Abstract: In this thesis the use of a vision grid is explored in order to train agents to play the arcade game Frogger, using Q-Learning combined with a Multiple Layer Perceptron. The game Frogger can be split up into two smaller tasks, the first being crossing the road and the second being crossing the river and reaching a goal. As these tasks are not connected, the possibility of creating an agent using two neural networks that each complete one task will be explored and compared to the performance of a single neural network. Furthermore, the use of single-action networks and learning from demonstration will also be explored. The results show that while the single-action networks and the two neural networks approaches are both able to complete the road section of the game with near perfect performance, none of the approaches were able to play the game on the level of a human. The single-action network agents were found to generalize better than the other agents on the road section. Learning from demonstration did not significantly improve performance for the two neural network or single-action network agents on the road section.

Keywords: machine learning, neural networks, reinforcement learning, Q-learning, learning from demonstration, video games

1 Introduction

Reinforcement learning (Sutton and Barto, 1998) has proven itself to be a viable approach for solving many problems in which an agent has to thrive in a large or dynamic environment. It has been successfully applied to board games such as backgammon (Tesauro, 1995) and more recently Go (Silver et al., 2016) which are too complex to be solved using more conventional strategies. In the past, strategies such as mini-max search have been successfully used to create agents that are able to play games with a large state space such as Chess on the level of a human expert (Buro, 2002), but for games with an even larger state space such as Go the approach becomes unfeasible due to high computation times. Reinforcement learning is better suited for these types of problems, as it does not depend on searching through a large amount of possible future states in order to decide on which action to take.

Q-learning (Watkins, 1989; Watkins and Dayan, 1992) combined with a neural network has been used to create agents that are on or even surpass human level of play in several video games (Bom et al., 2013; Mnih et al., 2013) and is thus an interesting strategy to explore for creating new agents that can play video games on a high level. Q-learning is a viable approach for creating agents in distinctly different video games as the technique is model-free and the way the state is presented to the network can therefore be constructed by the modeler in order to fit any specific game. We will discuss the workings of Q-learning in section 3.1.

In several earlier studies learning from demonstration (LfD) has led to successfully trained agents when applied to agents learning to play a video game (Ozkohem et al., 2017; Weber et al., 2012). The study of Ozkohem et al. (2017) in particular, where it was thought that the problem was too complex to be learned by conventional reinforcement learning, LfD proved to be a viable way of still being able to successfully train an agent. A similar level of complexity is also present in the game Frogger, as the agent needs to reach five goals that are all in a different location in order to win. Because of this, LfD might also be effective for training a Frogger agent.

1.1 Frogger

Frogger is a video game that was released by Konami in the year 1981. In Frogger, the player controls a frog that has to vertically cross a road and a river on which agents move horizontally. A visual representation of the game is given in figure 2.1.

Firstly, the player must cross the road while avoiding the agents (cars) that are driving there, as colliding with
a car means that the player loses a life. In the context of this game, losing a life means that all goals are reset and that the player has to start over from the starting position. When a goal is reset, the state of that goal will change to “open”, which means that the player has to reach that goal in order to win. When a player arrives at an open goal, that goal’s state changes to “closed”. The player should only move towards open goals. When the player moves into a closed goal; the player loses a life. Secondly, the player must cross the river. The rules are different here than they were during the road section. Now the player must keep the frog on top of the agents (logs and turtles), jumping from agent to agent in order not to drown. When the frog jumps into the water, or falls into the water due to any other reason the player loses a life and has to start over at the starting position. This means that the player will also need to cross the road again before getting a second try at crossing the river. In later levels crocodiles will also appear in the river and in the goals. The player should not collide with the crocodiles, as this causes the player to lose a life.

The player traditionally has three lives in which they have to reach all five goals located at the end of the level. After reaching all goals the player has completed the level and is able to start the next level in which a few new enemies appear and the speed of the agents is increased slightly. Our implementation differs from the traditional game in this respect, we will elaborate further on this in section 2.

In the original game, points are awarded for several actions. When the player moves north to a height they have not yet encountered, they are awarded 10 points. When the player reaches a goal, they are awarded 50 points, along with 10 points for every half second of time they have left. Reaching all 5 goals awards the player with 1,000 points. 200 bonus points can be earned by eating flies that randomly appear in the level or picking up a frog and guiding them home. These frogs appear randomly on logs in the river. At 20,000 points an additional life is awarded, and no additional lives are awarded after that. In our implementation the amount of points the agent obtains are recorded in order to judge performance, but are not used for any other purpose. The bonus life that is originally awarded is also not present in our implementation.

1.2 Reasons for study

Frogger is an interesting game to attempt to solve using reinforcement learning due to a number of factors. Firstly, the large number of agents on the screen at all times forces the player to plan ahead and think about future moves in addition to their current move. Secondly, the difference in failure criteria between the road- and river sections forces the player to adopt different strategies depending on their place in the world. Thirdly, there is a myriad of ways the player can lose a life. Finally the five distinct goals that need to be reached in order to beat the game make Frogger a challenging game to learn. It will be interesting to see if and in what way a trained agent is able to adapt to these factors. In this paper several approaches will be explored in both state-representation and different uses of neural networks to see whether or not it is possible to beat the game Frogger by using neural networks and Q-Learning.

1.3 Outline of thesis

Section 2 describes our implementation of the game Frogger, along with the differences between the implementation and the original. Section 3 discusses the theory behind the reinforcement learning techniques that have been used for the experiments. Section 4 describes the experiments that were performed. Section 5 describes the results that were obtained. Section 6 discusses the results in further detail, along with possible explanations for the results. Finally, section 7 presents the conclusions and discusses possible further research.

2 Implementation

We have written our own implementation of Frogger in order to have more control over how a representation of the environment (state representation) can be constructed from information present in the simulation. The
A graphical representation of the game can be seen in Figure 2.1.

In Figure 2.1 the player is situated at the bottom of the screen. Above it is the road, populated by cars. Above the road is the river, in which turtles and logs can be observed. Above the river the five goals can be seen. The goals second and third from the left have already been reached (i.e., they are in the closed state), as can be seen by the frogs that are inside them.

The game keeps track of the current state of all agents, including their speed, location and direction of movement.

On the road section most lanes are populated with cars that drive slowly, but are high in number, however, on one of the lanes, only two cars drive, but they move with a higher speed than the other cars.

On the river, both turtles and logs float across the screen. They move with the same speed, but on a set number of time steps the turtles dive under water for a few time steps. During this time, the player cannot stand on them. If the player is standing on them when they dive, the player will drown and lose a life.

At every time step all agents, including the frog, act. All agents except for the frog have simple and clearly defined behaviors; they all move with their own pre-defined speed from east to west or from west to east. The frog acts by feeding the current state representation to the neural network and executing the action that the neural network assigned the highest Q-value to. Several exploration strategies have been used in this project that will force the agent to take actions that do not have the highest Q-value. This will enable the agent to discover new strategies that might be better than the current dominant strategy.

Our implementation differs from the original game in several factors. The version we have written consists of only the first level of the original game and does not have the bonuses that occur at random. This was a deliberate choice, as random rewards might introduce noise into the measurements and therefore give a less accurate representation of the performance of the agent. Our version also differs from the original in the graphics that it uses, but this does not influence the agent as it does not perceive its environment through looking at the screen, but through features extracted from its environment.

The game calls its main update loop ten times per second; in the update loop all agents make their move and the image on the screen is refreshed. As all agents are allowed to move ten times per second a well-trained agent should be able to complete the level relatively quickly compared to a human player.

The agent only has one life per game. This is implemented by resetting the position of the Frog and the state of the five goals whenever the Frog loses a life. When this happens the locations of all other agents are purposefully not reset in order for the Frog to get a wide amount of different input for the same location in the game and thereby get a more robust network that can adapt to a large number of similar situations later in the game.

2.1 State representation

Vision grid

We used a vision grid (Shantia et al., 2011) centered on the agent in order to give the agent an idea of where other agents are located. The vision grid is filled with values ranging from minus one to one. Zero denotes an empty space, non-zero values denote spaces where (part of) an agent is located. Negative values denote agents moving west, positive values denote agents moving east.

As the state space of the simulation is continuous, the data gathered from the simulation needs to be discretized in order to be used as input for the neural net-
work. This discretization was done by distributing each agent over one or several nodes (spaces in the vision grid) where they were located at that time step. For example, if an agent that is moving east is two thirds in space A and one thirds in space B next to space A, space A would get an activation value of 0.66 and space B a value of 0.33. If the agent was moving west, the values would be multiplied by minus one. This way the direction in which the agent is moving is also represented in the input.

Each node in the vision grid represents a region of 30 by 30 pixels in the game world. As the grid is centered on the frog the spaces always refer to the same regions with respect to the frog. This size was chosen as the frog itself is also 30 by 30 pixels. This way, when the node to the north of the frog has a value of zero, and the nodes east and west of that node have a value below a certain threshold, the agent would be able to conclude that it is unlikely that an agent will be in that node in the coming time step and that it is therefore safe to move north.

The vision grid looks four steps to the north of the agent and two steps to the south, east and west of the agent, for a grid size of seven by five. Tests were conducted in which the vision grid was of a different size. The smaller grids were found to perform worse, whereas the larger grids did not improve much.

In addition to the vision grid the agent also gets other information from the game world. This information consists of:

- How much time has passed (normalized between zero and one);
- The x and y coordinates of the agent given as two values, x normalized between minus one and one, y normalized between zero and one;
- the x and y coordinates of the closest empty goal given in the same format as the coordinates of the agent;
- the x and y coordinates of the closest empty goal relative to the x and y coordinates of the agent;
- The remaining time before the turtles start diving given as a normalized value between zero and one;
- The remaining time that the turtles will be underwater and therefore aren’t safe to step on. Given as a normalized value between zero and one.

A visual representation of the input variables can be seen in figure 2.2. As there are five distinct goals that the agent has to reach, it is important that the agent knows which ones it has already reached. This has been implemented by giving the absolute goal position and the inverted relative position of the closest goal to the agent. The closer the agent comes to the goal, the higher these values become. This was chosen over normal relative goal positions after early experiments showed that inverted relative positions gave better results.

It is also important that the network knows how much time has passed, as the frog will automatically lose a life and therefore has to start over when 60 seconds have passed.

### High-level input vector

As an alternative to the vision grid, it was attempted to train the agent using a high-level input vector. This input vector consisted of a single vector of values. Each value represented the location of the closest agent in a row in the agent’s environment. The rationale behind
this approach was that when the amount of inputs would be significantly reduced, this would in turn decrease computation time and increase the value of each input.

In addition to the vector, the same additional inputs as used with the vision grid are presented to the network. Early during testing it became apparent that the vector was not able to perform at the same level as the vision grid and therefore the approach was scrapped before the final experiments began.

2.2 Neural networks

We have written a multi-layer perceptron (MLP) (Rumelhart et al., 1986) in order to combine Q-Learning with a neural network. The MLP uses a sigmoidal activation function for the neurons in hidden layers, and a linear activation function for the neurons in the output layer. By using a linear activation function in the output layer the network is not bound to output values between zero and one and is therefore able to learn Q-values in a wide numerical range.

The network consists of three layers; an input layer, a single hidden layer of 75 neurons, and an output layer of variable length.

The input of the network consists of all inputs described in the state representation; for every node in the vision grid there is one neuron in the input layer in which its value is stored. The length of the output layer depends on the type of network that needs to be implemented.

The network is initialized with the bias value and weights of every neuron initialized randomly between -0.3 and 0.3.

Multi-action networks

The multi-action networks that will be used in the experiments have five neurons in their output layer. Each output neuron corresponds to one of the five actions the agent can take at every time step. The multi-action network will have to approximate the Q-values of all five actions given the input-representation of a state.

Single-action networks

As an alternative to a multi-action network the use of single-action networks will be tested. Single-action networks only have a single neuron in their output layer and give an approximation of the Q-value of a single state-action pair. Because each network only returns the output for a single state-action pair, five single-action networks are needed for every one multi-action network that was used in order to get a Q-value for every possible action in the current state.

Experiments will be run using a single multi-action network for the entire game, using two multi-action networks, one for the road and one for the river, and using 10 single-action networks, 5 for each output for the road section, and 5 for the river.

3 Reinforcement learning

Reinforcement learning (Sutton and Barto, 1998) is a machine learning strategy in which an agent learns how to map (sensory) input to actions in order to maximize a numerical reward signal for solving a certain problem. This is done by letting the agent interact with the environment for a set amount of time steps and giving it numerical rewards according to its performance. For example, in a video game the reward given could be of a positive value when the agent wins the game and of a negative value when the agent loses the game.

Each time step the agent receives input from the environment in the form of a state representation. It then has to decide on an action. What action it takes depends on the exploration strategy it follows. The exploration strategies used for the experiments in this paper will be elaborated on in section 3.4. Exploration strategies often force the agent to make a sub-optimal move according to some level of chance. This way, when the agent gets stuck in a sub-optimal strategy it is still able to break out of it and discover that an action it assigned a lower utility earlier now is better than its current strategy.

3.1 Q-Learning

Q-learning (Watkins, 1989; Watkins and Dayan, 1992) is a reinforcement learning method in which every state-action pair is given a value (Q-value) that indicates its utility. The Q-values of states are often initialized either at random or uniform and are then, by using gradient descent, updated until the values are representative enough of their state-action pairs (for example by looking at the rate of change of the Q-values).

Q-learning is originally implemented by constructing a large table of Q-values for state-action pairs and at every time step updating one or several Q-values in the table according to (some variation of) equation 3.1.
In Equation 3.1, $s$ is the state the agent is currently in, $a$ is an action that the agent took in state $s$, $s'$ is the state the agent will be in after taking action $a$ in state $s$, and $a'$ is the action with the highest utility in state $s'$. $R$ is the immediate reward received from taking action $a$ in state $s$. The final parameter, $\gamma$ is a factor governing how heavily the algorithm should weigh the future rewards; often this factor is called the discount factor.

For problems with a small state space and well-defined spaces a table works efficiently, but for problems with a relatively large or continuous pace a tabular approach can be inefficient, as the table needs to contain every possible state-action pair. For these large state spaces the table can become unusably large, leading to high memory usage. For continuous state spaces, the states that are observed are almost never identical, making it unlikely that the exact same state is observed twice, let alone often enough for the algorithm to converge to representative and usable Q-values.

There is a solution for both of these problems. By combining Q-learning with a neural network such as an MLP, table-lookup can be approximated using a fraction of the memory otherwise required, so a neural network can be a viable alternative when dealing with a large state space. By the design of neural networks one can also assume that the network is able to generalize well, i.e. two state-action pairs that look similar will also lead to similar state-action values. This ability to generalize makes it possible for the network to converge to representative Q-values for a vast amount of state-action pairs, even for states which it has not encountered yet but are sufficiently similar to states that it encountered during training. This means that a neural network should be able to work in a continuous space.

To update the Q-values in a neural network, the error between the output of the network and the desired Q-value is back-propagated through the network. The desired Q-value of a state-action pair is calculated by altering the original equation for calculating Q-values. The equation used in non-terminal states can be found in Equation 3.2.

$$Q(s,a) = (1 - \alpha)Q(s,a) + \alpha[R + \gamma \max_a Q(s',a')]$$ (3.1)

$$Q_{target}(s,a) = R$$ (3.3)

The agent learns these target Q-values by taking the difference between the target Q-value and the Q-value given by the neural network, and then back-propagating this difference through the neural network as error. As this is done separately for every action the agent takes, it can be said that the agent uses online learning.

### 3.2 Learning from demonstration

As during initial tests an agent trained purely with reinforcement learning did not perform well on the river section of the game the use of learning from demonstration will also be explored for training an agent. It is possible that the task of reaching several goals in order to win the game is too complex for the agent to learn on its own, but it might be accomplished when a human player first plays the game in order to demonstrate how the agent should act in certain circumstances.

In learning from demonstration, an actor skilled in the game or task to be learned (in this case the author) first performs the task a number of times, while the program records all the state-action pairs and rewards that occur during the demonstration. Using this data, the agent can then be trained for a number of cycles. After this initial training session, the agent can further fine-tune its parameters by employing regular trial-and-error based reinforcement learning.

Learning from demonstration has been successfully used for creating agents that can play video games on a high level (Ozkohen et al., 2017; Weber et al., 2012) and for teaching robots complex movements (Atkeson and Schaal, 1997)

For this project, the author has played the game for approximately one hour while the game saved every state-action pair and reward that occurred during that time. After demonstrating, the data was refined by removing approximately half of all state-action pairs in which the action was to do nothing. This was done as a human player is too slow to make an action ten times per second (which is the frequency at which the program expects an action), which led to a large number of state-action pairs in which the action is to do nothing. By removing part of these pairs the data has less redundancy and can be used to train an agent in a shorter
amount of time.

The agent can be trained with this data by iteratively feeding the recorded state representations into the neural networks and back-propagating the difference between target Q-value and Q-value given by the network through the network for the recorded action.

3.3 Reward function

When the implementation was first written, an action-based reward function was constructed. In this function the agent gets a reward for performing specific actions. The rationale behind this reward function was to incentivize the agent to go up towards the goals, while not giving large enough rewards such that the agent will take unnecessary risks or get stuck because it doesn’t want to take an action with a negative reward. The used reward values for each action can be found in Table 3.1. The rewards given for certain special states can be found in Table 3.2. In Table 3.2, ”reached the line” is a reward that is given for reaching the purple line dividing the road and the river. This was implemented to give the agent a clear goal for the first section of the task (crossing the road). The reward can only be awarded once per life in order to avoid exploitation by repeatedly trying to obtain the reward. Furthermore, all terminal states are exclusive, so compound rewards cannot be given. This way there can be no positive reward for both dying and reaching a goal during the same time step. No additional reward is given for reaching all five goals. The reward for reaching a single goal should suffice for the agent to learn an optimal strategy. If a bigger reward would be given for reaching all five goals the agent might favor the last goal it reached over other goals as it yielded a higher reward than the other goals. Furthermore, by giving a reward for reaching a goal, a strategy that reaches all goals should still emerge as this would still yield the highest total reward for the agent.

<table>
<thead>
<tr>
<th>Movement</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>up</td>
<td>1.0</td>
</tr>
<tr>
<td>down</td>
<td>-1.0</td>
</tr>
<tr>
<td>left</td>
<td>0.0</td>
</tr>
<tr>
<td>right</td>
<td>0.0</td>
</tr>
<tr>
<td>stand still</td>
<td>-0.5</td>
</tr>
</tbody>
</table>

Table 3.1: Reward table for actions

As an alternative to the action-based rewards, a distance-based reward function was written. For this reward function the agent gets a negative reward ($R$) at every time step, the severity of which is based on the Euclidean distance between the agent and the nearest goal. The rewards for special states described in Table 3.2 are still awarded. It is implemented using equation 3.4, in which $P$ denotes the penalty. For these experiments the parameter $P = -5.0$ will be used. The x and y values are extracted from the simulation as the coordinates of the center of the agent and the center of the closest open goal.

$$R = P \times \sqrt{(goal_x - agent_x)^2 + (goal_y - agent_y)^2} \quad (3.4)$$

Other parameters ($P = -10$ and $P = -2$) were also explored, but a value of $P = -5$ was found to perform more efficient compared to the others.

During early tests it quickly became apparent that the distance-based reward function vastly outperformed the action-based reward function in both execution time of the agent and the final score achieved.

3.4 Exploration strategies

Several exploration strategies have been implemented for use in the experiments:

**Greedy** always chooses the action with the highest Q-value over all state-action pairs in its current state. As it always chooses the action it deems best, it doesn’t explore any state that currently has a low Q-value and is therefore prone to getting stuck in a suboptimal dominant strategy.

**ε-greedy** chooses the action with the highest Q-value 1 – ε percent of the time, otherwise it chooses a random action. ε-greedy is a good trade off between the greedy exploitation of a working strategy the agent has found and the random exploration that gets the agent out of local maxima.

**Diminishing ε-greedy** works similar to regular ε-greedy, but lowers the value of ε as time goes on. The implementation used for the experiments uses the current generation of the simulation and the total amount
of generations in order to determine the value for $\epsilon$. The equation can be found in equation 3.5.

$$\epsilon = \max(\epsilon_{\text{base}} \ast (1.0 - \frac{\text{currentGen}}{\text{totalGen}}), \epsilon_{\text{min}})$$

Use of an exploration strategy that incorporates some form of randomness during the agent’s training phase is important, as agents trained using greedy reinforcement learning are prone to getting stuck in strategies that cause it to not perform well. Occasionally making a random move can cause the agent to break out of these strategies and discover a strategy superior to what it was originally using.

4 Experiments

For all experiments a diminishing learning rate will be used. The learning rate will start with a base value of 0.0001 and will diminish according to equation 4.1.

$$LR = LR_{\text{base}} \ast (1.0 - \frac{\text{currentGen}}{\text{totalGen}})$$

As currentGen runs from 0 to totalGen - 1, the learning rate becomes very small, but will never reach zero. This was implemented as learning with a constant LR of 0.0001 or higher tended to overtrain after approximately twenty epochs, even though initial tests showed the network could still improve with a lower learning rate. The rewards used are given in section 3.3. For all experiments the distance-based reward function will be used. Diminishing $\epsilon$-greedy will be used as the exploration strategy for all experiments with an $\epsilon_{\text{base}}$ of 0.2 and an $\epsilon_{\text{min}}$ of 0.05.

The experiments are conducted by initializing the neural networks and using online training to train them for 100 generations. Each generation consists of a training phase of 10,000 games and two testing phases of 1,000 games. The level in the first testing phase will be the same as the one the agent trained on. The second testing phase will have all agents going in their opposite direction in order to test whether or not the behavior the agent learned generalizes well. During the training phase the difference between the Q-value of the chosen action and the desired Q-value of that state-action pair (this difference will be called the error) is back-propagated through the network. During the training phase the agent will make use of an exploration strategy.

During the testing phase the agent does not explore (i.e. it uses greedy as its exploration strategy) and does not back-propagate errors through the neural network.

Ten simulations will be run per strategy. The final results will be obtained by averaging the results of the simulations. Results will be collected on the win percentage, the road-completion percentage, and how many points the agent would have scored if this was the original game of Frogger.

Six experiments will be conducted in total, three without using demonstration learning and three using demonstration learning.

These six experiments can be divided into three groups: one group will train agents using one MLP, one group will train agents using two MLPs, and the final group will train agents using ten single-action networks. In these groups one experiment will be conducted in which the agent uses pure trial-and-error based reinforcement learning in order to learn to play Frogger, and one experiment will be conducted by first pre-training the network for 50 cycles on the data collected during the demonstration and then letting the agents refine their strategies using trial-and-error based reinforcement learning.

The road completion percentage will be calculated by dividing the number of games where the agent crossed the road by the total amount of games played. The win percentage is calculated in a slightly different manner. As there are five goals the agent has to reach in order to win the game and the agent had trouble even reaching a single goal during preliminary experiments, the win percentage for each game is calculated as the percentage of goals reached. This way, if the agents would never reach all five goals, but would consistently reach two out of five goals, this could still be observed.

The points are recorded mainly to see if the agent improves without having to rely on the win percentages as these have been very low in preliminary experiments and might not reflect improvements well.

In the experiments the turtles on the river do not dive. This was put in place as a simplification in order to improve the agent’s performance on the river. If the agent was able to perform well without the turtles diving further experiments could be done where the turtles do dive, but if the agents already perform poorly without the diving the diving will only make the results worse, i.e. it could be concluded that the agent is unable to win the game without having the turtles dive, but it could not be concluded that the agents perform well on the river.
5 Results

For all experiments, figures are created showing the performance of the agents on the test games as the amount of generations progresses. Results on road-completion, win percentage and points gained will be shown in these figures.

The results obtained from the simulations that do not use LfD are shown in figure A.1; the results obtained by using LfD can be found in figure A.2. In both figures graphs can be seen, showing the road completion percentage, win percentage and amount of points gained. In figures A.3 and A.4 the results for the generalization tests of the non-LfD and the LfD experiments respectively can be found.

In table 5.1 the performance of all the fully-trained networks can be found. The table labelled normal holds the results obtained during the non-LfD experiments. As can be seen in table 5.1 and figures A.1 and A.2 none of the tried approaches perform even close to the level of a human expert, the highest win percentage being 0.07 achieved by using two MLPs and no demonstration learning. This score indicates that the agent was able to reach one of the five goals approximately once every three lives. As the road completion percentage is 0.97 this means that most agent deaths must have occurred on the river. The results do not show where exactly the agent died.

The single MLP approach generally performed the worst, having both the lowest road completion rate, win percentage and gained points in both the non-LfD experiment and the LfD experiment.

5.1 Analysis of results

To test whether there are significant differences between the obtained results, Wilcoxon signed-rank tests will be used. Wilcoxon tests were chosen over the more conventional t-tests as all of the data obtained is not normally distributed and Wilcoxon tests are non-parametric.

5.1.1 Differences between the three types of networks used

The single-action network agents performed about equal to the two-MLP agents with regard to road-completion rate ($W(10) = 49.5, p = 1.0$), but underperformed when looking at the points gained ($W(10) = 96, p = 5.8e-04$) and the win percentage ($W(10) = 98, p = 3.3e-04$).

The one-MLP agent performed significantly worse that the two-MLP agent, considering both road completion ($W(10) = 100, p = 1.8e-04$), points gained ($W(10) = 100, p = 1.8e-04$) and win percentage ($W(10) = 100, p = 6.3e-05$).

5.1.2 Differences between non-LfD and LfD

Learning from demonstration did not significantly change the performance for the one MLP agents with regard to the road completion percentage ($W(10) = 32, p = 0.19$) and points gained ($W(10) = 33, p = 0.21$). For the overall win percentage a statistic could not be derived, as both types of agent never managed to reach the goal. The performance of the two MLP agents did not change when pre-training the networks using LfD with regard to road completion ($W(10) = 57.5, p = 0.60$), but slightly decreased in performance when looking at the points gained ($W(10) = 82, p = 0.017$) and win rate ($W(10) = 82, p = 0.017$). For the single-action network agents, the performance did not change with regard to the road completion percentage ($W(10) = 32.5, p = 0.19$), points gained ($W(10) = 61, p = 0.43$) or win percentage ($W(10) = 56.5, p = 0.65$).

5.1.3 Results on generalization

When looking at the generalization results for non-LfD experiments in table 5.1 one can see that the road-completion percentages for the two-MLP agents went down from 0.98 to 0.79. While this is still a decent performance, it did go down by almost 20 percent. This was not the case for the single-action network agents, which only went down from 0.98 to 0.93. This means that even though the single-action network agents did not outperform the two-MLP agents during the original experiments, they do generalize better, which is especially important if the trained agents would have to play the entire game and not just the first level.

The points gained decreased for most agents, the exception being the one-MLP agents, whose score increased the amount it earned by around ten points. This seems to indicate that it performed better during the generalization tests ($W(10) = 13, p = 5.83-03$).

The win percentages were already very low in all experiments, so it is not surprising that the agents never managed to reach the goal during the generalization experiments.
### Table 5.1: Results of all experiments performed. Road completion and Win percentage are given as values between zero and one. Points gained are the points the agent would have scored if this would have been the original game of Frogger. The row labeled "one" denotes the results of the one-MLP experiments; "two" denotes the two-MLP experiments, and "SA" denotes the single-action network experiments. All values are computed by taking the average of the results after 100 epochs of training.

<table>
<thead>
<tr>
<th></th>
<th>Road completion</th>
<th>Points gained</th>
<th>Win percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Results normal</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>0.40 SD: 0.18</td>
<td>37 SD: 8</td>
<td>0.00 SD: 0.00</td>
</tr>
<tr>
<td>two</td>
<td>0.98 SD: 0.02</td>
<td>243 SD: 58</td>
<td>0.03 SD: 0.01</td>
</tr>
<tr>
<td>SA</td>
<td>0.98 SD: 0.02</td>
<td>123 SD: 44</td>
<td>0.01 SD: 0.01</td>
</tr>
<tr>
<td><strong>Results LfD</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>0.51 SD: 0.15</td>
<td>41 SD: 7</td>
<td>0.00 SD: 0.00</td>
</tr>
<tr>
<td>two</td>
<td>0.98 SD: 0.02</td>
<td>139 SD: 86</td>
<td>0.01 SD: 0.01</td>
</tr>
<tr>
<td>SA</td>
<td>0.99 SD: 0.02</td>
<td>106 SD: 20</td>
<td>0.01 SD: 0.00</td>
</tr>
<tr>
<td><strong>Results normal generalization</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>0.37 SD: 0.12</td>
<td>50 SD: 7</td>
<td>0.00 SD: 0.00</td>
</tr>
<tr>
<td>two</td>
<td>0.79 SD: 0.07</td>
<td>50 SD: 18</td>
<td>0.00 SD: 0.00</td>
</tr>
<tr>
<td>SA</td>
<td>0.93 SD: 0.06</td>
<td>49 SD: 5</td>
<td>0.00 SD: 0.00</td>
</tr>
<tr>
<td><strong>Results LfD generalization</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>0.36 SD: 0.11</td>
<td>50 SD: 5</td>
<td>0.00 SD: 0.00</td>
</tr>
<tr>
<td>two</td>
<td>0.88 SD: 0.09</td>
<td>55 SD: 4</td>
<td>0.00 SD: 0.00</td>
</tr>
<tr>
<td>SA</td>
<td>0.85 SD: 0.11</td>
<td>48 SD: 8</td>
<td>0.00 SD: 0.00</td>
</tr>
</tbody>
</table>

### 6 Discussion

Both the two-MLP agents and the single-action network agents managed to get a high road-completion percentage that almost reaches perfect play. The one-MLP agents performed the worst in all experiments. The single-action network agents performed on par with the two-MLP agents, but were able to generalize their behavior better, judging from their road-completion percentage.

The two-MLP and single action agents were both able to cross the road nearly every time, leading to a road completion percentage close to a hundred percent. This proves that the experiment set-up is viable. It remains unclear why the same approach did not work on crossing the river. The inverted relative location of the closest goal and the relative position of the closest goal were used as input to the network to give the agent an idea of where the closest goal was located, as this seemed to work best in preliminary experiments. It might be the case that the networks failed to link this input with the location of the goals, which would have led the agents to not know where the goals were. It might also be the case that the crossing of the river itself was too hard for the agents to learn.

It is interesting that the single-action network agents were able to generalize the behavior they learned on the road such that it led to similar results during the generalization test. This was also observed in the study by Bom et al. (2013) in which single-action network agents were able to generalize their behavior to an unknown maze better than multiple-action network agents. This seems to indicate that single-action networks generalize better than multi-action networks in all 2D video games.

None of the trained agents were able to successfully play the game even close to human level; the highest average win percentage being 0.07 reached by the two-MLP agents.

### 7 Conclusion and future research

To summarize, for this project agents have been trained to play the game Frogger using Q-learning combined with neural networks. A vision grid along with several selected features were used as the state representation. Three different approaches were taken to designing the networks: a single neural network; two neural networks and several single-action networks. Overall, the single neural network performed the worst and
proved to not be viable for this problem. The two neural networks performed better than the single neural network, but were still not able to cross the river and reach all open goals. Finally, the single-action network agents performed similar to the two neural networks, but were found to generalize their behavior better.

Experiments using learning from demonstration to pre-train the networks were also performed in an attempt to train a successful agent. Learning from demonstration did not improve the performance of the two network and single-action network agents, but did improve the performance of the single network agent slightly.

We conclude that the approaches presented in this paper to train an agent that can successfully play the game Frogger did not succeed. Even though promising results were obtained for the road by the two MLP and single-action network agents, none of the trained agents were able to play the complete game on the level of a human.

7.1 Future research

Looking back on the project it would have been interesting to record where exactly the agent had died in the world. This would have given insight as to whether the agent was able to cross the river but was unable to locate a proper goal and died by jumping in between goals or in an already filled goal or that it died somewhere on the river. Another thing that would have been worth testing is whether or not the agent is able to find the goals when the river is replaced with a second road. When looking at the road-completion percentages in table 5.1 it can be seen that the two-MLP and single-action network agents are very capable of crossing the road, so creating an agent that is able to cross two roads should be fairly straightforward; the only difference being that it would have to reach a goal at the end of the second road. This would give some insight as to whether the input representation of the goals are good enough for the agent to successfully reach all open goals.

It is a bit disappointing that the agents were all unable to perform on human level throughout the entire game. The performance of the single MLP on the entire game indicates that a single one-layer MLP can never get close to human level play. The performance of the two-MLP and single-action network agents on the road indicates that human-level play can be reached using this input and neural network architecture. Therefore it seems likely that the problem of crossing the river and reaching a goal is too hard to be solved using the used framework. It might be interesting to see whether a successful agent can be obtained by training using a convolutional neural network that takes all pixel values of the world as its input, such as was done and led to successful results for similar video games in earlier studies (Mnih et al., 2013, 2015).

8 Acknowledgements

The author would like to thank Jip Maijers for helping write a significant part of the code used in the experiments.

References


A Figures

Figure A.1: Results obtained from the experiment in which agents were trained without the use of LfD. All graphs show the averaged results for all simulations over the number of generations the agents have been trained for.

Figure A.2: Results obtained from the experiment in which agents were pre-trained using LfD.
Figure A.3: Generalization results obtained without pre-training using LfD

Figure A.4: Generalization results obtained by pre-training the agents using LfD