university of groningen / faculty of mathematics and natural sciences / artificial intelligence

# Learning to Play Donkey Kong Using Neural Networks and Reinforcement Learning

Bachelor's Project Thesis

Paul Ozkohen, s2575973, p.m.ozkohen@student.rug.nl,
Jelle Visser, s2238160, jellevisser1994@hotmail.com,
Supervisor: Dr. M. Wiering

**Abstract:** Neural Networks and Reinforcement Learning have successfully been applied to various games, such as Ms. Pacman and Go. This research combines Multilayer Perceptrons and a class of Reinforcement Learning algorithms called Actor-Critic to make an agent learn to play the arcade classic Donkey Kong game. Two neural networks are used in this study, the Actor and the Critic. The Actor neural network learns to select the best action given the game state, the Critic tries to learn the value of being in a certain state. First, a base game-playing performance is obtained by making the agent learn from demonstration data, which is obtained from humans playing the game. After the off-line training of the Actor on the demonstration data, the agent tries to further improve its base performance using feedback from the Critic. The Critic gives feedback by comparing the value of the state before and after taking the action. Results show that an actor pre-trained on demonstration data is able to achieve a good baseline performance. Applying Actor-Critic methods, however, does usually not improve performance, in many cases even decreasing it. Possible reasons include the game not fully being Markovian and other difficulties.

## 1 Introduction

Games have become a subject of interest for machine learning in the last few decades. Playing games is an activity enjoyed exclusively by humans, which is why studying them in the pursuit of Artificial Intelligence is very enticing. Building software agents that perform well in an area that requires human-level intelligence would thus be one step closer to creating Strong, General Artificial Intelligence, which can be considered one of the primary goals of the entire field.

Reinforcement learning techniques have often been used to achieve success in creating game-playing agents (Silver et al., 2016; Mnih et al., 2015). Reinforcement learning requires the use of certain functions, such as a policy function that maps states to actions and a value function that maps states to values. The values of these functions could, for example, be stored in tables. However, most non-trivial environments have a large state space, particularly games where the states are often continuous. Unfortunately, tables would have to become enormous in order to store all the necessary function information. To solve this problem, function approximation is often applied. Consequently, general function approximators such as Neural Networks are often used in conjunction with Reinforcement Learning. A famous recent example of this is the ancient board game Go, in which Deep Mind's AI AlphaGo was able to beat the world's best players at their own game (Silver et al., 2016). Besides traditional games, the combination of neural nets and reinforcement learning can be used to learn to play video games as well. For example, Deep Mind used a combination of con-

volutional neural networks and Q-learning to achieve good gameplay performance at 49 different Atari games, and was able to achieve human-level performance on 29 of them (Mnih et al., 2015). This study shows how a Reinforcement Learning agent can be trained purely on the raw pixel image. The upside of the research showed that a good game-playing performance could be obtained without handcrafting game-specific features. The Deep Q-Network was able to play the different games without any alterations to the architecture of the network or the learning algorithms. However, the downside is that deep convolutional networks require exceptional amounts of computing power and time. Furthermore, one could speculate how well performance of each individual game could be improved by incorporating at least some game-relevant features. Still, it is impressive how the network could be generalized to very different games.

Alternatively, another approach is to instead use hand-crafted features specific to the game itself. One such game where this was successfully applied is Ms. Pac-Man, where an AI was trained to achieve high win rates at the game using higher-order, game-specific features (Bom et al., 2013). This approach shows that good performance can be obtained with a small amount of inputs, therefore severely reducing computation time.

In this thesis, we present an approach to machine learning in games that is more in line with the second given example. We apply reinforcement learning methods to a video game based on Donkey Kong, an old arcade game that was released in 1981 by Nintendo. The game features a big ape called Donkey Kong, who captures princess Pauline and keeps her hostage at the end of each stage. It is up to the hero called Jumpman, nowadays better known as Mario, to climb all the way to the end of the level to rescue this damsel in distress. Besides climbing ladders, the player also has to dodge incoming barrels being thrown by Donkey Kong, which sometimes roll down said ladders.

This game provides an interesting setting for studying reinforcement learning. Unlike other games, Donkey Kong does not require expert strategies in order to get a decent score and/or get to the end of the level. Instead, timing is of the utmost importance for surviving. One careless action can immediately lead Mario to certain death. The game also incorporates unpredictability, since barrels often roll down ladders in an unpredictable way. The intriguing part of studying this game is to see whether reinforcement learning can deal with such an unpredictable and timing-based continuous environment.

For this study, we used a specific reinforcement learning technique called Actor-Critic (Sutton and Barto, 1998). In each in-game step, the Actor (player) tries to select the optimal action to take given a game state, while the Critic tries to estimate the given state's value. Using these state-value estimates, the Critic gives feedback to the Actor, which should improve the agent's performance while playing the game. A special form of Actor-Critic was used: the Actor-Critic Learning Automaton (ACLA) (Wiering and Van Hasselt, 2007).

Both the Actor and the Critic are implemented in the form of a Multilayer Perceptron (MLP). Initializing the online learning with an untrained MLP would be near-impossible: the game environment is too complex and chaotic for random actions to lead to good behavior (and positive rewards). In order to avoid this, both the Actor and the Critic are trained offline on demonstration data, which is collected from demonstration games being played by human players.

The main question this study seeks to answer is: is a combination of neural networks and Actor-Critic methods able to achieve good gameplay performance in the game Donkey Kong?

## 2 Game Implementation

A framework was developed that allows the user to test several reinforcement learning techniques on a game similar to the original Donkey Kong. The game itself can be seen in Figure 2.1 and was recreated from scratch as a Java application.

The goal of the game is to let the player reach the princess at the top of the level. The agent starts in the lower-left corner and has to climb ladders in order to ascend to higher platforms. In the top-left corner, we find the game's antagonist Donkey Kong, which throws barrels at set intervals. The barrels roll down the platforms and fall down when reaching the end, until they disappear in the lower-left of the screen. When passing a ladder, each barrel has a 50% chance of rolling down the ladder, which adds a degree of unpredictability to the barrel's course. The agent touching a barrel results in an instant loss, while jumping over them nets a small score. Additionally, two powerups (hammers) can be picked up by the agent when he collides with them by either a walking or jumping action, which results in the barrels temporarily being destroyed upon contact with the agent, netting a small score gain as well. The agent can execute one out of seven actions: walking (left or right), climbing (up or down), jumping (left or right) or doing nothing (standing still). The game takes place in $680 \times 580$ window. Mario moves to the left and right at a speed of 1.5 pixels, while Mario climbs at a speed of 1.8 pixels. A jump carries Mario forward around 10 pixels. Therefore, this means that Mario needs to take a lot of actions to reach the princess from his initial position.

The game contains several different settings that determine how the player is controlled. There is a manual mode, which allows a human player to play the game using keyboard controls. If done in combination with the demonstration mode, the game automatically writes information about the game state for each time step to a file, which can be used for learning from demonstration (section 4). Additionally, there is an automatic mode, which can be executed in two ways: either the agent is controlled by an MLP trained on demonstration data without receiving any input ("Actor-only mode"), or the agent is controlled by the MLP while receiving feedback from another MLP ("Actor-Critic mode"). At each time step in the automatic mode, the MLP takes input from the game state and produces one out of the seven possible actions as the output. When active, the Critic provides feedback to the agent.

While this implementation of the game is quite close to the original game, there are several differences between the two versions of the game:

- The game speed of the original is slower than in the recreation.
- The barrels are larger in the original. To reduce the difficulty of our game, we made the barrels smaller.
- The original game contains an oil drum in the lower-left corner which can be ignited by a certain type of barrel. Upon ignition, the barrel produces a flame that chases the player. This has been entirely left out in the recreation.
- The original game consists of several different levels. The recreation only consist of one level, which is a copy of the first level from the original.
- The original game uses a relatively complex algorithm for determining whether a barrel will go down a ladder or not, which appears to be based on the player's position relative to the barrel and the player's direction. The code of the original is not available, so instead the barrels' odds of rolling down a ladder is set to be simply 50% at any given time.

While there are a few notable differences between the original and its recreation, both versions are still quite similar. It is therefore reasonable to assume that any AI-behavior in the recreation would translate to the original as well.

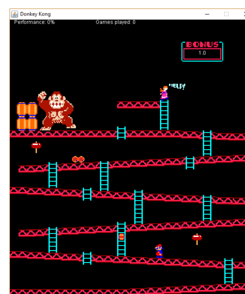


**Figure 2.1: Recreation of Donkey Kong.**

# 3 Multilayer Perceptrons

The Actor and Critic are implemented in the form of an MLP, a simple feed-forward network consisting of an input layer, one or more hidden layers and an output layer. Like the game itself, the MLP was built from scratch in Java, meaning no external packages were used. In this section, we look at the way the MLP used for this study is built up.

## 3.1 MLP input

The inputs for the neural net are derived from the game state using several different input algorithms. This section provides an overview of how these algorithms are implemented. The first two algorithms are several varieties of grids that are used to track the location of objects in the game. Each cell in each grid corresponds to one input for the MLP. Besides these grids, several additional inputs provide information about the current state of the game.

### 3.1.1 Vision Grids

There are three types of objects in the game that the agent can interact with: barrels, powerups and ladders. We use three different vision grids that keep track of the presence of these objects in the immediate surroundings of the agent. A similar method was used by Shantia et al. (2011) for the game Starcraft.

First of all, the MLP needs to know how to avoid getting killed by barrels, meaning it needs to know where these barrels are in relation to the agent. Barrels that are far away pose no immediate threat. This changes when a barrel is on the same platform level as the agent: at this point, the agent needs to find a way to avoid a collision with this barrel. Generally, this means trying to jump over it. Barrels on the platform level above the agent need to be considered as well, as they could either roll down a ladder or fall down the end of the platform level, after which they become an immediate threat to the agent.

The second type of objects, ladders, are the only way the agent can climb to a higher platform level, which is required in order to reach the goal. The MLP therefore needs to know if there are any ladders nearby and where they are.

Finally, the powerups provide the agent the ability to temporarily destroy the barrels, making the agent invincible for a short amount of time. The powerups greatly increase the odds of survival, meaning it's important that the MLP knows where they are relative to the player.

In order to track these objects, we use a set of three grids of $7 \times 7$ cells, where each grid is responsible for tracking one object type. The grids are fixed on the agent, meaning they move in unison. During every time step, each cell detects whether it's colliding with the relevant object. Cells that contain an object are set to 1.0, while those that do not are set to 0.0. This results in a set of $3 \cdot 49 = 147$ boolean inputs.

The princess is always above the player, while barrels that are below the player pose no threat whatsoever. We are therefore not interested in what happens in the platform levels *below* the agent, since there rarely is a reason to move downwards. Because of this, the vision grids are not centered around the agent. Instead, five of the seven rows are above the agent while there is only one row below.

A visualization of the vision grid can be seen in Figure 3.1.

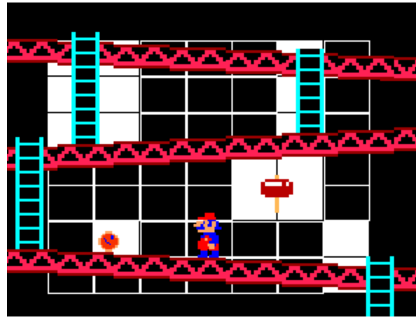

**Figure 3.1: Visualization of the vision grid that tracks objects directly around the agent, granting Mario local vision of its immediate surroundings. Note that while only one grid can be distinguished, there are actually three vision grids stacked on top of each other, one for each object type.**

### 3.1.2 Agent Tracking Grid

The MLP requires knowledge of the location of the agent in the environment. This way it can relate outputs (i.e. player actions) to certain locations in the map. Additionally, this knowledge is essential for estimating future rewards by the Critic, which will be explained further in section 5.

The agent's location in the game is tracked using a $20 \times 20$ grid that spans the entire game environment. Like the vision grid, each cell in the agent tracking grid provides one boolean input. The value of a cell is 1.0 if the agent overlaps with it, 0.0 if it does not. This grid provides $20 \cdot 20 = 400$ boolean inputs. A visualization of the agent tracking grid can be seen in Figure 3.2.

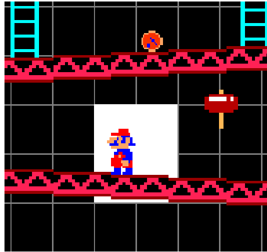

**Figure 3.2: Visualization of the level-wide grid that tracks the current location of the agent. While not visible in this image, the grid spans the entire game environment**

### 3.1.3 Additional Inputs

Four additional inputs are extracted from the game state. These are as follows:

- A boolean that tracks whether the agent can climb (i.e. is standing close enough to a ladder). This prevents the agent from trying to climb while this is not possible.
- A boolean that tracks whether the agent is currently climbing. This prevents the agent from trying to do any other action besides climbing while on a ladder.
- A boolean that tracks whether the agent currently has an activated powerup. This is used to teach the MLP that it can destroy barrels while under the influence of a powerup, as opposed to having to jump over them.
- A real decimal number in the range [0,1] that tracks how much time a powerup has been active. Given the time passed since the powerup was obtained (t) and the total time a powerup remains active (d), the time that the current powerup has been active is represented as the following fraction: TimeLeft = t / d.

The total amount of inputs is the sum of 147 vision grid cells, 400 agent tracking grid cells and 4 additional inputs, resulting in 551 inputs in total.

## 3.2 MLP output

### 3.2.1 Actor

The output layer consists of seven neurons, each neuron representing one of the seven possible player actions: moving left or right, jumping left or right, climbing up or down, or standing still. During training using demonstration data, the target pattern is encoded as a one-hot vector: the target for the output neuron corresponding to the action taken has a value of 1.0, while all other targets are set to 0.0.

During gameplay, the MLP picks an action based on softmax action selection (Sutton and Barto, 1998). In this method, each action is given a probability based on its activation. Using a Boltzmann distribution, we can transform a vector $\boldsymbol{a}$ of length $n$, consisting of real output activation values, into a vector $\boldsymbol{\sigma}(\boldsymbol{a})$ consisting of $n$ real values in the range [0,1]. The probability for a single output neuron $i$ is calculated as follows:

$$\sigma(a_i) = \frac{e^{a_i/\tau}}{\sum_{j=1}^{n} e^{a_j/\tau}} \quad \text{for i = 1, ..., n} \tag{3.1}$$

Here, $\tau$ is a positive real temperature value, which can be used to induce exploration into action selection. For $\tau \to \infty$, all actions wil be assigned an equal probability, while for $\tau \to 0$ the action selection becomes purely greedy.

During each in-game timestep, each output neuron in the Actor-MLP is assigned a value using equation 3.1. This value stands for the probability that the Actor will choose a certain action during this timestep.

#### 3.2.2 Critic

The output layer of the Critic-MLP consists of one numeric output, which is a value estimation of a given game state. This will be explained further in section 5.2.

### 3.3 Activation Functions

Two different activation functions were used for the hidden layers: the sigmoid function and the Rectified Linear Unit (ReLU) function.

Given an activation $a$, the sigmoid output value $\sigma(a)$ of a neuron is calculated as follows:

$$\sigma(a) = 1/(1 + e^{-a}) \tag{3.2}$$

The ReLU output value is calculated using:

$$\sigma(a) = max(0, a) \tag{3.3}$$

Both activation functions are compared in order to achieve the best performance for the MLP. This will be elaborated upon in section 6.1.

## 4 Learning From Demonstration

Reinforcement Learning alone is often not enough to learn to play a complex game. Hypothetically, we could leave out offline learning and initialize both the Actor and the Critic with an untrained MLP, which the Critic would have to improve. In a game like Donkey Kong, however, this would lead to initial behavior to consist of randomly moving around without getting even remotely close to the goal. In other words: it would be hard to near-impossible for the Actor to reach the goal state, which is necessary for the Critic to improve gameplay behavior.

The above means that we need to pre-train both the Actor and the Critic in order to obtain a reasonable starting performance. For this, we utilized Learning from Demonstration (LfD) (Atkeson and Schaal, 1997). A dataset of input and output patterns for the MLP was created by letting both authors play 50 games each. For each timestep, an input pattern is extracted from the game state using the different input algorithms. Additionally, the action chosen by the player at that exact timestep is stored as well. The observed reward is stored at each time-step as well. The critic uses this reward to create the target explained in Section 5.1, Equation 5.3. All these corresponding input-output patterns make up the data on which the MLPs are pre-trained.

## 5 Reinforcement Learning Framework

The elements of reinforcement learning are combined in the general framework called the Markov Decision Process (MDP), which is the framework that is used in most reinforcement learning problems (Sutton and Barto, 1998; Wiering and van Otterlo, 2012). The MDP can be viewed as a tuple $< S, A, P, R, \gamma >$, where $S$ is the set of all states, $A$ is the set of all actions, $P(s_{t+1}|s_t, a)$ represents the transition probabilities of moving from state $s_t$ to state $s_{t+1}$ after executing action $a$ and $R(s_t, a, s_{t+1})$ represents the reward function. $\gamma$ represents the discount factor, indicating the importance of future rewards. Since in Donkey Kong there is only one main way of winning the game, which is saving the princess, the future reward of reaching her should be a very significant contributor to the value of a state. Furthermore, as explained in Section 2, the agent does not move very far after each action selection. When contrasted with the size of the game screen, this means that at around 2000 steps are needed to reach the goal, where 7 actions are possible at each step, leading to a very challenging environment. For these reasons, the discount factor $\gamma$ is set to 0.999, in order to cope with this long horizon, such that values of states that are, for example, a 1000 steps away from the goal still get a portion of the future reward of the saving the princess.

Besides this tuple, we also have the decision making agent, which takes actions to transition from states to other states. Furthermore, a value function $V(s_t)$ is defined, which maps a state to the expected value of this state, indicating the usefulness of being in that state. Besides the value function, we also define a policy function $\pi(s_t)$ that maps a state to an action. The goal of the reinforcement learning process is to find an optimal policy $\pi^*(s_t)$ such that an action is chosen in each state as to maximize the obtained rewards in the environment. The process is assumed to contain the Markov property. This property assumes that the history of the process is not important to determine the probabilities of state transitions. Therefore, the transition to a state $s_{t+1}$ depends only on the current state $s_t$ and action $a_t$ and not on any of the previous states encountered.

In our Donkey Kong framework, the decision-making agent is represented by Mario, who can choose in each state one of the seven actions to move to a new state, where the state is uniquely defined by the combination of features explained earlier. Like in the work done by Bom et al. (2013), we use a fixed reward function based on specific in-game events. Choosing actions in certain states can trigger these events, leading to positive or negative rewards. We want the agent to improve its game-playing performance by altering its policy. Rewards give an indication of whether a specific action in a specific state led to a good or a bad outcome. In Donkey Kong, the ultimate goal is to rescue the princess at the top of the level. Therefore, the highest positive reward of 200 is given in this situation. One of the challenging aspects of the game is the set of moving barrels that roll around the level. Touching one of these barrels will immediately kill Mario and reset the level, so this behavior should be avoided at all costs. Therefore, a negative reward of -10 is given, regardless of the action chosen by Mario. Jumping around needlessly should be punished as well, since this can lead Mario into a dangerous state more easily. For example, jumping in the direction of an incoming barrel can cause Mario to land right in front of it, with no means of escape left. The entire set of events and the corresponding rewards are summarized in Table 5.1.

| Event | Reward |
|---|---|
| Getting hit by barrel | -10 |
| Jumping over a barrel | +3 |
| Pick up powerup | +1 |
| Destroy barrel with powerup | +2 |
| Save princess | +200 |
| Needless jump | -20 |

**Table 5.1: Game events and their corresponding rewards. A 'needless' jump penalty is only given if the agent jumped, but didn't jump over a barrel nor did the agent pick up a powerup.**

## 5.1 Temporal Difference Learning

In this research, we apply a subset of Reinforcement Learning algorithms called Temporal Difference Learning (Sutton, 1988). The advantage of Temporal Difference (TD) methods is that they can immediately learn from the raw experiences of the environment as they come in and no model of the environment needs to be learned. This means that we can neglect the $P$ part of the MDP tuple explained earlier. TD methods allow learning updates to be made at every time step, unlike other methods that require the end of an episode to be reached before any updates can be made, such as Monte Carlo algorithms. The important prediction function related to the TD methods that we use is the value function, which estimates the value of each state based on future rewards that can be obtained, starting at this state. Therefore, the value of a state $s_t$ is the expected total sum of discounted future rewards starting from state $s_t$, as explained by Sutton and Barto (1998):

$$V(s_t) = E\Big[\sum_{k=0}^{\infty} \gamma^k * R_{t+k+1}\Big] \qquad (5.1)$$

Here, $s_t$ is the state at time $t$, $\gamma$ is the discount factor and $R_{t+k+1}$ is the reward at time $t + k + 1$. We can take the immediate reward

observed in the next state out of the sum, together with its discount factor:

$$V(s_t) = E\Big[R_{t+1} + \gamma * \sum_{k=0}^{\infty} \gamma^{k+1} * R_{t+k+2}\Big] \quad (5.2)$$

We observe that the discounted sum in equation 5.2 is equal to the definition of the value function $V(s_t)$ in equation 5.1, except one time step later into the future. Substituting equation 5.1 into 5.2 gives us the final value function prediction target:

$$V(s_t) = E\Big[R_{t+1} + \gamma * V(s_{t+1})\Big] \quad (5.3)$$

Therefore, the predicted value of a state is the reward observed in the next state plus the discounted next state value.

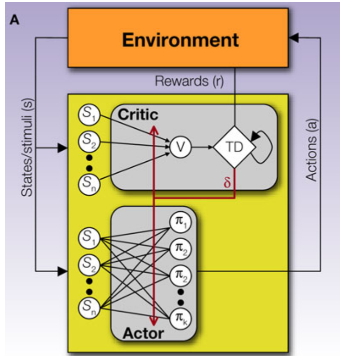


**Figure 5.1: The architecture of Actor-Critic methods (Takahashi et al., 2008)**

## 5.2 Actor-Critic methods

Actor-Critic methods are based on the Temporal Difference learning idea. However, these algorithms represent both the policy and the value function separately, both with their own weights in a neural network or probabilities/values in a table. The policy structure is called the Actor, which takes actions in states. The value structure is called the Critic, who criticizes the current policy being followed by the Actor. The structure of the Actor-Critic model is illustrated in Figure 5.1.

The environment presents the representation of the current state $s_t$ to both the Actor and the Critic. The Actor uses this input to compute the action to execute, according to its current policy. The Actor then selects the action, causing the agent to transition to a new state $s_{t+1}$. The environment now gives a reward based on this transition to the Critic. The Critic observes this new state and computes its estimate for this new state. Based on the reward and the current value function estimation, both $R_{t+1}$ and $\gamma * V(s_{t+1})$ are now available to be incorporated into both making an update to the Critic itself, as well as computing a form of feedback for the Actor. The Critic looks at the difference of the values of both state $s_t$ and $s_{t+1}$. Together with the reward, we can define the feedback $\delta_t$ at time $t$, called the Temporal-Difference error, as follows:

$$\delta_t = R_{t+1} + \gamma * V(s_{t+1}) - V(s_t) \quad (5.4)$$

When one of the two terminal states is encountered, getting hit by a barrel or saving the princess, the value of the next state, $\gamma * V(s_{t+1})$, is set to 0.

If we neglect $R_{t+1}$ for now, we observe that $\gamma * V(s_{t+1}) - V(s_t)$ will be positive if the next state $s_{t+1}$ yields a (much) higher value than the previous state $s_t$. This means that we have improved our situation and entered a better state, meaning that the action selected in state $s_t$ should be positively reinforced. The opposite is also true: a negative $\delta_t$ means our condition has worsened, so the 'bad' action should be penalized. The reward $R_{t+1}$ can help sway the TD-error into the positive or negative direction.

The tendency to select an action has to change, based on the following update rule, Sutton and Barto (1998):

$$h(a_t|s_t) = h(a_t|s_t) + \beta\delta_t, \quad (5.5)$$

where $h(a_t|s_t)$ represents the tendency or probability of selecting action $a_t$ at state $s_t$ and $\beta$ is a positive step-size parameter between 0 and 1.

In the case of Neural Networks, both the Actor and the Critic are represented by their own Multilayer Perceptron. The feedback computed by the Critic is given to the Actor network, where the weights of the output node of the Actor corresponding to the chosen action are

directly acted upon. These weights have to be changed in a way such that the tendency of the action chosen is either positively or negatively reinforced, based on the size and sign of $\delta_t$. Therefore, we form new targets for all of our seven output nodes. Since the feedback is only due to one action selected at the previous state, we assume $\delta_t = 0$ for all output nodes not corresponding to the action selected. Thus, for every output node, we define the target as follows: target(node) = activation(node) + $\delta_t$. Comparing this to equation 5.5, the $\beta$ parameter is assumed to be equal to one, as we already use a learning rate in the neural net itself. After setting the targets, the backward propagation algorithm should change the weights in such a way to increase or decrease the tendency of the chosen action next time the same state as the one observed at time $t$ is encountered.

The Critic is also updated by $\delta_t$. Since the Critic approximates the value function $V(s_t)$ itself, the following equation,

$$\begin{aligned} V(s_t) &= V(s_t) + \delta_t, \\ &= V(s_t) + R_{t+1} + \gamma * V(s_{t+1}) - V(s_t) \end{aligned}$$

is reduced to:

$$V(s_t) = R_{t+1} + \gamma * V(s_{t+1}), \tag{5.7}$$

which is, once again, the value function target for the Critic. Combining equation 5.7 and equation 5.4, we can see that, as the Critic keeps updating and improves its approximation of the value function, $\delta_t = V(s_t) - V(s_t)$, meaning $\delta_t$ converges to 0. This means that the Temporal Difference error will keep decreasing, meaning that the impact of the feedback on the Actor will decrease as well, hopefully at the time where the Actor converges to an optimal policy.

### 5.2.1 ACLA

In this study, we used the Actor-Critic algorithm called Actor-Critic Learning Automaton (Wiering and Van Hasselt, 2007). This algorithm functions in the same basic way as standard Actor-Critic methods, except in the way the Temporal Difference error is used for feedback. As explained before, standard Actor-Critic methods calculate the feedback $\delta_t$, then use this value to alter the tendency to select certain actions by changing the parameters of the Actor. ACLA does not use the exact value of $\delta_t$, but only looks at whether or not an action selected in the previous state was good or bad. Therefore, instead of the value, the sign of $\delta_t$ is used. While the Critic still uses the value of $\delta_t$ for its update, ACLA slightly changes the way the Actor's weights should be changed. After calculating $\delta_t$, the targets for the output nodes of the Actor are set, depending on the sign of $\delta_t$. If $\delta_t$ is positive, we reinforce the action selected at time $t$, by using the same one-hot vector encoding that was used during the demonstration learning phase. For example, if action 6 needs to be reinforced, the targets are represented as the one-hot target vector [0 0 0 0 0 1 0]. If $\delta_t$ is negative, we penalize the chosen action while keeping the other actions the same. The target for the 'bad' action is 0, while the target of all the other output nodes is the Softmax value they had at state $s_t$, meaning that the error for these output nodes is 0. After this, a backward pass will adjust the weights to achieve the desired effect.

## 5.3 Implementation

There are a few implementation details regarding the reinforcement learning methods that need to be specified. Preliminary tests showed that standard Actor-Critic and ACLA with negative reinforcement caused the Actor network to become unstable, resulting in bad playing behavior that disturbed the entire performance. For example, this instability sometimes caused Mario to forget how to climb a ladder, getting stuck into a jumping loop around it instead. Therefore, after more tests, ACLA with only positive feedback was implemented. This means that any negative feedback computed by the Critic is not propagated to the Actor. This resulted in more stability.

Another method was implemented to increase the stability of the network during training. With Temporal Difference learning, data to learn from comes in at every time step. How-

ever, this data always comes in the same order due to the nature of the game, making the samples the network has to learn from highly correlated with each other. This high correlation makes it harder for the Neural network to learn. In order to reduce this sample correlation, a method called Experience Replay (Lin, 1992) is applied. With this technique, a memory dataset with a fixed size of 50000 is used, where state transitions and rewards are stored. Then, at every time step, a sample is randomly and uniformly sampled from the memory dataset and the Actor is trained on this sample. In our experiments, to reduce computation time, a sample contained one experience. Preliminary tests seem to show that Experience Replay is the biggest factor that contributes to the stability of the network being trained with ACLA.

One final addition to the game environment is the inclusion of two invisible walls, one to the right of the last ladder in the bottom right of the screen. The other invisible wall is placed to the left of the ladder right under the princess. Mario can not pass through these. The inclusion of these walls prevent unstable behavior from completely ruining an entire run. Some networks seem to become unstable in only a smart part of the game environment, almost always the two locations where the invisible walls are placed. For example, a certain network often became unstable after around 130 games played, causing the agent to keep jumping around the ladder in the bottom-right corner of the screen instead of climbing it. However, the network still performed well in all other sections of the environment.

Another implementation issue relates to the Markov property. The Donkey Kong game violates this property, resulting in numerous workarounds that needed to be implemented in order for the right states to be associated with good or bad actions. Donkey Kong violates the Markov property because not every state transition depends purely on the current state. For example, during a jump, the agent can not select any other actions until it lands on the ground again. Therefore, during a jump, all the state transitions in the states where Mario is in the air after a jump depend heavily on the state where the initial jump was selected, which in our game implementation can almost be 50 states back into the past. Finally, the vision grid gives Mario only local vision, meaning that the entire game state is only partially observable for the agent.

# 6 Experiments and Results

In order to test the performance of the Actor-Critic methods, several experiments were performed. We define the performance as the percentage of games where the agent was able to reach the princess: $\frac{gamesWon}{gamesPlayed}$.

In the first experiment, the parameters for the MLP trained using learning from demonstration were optimized in order to achieve a good baseline performance. We then perform 10 runs of 100 games to see how the optimized Actor performs without any Reinforcement Learning.

For the second experiment, we compare the performance of only the Actor versus an Actor trained with ACLA for 5 different models. Between each model, the performance of the Actor-MLP is varied: we do not only want to see if ACLA is able to improve our best Actor, but we want to know whether it can heighten the performance of lower-performing Actors as well.

## 6.1 Parameter Optimization

The MLP has several parameters that influence the performance of the network: the learning rate, temperature (see section 3.2), the amount of hidden nodes per layer, the amount of hidden layers and the activation function (see section 3.3) We want the Actor-MLP to achieve a reasonable baseline performance, meaning the parameter settings had to be optimized. For each parameter value, the MLP was initialized and trained 10 times, each time playing 100 games after training. While testing a parameter, all other parameters were kept at a default value.

First, we compared parameter settings using only the sigmoid activation function. Table 6.1 shows the tested parameter settings while using

only the sigmoid functions, together with the default setting for each parameter.

**Sigmoid**

| Parameter | Parameter values | Default |
|---|---|---|
| Learning rate | {0.0005, 0.001, 0.005, 0.01, 0.05} | 0.01 |
| Temperature | {0.5, 1, 2, 4, 8} | 1 |
| N hidden nodes | {25, 50, 100, 150, 200} | 100 |
| N hidden layers | {1, 2, 3} | 1 |

**Table 6.1: The values that were tested for each parameter using a sigmoid activation function. The third column shows the parameter's default setting.**

ReLU typically requires a lower learning rate than a sigmoid activation function. Furthermore, we wanted to compare the effects of combining multiple hidden layers between both activation functions. Table 6.2 shows the tested parameter settings while using only the ReLU function (temperature = 1, 100 hidden nodes). Note that we used either only sigmoid or only ReLU activation functions: due to both functions requiring different learning rate values, we did not test a combination of sigmoid and ReLU while using 2 or more hidden layers.

**ReLU**

| Parameter | Parameter values | Default |
|---|---|---|
| Learning rate | {0.00005, 0.0001, 0.0005 0.001, 0.005} | 0.01 |
| N hidden layers | {1, 2, 3} | 1 |

**Table 6.2: The values that were tested for each parameter using a ReLU activation function. The third column shows the parameter's default setting.**

The full training data set consists of data from 100 played games, both authors each having played 50 games. However, in order to keep the training time at a reasonable level, a data set of 50 games was used during parameter optimization. For the stop criterion we look at the difference in the error between two training epochs. The error in each output neuron is defined as $\frac{1}{2}(t-o)^2$, where $t$ is the neuron's target and $o$ its output. The MLP stops training when the average error over all 7 output neurons for all input patterns in the dataset becomes lower than 0.0007. Additionally, when multiple hidden layers are used, each consecutive hidden layer has half the amount of nodes of the previous layer. If the MLP has three layers with $n$ nodes in the first layer, the second and third layers each have $\frac{n}{2}$ and $\frac{n}{4}$ hidden nodes, respectively.

## 6.2 Parameter Optimization Results

Tables 6.3 to 6.8 show the average performance of 10 runs of 100 games for all tested parameter setting. The best setting for each parameter is in boldface.

**Learning rate (sigmoid)**

| Value | Mean performance |
|---|---|
| 0.0005 | 10.3% (SD 2.2) |
| 0.001 | 14.2% (SD 3.3) |
| 0.005 | 37.3% (SD 9.5) |
| **0.01** | **40.6% (SD 5.4)** |
| 0.05 | 38.5% (SD 4.9) |

**Table 6.3: The average performance of 10 runs of 100 games for 5 different learning rate values using a sigmoid activation function**

**Temperature**

| Value | Mean performance |
|---|---|
| 0.5 | 38.2% (SD 7.4) |
| 1 | 35.9% (SD 6.2) |
| 2 | 41.0% (SD 5.9) |
| **4** | **45.9% (SD 6.3)** |
| 8 | 34.7% (SD 5.5) |

**Table 6.4: The average performance of 10 runs of 100 games for 5 different temperatures**

**N hidden nodes**

| Value | Mean performance |
|---|---|
| 25 | 36.7% (SD 4.9) |
| 50 | 33.4% (SD 9.2) |
| 100 | 37.3% (SD 8.2) |
| 150 | 36.1% (SD 5.0) |
| **200** | **42.3% (SD 7.0)** |

**Table 6.5: The average performance of 10 runs of 100 games for 5 different hidden node sizes**

**N hidden layers (sigmoid)**

| Value | Mean performance |
|---|---|
| 1 | 38.5% (SD 6.4) |
| **2** | **42.2% (SD 5.0)** |
| 3 | 38.9% (SD 3.5) |

**Table 6.6: The average performance of 10 runs of 100 games for 3 different amounts of hidden layers using a sigmoid activation function**

**Learning rate (ReLU)**

| Value | Mean performance |
|---|---|
| 0.00005 | 9.0% (SD 2.5) |
| 0.0001 | 14.3% (SD 3.7) |
| 0.0005 | 27.2% (SD 5.1) |
| 0.001 | 34.3% (SD 4.6) |
| **0.005** | **41.7% (SD 10.9)** |

**Table 6.7: The average performance of 10 runs of 100 games for 5 different learning rate values using a ReLU activation function**

**N hidden layers (ReLU)**

| Value | Mean performance |
|---|---|
| **1** | **39.3% (SD 6.6)** |
| 2 | 37.6% (SD 7.9) |
| 3 | 34.2% (SD 6.8) |

**Table 6.8: The average performance of 10 runs of 100 games for 3 different amounts of hidden layers using a sigmoid activation function**

Tables 6.3 and 6.7 show that the highest performance achieved by varying the learning rate is comparable between an MLP using sigmoid and an MLP using ReLU when using 1 hidden layer. However, Table 6.6 shows that adding more hidden layers further improves the performance for a sigmoid-MLP, which is not the case with ReLU (Table 6.8). We therefore chose the settings found in Table 6.9 for training an optimized network. The bottom row shows the average performance of 10 MLPs trained with these settings, each having played 100 games. This time, the full dataset of 100 games was used. Furthermore, the stop criteron was set to a minimal error difference of 0.0001.

| Parameter | Value |
|---|---|
| Learning rate | 0.01 |
| Temperature | 4 |
| N hidden nodes | 200 |
| N hidden layers | 2 |
| Activation function | Sigmoid |
| Performance | 49.6% |

**Table 6.9: The optimized parameter settings. The bottom rows shows the average performance of 10 runs of 100 games**

Note that since all parameters (except for the one being tested) were kept at a constant value, some trials were repeated. For example, the trials leading to the results as shown in the fourth row in Table 6.3 and the second row in Table 6.4 use the same parameter settings, since the default learning rate value for sigmoids is 0.01 and the default temperature is 1. One would expect the average performance for both trials to be approximately the same. How a network performs, however, is dependent on how it is initialized and trained. Since each network is re-initialized 10 times for each trial, there are quite some fluctuations in the performance even when the parameters are kept at constant values. This difference is reflected by the standard deviations as shown in Tables 6.3 to 6.7.

## 6.3 Model Selection for RL

During the Reinforcement Learning experiments, the ACLA algorithm was applied to a few different Actor networks. The selected networks were selected based on their performance

on the 10 × 100 games. For example, the first model we considered is an Actor trained with the combination of the best parameters for the sigmoid activation function, found as a result of the parameter optimization. We consider two networks using the sigmoid activation functions and two networks using the ReLU activation function. The last model differs from the other 4: this model is only trained for 2 epochs, meaning that the model is quite bad, leaving much room for improvement. Besides model 5, the two sigmoid models were trained until a minimum change in error between epochs of 0.00005 was reached, while the two ReLU models had a minimum change threshold of 0.0007. The reason that the ReLU models' threshold is higher than the Sigmoid models', is that preliminary results have shown that the error did not decrease further after extended amounts of training for MLPs using ReLU.

Table 6.10 displays and details all 5 models that we considered and tried to improve during the Reinforcement Learning trials together with their performance and its standard error.

| Model | N hidden layers | N hidden nodes | learning rate | Activation function | Performance |
|---|---|---|---|---|---|
| 1 | 2 | 200 | 0.01 | sigmoid | 56.6% (SE: 1.08) |
| 2 | 1 | 50 | 0.001 | sigmoid | 29.9% (SE: 1.08) |
| 3 | 1 | 100 | 0.005 | ReLU | 48.6% (SE: 2.02) |
| 4 | 2 | 50 | 0.001 | ReLU | 50.6% (SE: 1.46 ) |
| 5 | 1 | 80 | 0.01 | sigmoid | 12.6% (SE: 0.90 ) |

**Table 6.10: Details of the 5 models that were used in the Reinforcement Learning trials**

## 6.4 Reinforcement Learning Experiments

This section explains how the reinforcement trials were set up. Each of the 5 models is trained during one ACLA session. This learning session lasts 1000 games, where the temperature of the Boltzmann distribution starts at a value of 8. This temperature is reduced every 200 games, such that the last 200 games are run at the lowest temperature of 4. Preliminary results showed that most networks performed best at this temperature. The ACLA algorithm is applied at every step, reinforcing positive actions. The learning rate of the Actor is set to 0.0001, so that ACLA can subtly push the Actor into the right direction. The Critic also uses a learning rate of 0.0001. The Critic is already trained well on the demonstration data as explained in Section 4, so the low learning rate allows the Critic to increase its approximation of the value function without the Critic being changed too much. Setting the learning rate too high causes the network to become unstable. In this event, state values can become very negative, especially when the Actor encounters a lot of negative rewards. The overall trend of the performance over time during the 1000 games session for model 1 and 5 can be seen in Figure 6.1 and Figure 6.2. These are displayed because they show the most interesting trends out of the 5 models. The performance is defined as $\frac{gamesWon}{gamesPlayed}$.

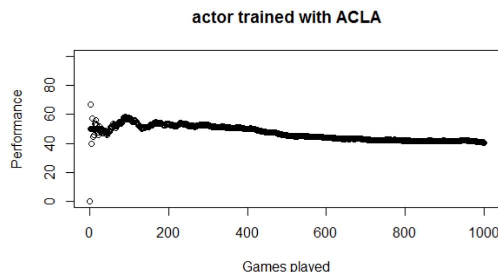


**Figure 6.1: Performance of model 1 trained with ACLA over 1000 games**

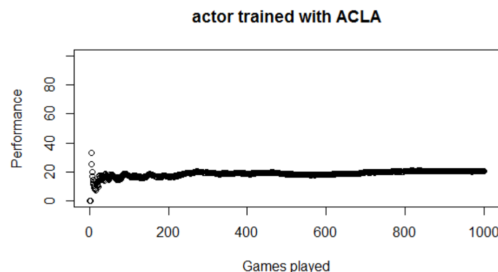


**Figure 6.2: Performance of model 5 trained with ACLA over 1000 games**

After the 1000-games training sessions, the performance of the Actors trained with ACLA were compared to the performance of the Ac-

tors before training with ACLA. For each of the 5 models, both Actors were tested in $10 \times 100$ games, both with a fixed temperature of 4. The results of the trained Actor performances are shown in Table 6.11. The final results are shown in Figure 6.3, where the performance of each model's Actor versus the model's Actor trained with ACLA are shown.

| Statistic | Model 1 | Model 2 | Model 3 | Model 4 | Model 5 |
|---|---|---|---|---|---|
| MEAN | 45.8% | 31.2% | 44.5% | 20.8% | 26.4% |
| SE | 1.45 | 1.46 | 0.76 | 1.34 | 1.59 |

**Table 6.11: Results of the models trained with ACLA on 10 runs**

## 6.5 Analysis of Results

Looking at Figure 6.3, the differences in performance can be seen for each model, together with standard error bars which have a length of $4 \times SE$. From this figure, we see that the error bars of models 2 and 3 overlap. This might indicate that these differences in performances are not significant. The other 3 models do not have overlapping error bars, suggesting significance. In order to test for significance, we use a nonparametric Wilcoxon rank sum test, since the performance scores are not normally distributed. The five Wilcoxon rank sum tests yield the results in Table 6.12. The Wilcoxon rank sum test confirms a significant effect of ACLA on models 1 ($W = 41.5$, $p < 0.05$), 4 ($W = 100$, $p < 0.05$) and 5 ($W = 0$, $p < 0.05$), but not on models 2 ($W = 41.5$, $p > 0.05$) and 3 ($W = 27$, $p > 0.05$). These results seem to confirm the observations made earlier with respect to the error bars in Figure 6.3.

| Model | W-value | p-value |
|---|---|---|
| 1 | 98.5 | 0.0003 |
| 2 | 41.5 | 0.54 |
| 3 | 27 | 0.087 |
| 4 | 100 | 0.0001 |
| 5 | 0 | 0.0001 |

**Table 6.12: Results of the Wilcoxon rank sum test on the 5 models**

# 7 Discussion

Using parameter optimization, we were able to find an MLP that is able to obtain a reasonable baseline performance by using learning from demonstration. The best model, model 1, was able to achieve an average performance of 56.6%. In addition to this, several MLPs were trained with different parameter settings, resulting in a total of 5 neural net models. The performance of these 5 models vary, so that we can see how robustly the Actor-Critic method is able to improve differently performing agents.

While the performance achieved by an Actor that is only trained offline is not too bad, ACLA does not usually seem to be able to improve this any further. Even worse, the Actor's performance can start to decline over time. Only a model that is barely pre-trained on demonstration data can obtain a significant improvement. We therefore conclude that a combination of neural nets and Actor-Critic is in most cases not able to achieve good gameplay performance in Donkey Kong.

How generalizable the trained agent is has yet to be established, as almost no research into this has been conducted in this thesis. However, one aspect of the generalizability has been explored, pointing out a flaw in the preparation of the demonstration data. During this demonstration phase, the location of the powerups is fixed. This causes the neural network to wrongly associate the specific Mario-tracking grid block with the need to jump to collect the powerup. This mistake is exposed when the powerups in question are temporarily removed: the trained Actor will jump in the specific Mario-tracking grid block that would normally give him the powerup, even if the powerup is not there. The probable solution to this problem is to randomize the location of the powerups every game, such that the neural network actually associates jumping when the powerup shows up close enough in the vision grid instead of associating the jump with the Mario-tracking grid blocks.

The results in section 6.2 show that lower learning rate values lead to bad performance for both the sigmoid and ReLU activation func-

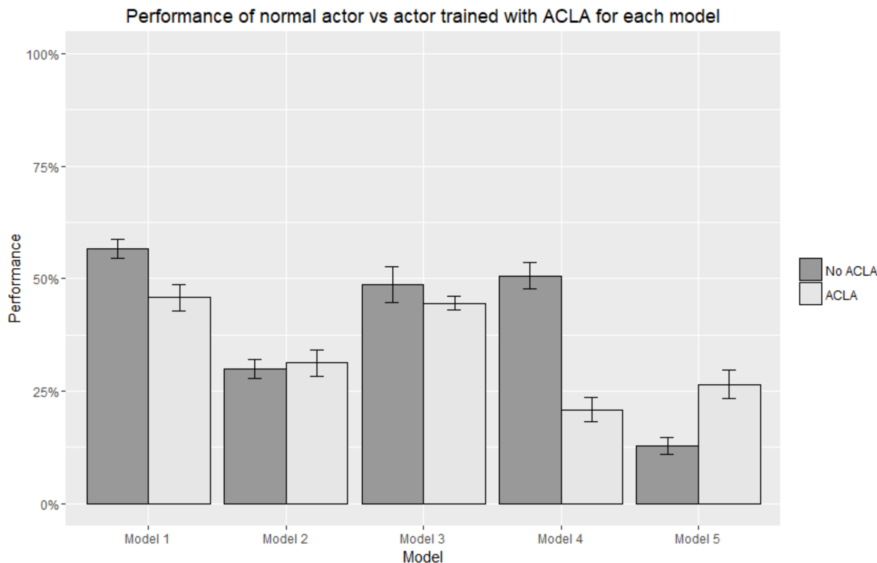


**Figure 6.3: Performances of Actor trained without ACLA vs. Actor trained with ACLA for each model. The error bars show two standard errors (SE) above and below the mean**

tions. This is partly due to the stop criterion used: once the difference in the error is below a certain threshold, the training stops. This threshold was kept equal for all parameter experiments. The lower the learning rate, the smaller the error change between epochs. This poses a problem for lower learning rate values: on one hand, the MLP needs to train longer to reduce the error to a reasonable level, while on the other hand the training is more likely to stop prematurely due to the error difference between two epochs being too low. The low performances achieved by these MLPs may therefore not be entirely representative of the performance they would have had if they were to have more time to train.

## 7.1 Future research

Future research could result in better playing performance than those obtained in this research. Actor-Critic methods turned out to not be able to improve the performance of the agent. Therefore, other reinforcement learning algorithms and techniques could be explored, such as Q-learning (Watkins, 1989), advantage learning (Baird, 1995) or Monte Carlo methods. A recent method has been introduced called the Hybrid Reward Architecture, which has been applied to Ms. Pac-Man to achieve a very good performance (Van Seijen et al., 2017). Applying this method to Donkey Kong could yield better results. Another option is inplementing a discrete version, where every entity in the environment moves in a grid instead of with continuous pixels, could also lead to better results, but such an environment would not be as difficult or interesting to research. Additionally, the generalizability of the agent to other levels can be explored in future research.

# References

Atkeson, C. G. and Schaal, S. (1997). Robot learning from demonstration. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, pages 12–20, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*,

pages 30–37. Morgan Kaufmann Publishers Inc.

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

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–533.

Shantia, A., Begue, E., and Wiering, M. (2011). Connectionist reinforcement learning for intelligent unit micro management in starcraft. *The 2011 International Joint Conference on Neural Networks*, pages 1794–1801.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* Cambridge: The MIT Press.

Takahashi, Y., Schoenbaum, G., and Niv, Y. (2008). Silencing the critics: understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an actor/critic model. *Frontiers in Neuroscience*, 2(1):86–99. Princeton Neuroscience Institute, Princeton University, Princeton, NJ, USA.

Van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., and Tsang, J. (2017). Hybrid reward architecture for reinforcement learning. Retrieved from https://arxiv.org/abs/1706.04208.

Watkins, C. J. (1989). Learning from delayed rewards. *PhD Thesis, University of Cambridge, England.*

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

Wiering, M. A. and Van Hasselt, H. (2007). Two novel on-policy reinforcement learning algorithms based on TD($\lambda$)-methods. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 280–287.

# A Division of Workload

- Developing the game environment: done in equal parts by Jelle and Paul
- Programming the MLP: Jelle
- Implementing Learning from Demonstration:
  - Storing game state information: Paul and Jelle
  - Reading game state information and converting them to MLP inputs and targets: Paul
- Implementing reinforcement learning methods (AC, ACLA): Paul
- Implementing the vision grids: Paul
- Creating learning from demonstration dataset: done in equal parts by Jelle and Paul
- MLP optimization and experiments: Jelle
- RL experiments: Paul
- Thesis:
  - Section 2, 3, 4, 6-6.2 written by Jelle
  - Abstract and section 1, 5, 6.3-6.5 written by Paul
  - Section 7 co-written by Jelle and Paul
  - Proofreading and correcting done in equal parts by Jelle and Paul
- Both authors met often to work together. During these sessions, they helped each other debug their code and solve problems that arose during the project.