly, one could add 3 more elements to the vector representing which action was taken last time an action was chosen. The reason this would help is to give a bit more direction to the input vector. In the current input no information from before the current state is given which is not enough information to discern between a location where the agent should turn around.
- The second thing that could be tried is something in the same vein as the previous one. Instead of directly adding actions to the vector we could add the previous state to the current one. This way we could add information about the last location. Multiplying it with some value α will make it sure that it is not as influential as the current state. The exact value of α is an option that should be explored.
- Also, the movement of the player character is not instantaneous but has a short speed-up and slow-down effect (the agent glides around slightly). Another feature that could improve performance would be the current speed or acceleration of the agent.
- The next improvement we mention here would be to transform the discrete states used to continuous ones (i.e. not have 1 value for a single location in a 560 length state vector but a continuous (x, y) pair as input). Additionally a Radial Basis activation function (Broomhead and Lowe, 1988) should be implemented and the velocities of the agent should be included in the vector as well. In doing so research can be done to see whether or not the performance could be improved in speed (generating the example continuous state would be more efficient than the current one). If the success rate changes as well it could be evidence that the algorithms also work for continuous inputs.
- Lastly, as the previous possible improvements have been about adding more information to the current state we should not forget the ability to handle it. We think that when adding more information the network will have to combine the data to be effective. Thus, we suggest looking into MLPs with more hidden layers which might allow the networks to combine information more effectively.

8.2 Alternative Reinforcement Learning Algorithms

In our research only a few RL techniques were tested. They were also tested solely using Learning from Demonstration and Experience Replay and without autonomous exploration. To better develop the knowledge of how RL relates to arcade games and video games, exploration methods and other algorithms should be tested.

Algorithms such as Actor-Critic Learning (ACL) (Konda and Tsitsiklis, 1999), NeuroEvolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen, 2002) or Neural Fitted Q-iteration (NFQ) (Riedmiller, 2005) are good suggestions. Some of these algorithms have even already been used to learn to play games with (Shantia, Begue, and Wiering, 2011).

8.3 Other Extensions and Future Work

Besides the improvements and ideas noted in earlier sections there is a plethora of extensions and improvements that could be implemented and researched in future work. We shall list some of the ones we think are most valuable to research.

- Instead of the current goal of the game (finding and moving to a goal) other goals could and should be implemented into the game to test the algorithms on other tasks besides finding a goal. Examples of this could be evading hostile targets (non-player characters that kill the player on contact, spike traps or even bullets passing through the level at set intervals). This would change the objective from finding a goal state to not only finding said goal state but also dealing with the obstacles on the way to the goal state. Implementing this will give us a good insight into the ability to handle obstacles and a dynamic environment. Not only that, but it will also tell us about the ability to handle two goals at the same time. One being the problem of navigating to the goal state and the other being the goal to not get killed in the process of achieving the original goal.

- Continuing with the idea of altering the goal of the game but in a different manner is the idea of having multiple agents in an environment working against one another to achieve some goal. For example, a simple game in which the agents have to shoot bullets at each other to get points for eliminating their opponents. This would not only test the algorithms in moving and achieving goals, but also dealing with other agents and their behaviour. This is already coming quite close to agents in modern video games and thus is good to look into.

- An alternative idea to collect experiences would be to first have a human play the game several times, after which tests are made, and if unsuccessful the experiences from the failing tests would be added to the experience set. In this way the agent does not only learn from positive experiences and could learn to 'repair' a wrong policy. Note that this only works when using Q-learning or Sarsa, and cannot be combined with the pure supervised learning approach.

- The last piece of future research we suggest is to try and see if these algorithms can be altered to explore the environments on their own and thus will no longer need human players for the generation of experiences. In order to do this, exploration methods should be included and engaged to generate the experiences needed for ER besides the online learning that is needed.

9 Conclusions

We constructed a 2D arcade game environment in which an agent is able to act. In the arcade game two different game logics for the agent have been implemented, a continuous one and a discretized one. The continuous game logic for the agent functions so that the agent is able to move on a pixel level. The discretized game logic means that the agent moves a fixed amount of pixels, 32 pixels at a time, in a certain direction. Learning from Demonstration was used in combination with several RL techniques. The implemented techniques are Supervised Learning, Q-Learning and SARSA. Experience Replay is used as an algorithmic extension to train the agent more effectively. A connectionists approach was used to learn the different value-functions and, in the case of Supervised Learning, action probabilities. To test the performance of the algorithms, three different levels were developed and we measured the performance of the agent in each level based on, amongst others, success rate and training time.

Three different levels are learnt by the agent.

An overview of the levels can be seen in figure 2.1 on page 3. Each level takes place in the same environment but with a different goal location and ups the difficulty.

9.1 Findings Using the Continuous Environment

First off we discuss the performance of the algorithms in the Level 1 environment. We obtained for every algorithm and activation function the same success rate, namely 100% and all the demonstrated actions are correctly learned. Thus we can conclude that all methods could be used.

In level 2 there are several algorithm-activation function pairs with a good success rate (success rate $\geq$ 90%). These are Supervised Learning with either ReLU or linear activation, SARSA in combination with the sigmoid or Q-learning with the linear activation function. Of these, Q-learning with linear activation seems to outperform the others.

In Level 3 we can see that all algorithms drop in performance and never reach above 60% success rates. SARSA with the ReLU activation function has the highest success rate of 60% whereas the rest of the algorithms have success rates in the range of 0-30%.

A chi-square test of independence was performed in which the success rates of the different algorithms are compared. A post hoc test is performed to find significant differences between any of the nine algorithms across levels 2 and 3. An initial glance at the results indicate that there are significant interactions, however after correcting for multiple hypotheses using Bonferroni no significant interactions were obtained ($p \geq 0.05$ for all interactions). There is no statistical evidence to prefer one algorithm in combination with a particular activation function over another given the success rate. This might be due to a small sample size.

In conclusion: there seems to be no significant evidence to use any method over another, though we do have pointers that in the simple environment all methods are viable. The other two levels we can make no claims about.

9.2 Findings Using the Discretized Environment

Whereas the success rates in the continuous environment in Level 3 were not that high we found something completely different in the discretized environment. In this scenario the only two combinations that performed poorly were SARSA and Q-learning when combined with the sigmoid. Besides this the lowest success rate achieved was 70% by Supervised Learning using the ReLU activation function. Apart from combinations with SARSA all combinations, when finding the goal at least once, had the same number of correctly learnt actions.

A chi-square test of independence was performed in which the success rates of the different algorithms were compared. A post hoc test is performed to find significant differences between any of the nine algorithms. After correcting for multiple hypotheses using Bonferroni several significant interactions were obtained ($p = 0.021$ for all significant interactions). These significant interactions occur when one algorithm has a success rate of 100% and the other algorithm has a success rate of 0%.

Thus we conclude that Q-learning and SARSA in combination with the sigmoid are methods that perform poorly. All other methods are viable in the discretized environment.

Division of Work by Team Members

- Development of the 2D arcade game (80% Henry, 20% Jeroen)
- Discretizing the steps of the agent (100% Henry)
- Building the MLP (75% Henry, 25% Jeroen)
- Building Experience Replay (50% Henry, 50% Jeroen)

- Building Learning From Demonstration (50% Henry, 50% Jeroen)
- Constructing Q-Learning (80% Henry, 20% Jeroen)
- Constructing SARSA (75% Henry, 25% Jeroen)
- Constructing Supervised Learning (5% Henry, 95% Jeroen)
- Writing the abstract (40% Henry, 60% Jeroen)
- Writing the introduction (40% Henry, 60% Jeroen)
- Writing the methods (60% Henry, 40% Jeroen)
- Writing the results (40% Henry, 60% Jeroen)
- Writing the discussion (40% Henry, 60% Jeroen)
- Writing the conclusion (50% Henry, 50% Jeroen)

References

Sander Adam, Lucian Busoniu, and Robert Babuska. Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(2):201–212, 2012.

Christopher G Atkeson and Stefan Schaal. Learning tasks from a single demonstration. In *Robotics and Automation. Proceedings., IEEE International Conference on*, volume 2, pages 1706–1712, 1997a.

Christopher G Atkeson and Stefan Schaal. Robot learning from demonstration. In *International Conference on Machine Learning*, volume 97, pages 12–20, 1997b.

Luuk Bom, Ruud Henken, and Marco Wiering. Reinforcement learning to train MS. Pac-man using higher-order action-relative inputs. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 156–163, 2013.

Vivek S Borkar and Sean P Meyn. The ODE method for convergence of stochastic approximation and reinforcement learning. *SIAM Journal on Control and Optimization*, 38(2):447–469, 2000.

David S Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, DTIC Document, 1988.

Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *NIPS*, volume 13, pages 1008–1014, 1999.

Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques, 2007.

Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

Gavin A Rummery and Mahesan Niranjan. On-line Q-learning using connectionist systems. Technical report, 1994.

Amirhosein Shantia, Eric Begue, and Marco Wiering. Connectionist reinforcement learning for intelligent unit micro management in starcraft. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1794–1801. IEEE, 2011.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

John N Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16 (3):185–202, 1994.

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

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

Marco Wiering and Martijn Van Otterlo. *Reinforcement learning, State-of-the-Art*, volume 12. Springer, 2012.