Learning to Play Donkey Kong Using Neural Networks and Reinforcement Learning

Paul Ozkohen[1], Jelle Visser[1], Martijn van Otterlo[2], and Marco Wiering[1(✉)]

[1] University of Groningen, Groningen, The Netherlands
{p.m.ozkohen,j.visser.27}@student.rug.nl, m.a.wiering@rug.nl
[2] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
m.van.otterlo@vu.nl

Abstract. Neural networks and reinforcement learning have successfully been applied to various games, such as Ms. Pacman and Go. We combine multilayer perceptrons and a class of reinforcement learning algorithms known as actor-critic to learn to play the arcade classic Donkey Kong. Two neural networks are used in this study: the actor and the critic. The actor 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 learning from demonstration, where data is obtained from human players. After this off-line training phase we further improve the 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 agent 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 decreases it. Possible reasons include the game not fully being Markovian and other issues.

Keywords: Machine learning · Neural networks
Reinforcement learning · Actor-critic · Games · Donkey Kong
Platformer

1 Introduction

Games have been a prime 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 (AI)* 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, or: general, AI, which can be considered one of the primary goals of the entire field.

The third author acknowledges support from the Amsterdam academic alliance (AAA) on data science.

B. Verheij and M. Wiering (Eds.): BNAIC 2017, CCIS 823, pp. 145–160, 2018.
https://doi.org/10.1007/978-3-319-76892-2_11

Reinforcement learning (RL) techniques have often been used to achieve success in creating game-playing agents [5,7]. RL 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 states are continuous. Unfortunately, tables would have to become enormous in order to store all the necessary function information. To solve this problem in RL, *function approximation* can be applied, often using *neural networks*. A famous recent example of this is the ancient board game Go, in which DeepMind's AI *AlphaGo* was able to beat the world's best players at their own game [7]. Besides traditional games, it was used to learn to play video games. For example, DeepMind used a combination of convolutional 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 [5]. That study shows how an RL algorithm can be trained purely on the raw pixel images. The upside of that research is that a good game-playing performance can 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.

An alternative approach is to use hand-crafted game-specific features. One such game where this was successfully applied is Ms. Pac-Man, where an AI was trained to achieve high win rates using higher-order, game-specific features [3]. This approach shows that good performance can be obtained with a small amount of inputs, therefore severely reducing computation time.

In this paper we present an approach to machine learning in games that is more in line with the second example. We apply RL methods to a video game based on *Donkey Kong*, an old arcade game that was released in 1981 by Nintendo [4]. 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 RL. 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 a random way. The intriguing part of studying this game is to see whether RL can deal with such an unpredictable and timing-based continuous environment. We specifically focus on the very first level of the Donkey Kong

game, as this incorporates all the important elements mentioned above while also making the learning task simpler. Other levels contain significantly different mechanics, such as springs that can launch Mario upwards if he jumps on it, or vertically moving platforms. We do not consider these mechanics in this research.

For this study we used a specific RL technique called *actor-critic* [9]. 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. More specifically, we employ a variant of actor-critic: the *actor-critic learning automaton* (ACLA) [14].

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 a set of games being played by human players.

The main question this paper 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? In the next sections we will first define the domain and its features, after which we discuss our machine learning setup and methodology and we conclude with results and discussion.

2 The Domain: A Donkey Kong Implementation

A framework was developed that allows the user to test several RL techniques on a game similar to the original Donkey Kong. The game itself can be seen in Fig. 1 and was recreated from scratch as a Java application.

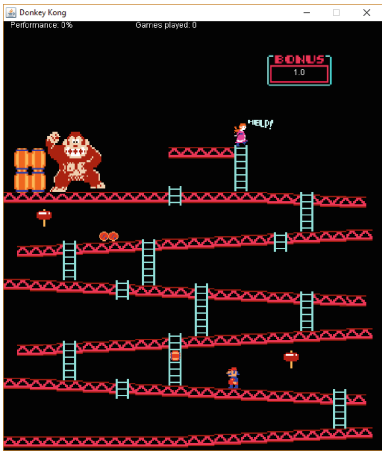

Fig. 1. Recreation of Donkey Kong.

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, who 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 player touching a barrel results in an instant loss (and "game over"), while jumping over them nets a small score. Additionally, two power-ups (hammers) can be picked up by the player when he collides with them by either a walking or jumping action, which results in the barrels being destroyed upon contact with the agent, netting a small score gain as well. This powerup is temporary. 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 a 680 × 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, which implies it requires many actions to reach the princess from the initial position.

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 some 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 we opted for a simple algorithm where the barrels' odds of rolling down a ladder is set to be simply 50% at any given time.

The built environment supports *manual* mode, in which a human player can interact with the game, and two *automated* modes in which an MLP is used to control Mario (either only using an actor network, or learning with a critic). While there are a few notable differences between the original game and our 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.

3 Generalization: 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.

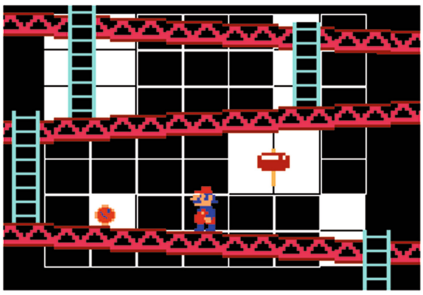

Fig. 2. Visualization of the vision grid that tracks objects directly around the agent, granting Mario local vision of his immediate surroundings. Note that while only one grid can be distinguished, there are actually three vision grids stacked on top of each other, one for each object type.

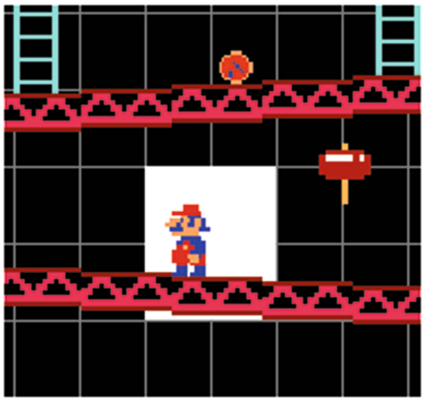

Fig. 3. Visualization of the level-wide grid that tracks the current location of the agent. While not visible in this image, the grid spans the entire game environment.

3.1 Feature Construction for MLP Input

This section provides an overview of how inputs for the MLPs are derived from the game state. Two algorithms employ 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.

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 Mario. A similar method was used by Shantia et al. [6] 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 Mario. Barrels that are far away pose no immediate threat. This changes when a barrel is on the same platform level as Mario: at this point, Mario 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 Mario 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 destroy the barrels, making Mario 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 Mario.

In order to track these objects, we use a set of three grids of 7×7 cells, where each grid is responsible for tracking one object type. The grids are fixed on Mario, 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 $3 \times 49 = 147$ 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, these *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. An example of the vision grid is shown in Fig. 2.

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 Sect. 5. The agent's location in the game is tracked using a 20×20 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 *agent tracking grid* provides $20 \times 20 = 400$ Boolean inputs. An example tracking grid can be seen in Fig. 3.

There are some additional features, such as Booleans that track whether Mario is currently jumping or climbing. The total amount of features is the sum of 147 vision grid cells, 400 agent tracking grid cells and 4 additional inputs, resulting in 551 in total. The four additional inputs are extracted from the game state 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. We compute it as the ratio $\frac{t}{d}$ between the time passed since the powerup was obtained (t) and the total time a powerup remains active (d).

3.2 MLP Output

For the **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 [9]. Here, each action is given a probability based on its activation. Using a *Boltzmann distribution*, we can transform a vector a of length n, consisting of real output activation values, into a vector $\sigma(a)$ consisting of n real values in the range $[0, 1]$. The probability for a single output neuron (action) i is calculated as follows:

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

where τ is a positive real temperature value which can be used to induce exploration into action selection. For $\tau \to \infty$, all actions will 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 Eq. 1. This value stands for the probability that the actor will choose a certain action during this timestep. The output layer of the **critic** consists of one numerical output, which is a *value estimation* of a given game state. This will be explained further in Sect. 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:

$$\sigma(a) = 1/(1 + e^{-a}) \tag{2}$$

The ReLU output value is calculated using:

$$\sigma(a) = max(0, a) \tag{3}$$

Both activation functions are compared in order to achieve the best performance for the MLP. This will be elaborated upon in Sect. 6.

4 Learning from Demonstration

RL alone is sometimes 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. This 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) [1]. A dataset of input and output patterns for the MLP was created by letting the first two authors play 50 games each. For each timestep, an input pattern is extracted from the game state as explained before. Additionally, the action chosen by the player at that exact timestep (and the observed reward) is stored. The critic uses the reward to compute a target value as is explained later. All these corresponding input-output patterns make up the data on which the MLPs are pre-trained.

5 Reinforcement Learning Framework

Our game is modeled as a *Markov Decision Process* (MDP), which is the framework that is used in most RL problems [9,13]. An MDP is a tuple $\langle S, A, P, R, \gamma \rangle$, 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 for this transition. The discount factor γ indicates 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 Sect. 2, the agent does not move very far after each action selection. When contrasted with the size of the game screen, this means that 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 γ 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 saving the princess. 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 RL is to find an *optimal* policy $\pi^*(s_t)$ such that an action is chosen in each state in order to maximize the obtained rewards in the future. The environment is assumed to satisfy the *Markov property*, which assumes that the history of states 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. [3], 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 1.

Table 1. Game events and their corresponding rewards. A 'needless' jump penalty is only given if the agent jumped, but did not jump over a barrel nor did the agent pick up a powerup.

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

5.1 Temporal Difference Learning

Our RL algorithms are a form of *temporal difference* (TD) learning [8,9]. The advantage of 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). Central to TD methods 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:

$$V(s_t) = E\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\Big] \tag{4}$$

Here, s_t is the state at time t, γ 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:

$$V(s_t) = E\Big[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^{k+1} R_{t+k+2}\Big] \tag{5}$$

We observe that the discounted sum in Eq. 5 is equal to the definition of the value function $V(s_t)$ in Eq. 4, except one time step later into the future. Substituting Eq. 4 into Eq. 5 gives us the final value function prediction target:

$$V(s_t) = E\Big[R_{t+1} + \gamma V(s_{t+1})\Big] \tag{6}$$

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

5.2 Actor-Critic Methods

Actor-critic methods are based on the TD 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, which criticizes the current policy being followed by the actor. The structure of the actor-critic model is illustrated in Fig. 4.

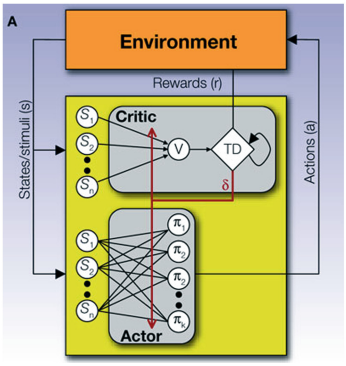

Fig. 4. The architecture of actor-critic methods [10].

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 δ_t at time t, called the TD error, as follows:

$$\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{7}$$

When a terminal state is encountered (hit by a barrel or saving the princess) the value of the next state, $\gamma V(s_{t+1})$, is set to 0. The tendency to select an action has to change, based on the following update rule [9]:

$$h(a_t|s_t) = h(a_t|s_t) + \beta\delta_t, \tag{8}$$

where $h(a_t|s_t)$ represents the tendency or probability of selecting action a_t at state s_t and β 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. The critic is also updated by δ_t. Since the critic approximates the value function $V(s_t)$ itself, the following equation (where the updated V' is computed from V):

$$\begin{aligned} V'(s_t) &= V(s_t) + \delta_t, \\ &= V(s_t) + R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \end{aligned}$$

is reduced to:

$$V'(s_t) = R_{t+1} + \gamma V(s_{t+1}), \tag{9}$$

which is, once again, the value function target for the critic. We can see that as the critic keeps updating and improves its approximation of the value function, $\delta_t = V'(s_t) - V(s_t)$ should converge to 0, which decreases the impact on the actor likewise, which can converge to a (hopefully optimal) policy.

We employ the actor-critic algorithm called *actor-critic learning automaton* [14]. This algorithm functions in the same basic way as standard actor-critic methods, except in the way the TD error is used for feedback. As explained before, standard actor-critic methods calculate the feedback δ_t and use this to alter the tendency to select certain actions by changing the parameters of the actor. ACLA does not use the exact value of δ_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 δ_t is used, and a one-hot vector is used as the target.

6 Experiments and Results

In our experiments 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 RL. 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 increase the performance of lower-performing actors as well.

6.1 Model Selection for RL

During the RL experiments, the ACLA algorithm was applied to a few different actor networks. The 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 a separate parameter optimization phase. We consider two networks using the sigmoid activation functions and two networks using the ReLU activation function. The fifth model differs from the other 4: this model is only pre-trained for 2 epochs. This small amount of pre-training means that the model is quite bad, leaving much room for possible improvement by the critic. 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 2 displays and details all 5 models that we considered and tried to improve during the RL trials together with their performance and standard error (SE).

Table 2. Details of the 5 models that were used in the RL trials. The performance means the % of trials in which Mario gets to the Princess in 100 games. The results are averaged over 10 simulations with MLPs trained from scratch.

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)

6.2 Online Learning Experiments

This section explains how the RL 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. Such low learning rates are required to update the approximations (that were already trained well on the demonstration data) *cautiously*. Setting the learning rate too high causes the networks to become unstable. In this event, state values can become very negative, especially when the actor encounters a lot of negative rewards.

After the 1000-games training sessions, the performance of the actors trained with ACLA were compared to the performance of the actors before training with ACLA. For each of the 5 models, both actors were tested in 10×100 games, both with a fixed temperature of 4. The results of the trained actor performances are shown in Table 3. The final results are shown in Fig. 5, where the performance of each model's actor versus the model's actor trained with ACLA are shown.

Table 3. Results of the models trained with ACLA on 10 runs

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

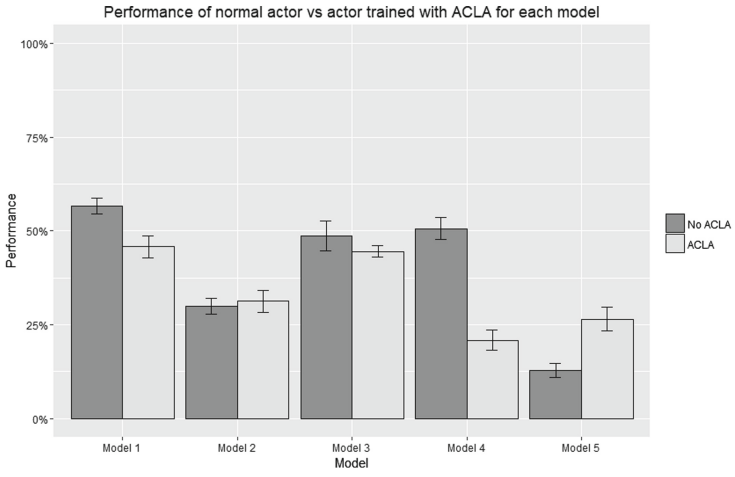

Fig. 5. Performances of the actor trained with vs. without ACLA for each model. The error bars show two standard errors (SE) above and below the mean

6.3 Analysis of Results

Looking at Fig. 5, 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 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 Fig. 5.

6.4 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 winning the game-level 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 varies based on how robustly the actor-critic method is able to improve these models.

While the performance achieved by an actor that is only trained offline is reasonable, 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 improve on a reasonable policy that was obtained through learning from demonstration.

7 Conclusions

We have presented our Donkey Kong simulation and our machine learning experiments to learn good gameplay. We have employed LfD to obtain reasonable policies from human demonstrations, and experimented with RL (actor-critic) to learn the game in an online fashion. Our results indicate that the first setting is more stable, but that the second setting has possibly still potential to improve automated gameplay. Overall, for games such as ours, it seems that LfD can go a long way if the game can be learned from relevant state-action examples. It may well be that for Donkey Kong, in our specific level, the right actions are clear from the current game state and additional delayed reward aspects play a less influencing role, explaining the lesser effect of RL in our experiments. More research is needed to find out the relative benefits of LfD and RL. Furthermore, in our experiments we have focused on the global performance measure of

percentage of games won. Further research could focus on more finegrained performance measures using the (average) reward, and experiment with balancing the various (shaping) rewards obtained for game events (see Table 1).

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 [12], advantage learning [2] 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 [11]. Applying this method to Donkey Kong could yield better results. Additionally, it would be interesting to see whether having more demonstration data positively affects performance. Since we only focused on the very first level of the game, further research is needed to make the playing agent apply its learned behavior to different levels and different mechanics.

References

1. Atkeson, C.G., Schaal, S.: Robot learning from demonstration. In: Proceedings of the International Conference on Machine Learning, pp. 12–20 (1997)
2. Baird, L.: Residual algorithms: reinforcement learning with function approximation. In: Proceedings of the Twelfth International Conference on Machine Learning, pp. 30–37 (1995)
3. Bom, L., Henken, R., Wiering, M.: Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs. In: IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (2013)
4. Donkey Kong fansite wiki. http://donkeykong.wikia.com/wiki/Nintendo. Accessed Sept 2017
5. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015)
6. Shantia, A., Begue, E., Wiering, M.: Connectionist reinforcement learning for intelligent unit micro management in Starcraft. In: The 2011 International Joint Conference on Neural Networks, pp. 1794–1801 (2011)
7. 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., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
8. Sutton, R.S.: Learning to predict by the methods of temporal differences. Mach. Learn. **3**(1), 9–44 (1988)
9. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, Cambridge (1998)
10. Takahashi, Y., Schoenbaum, G., Niv, Y.: Silencing the critics: understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an actor/critic model. Front. Neurosci. **2**(1), 86–99 (2008)
11. van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., Tsang, J.: Hybrid reward architecture for reinforcement learning (2017). https://arxiv.org/abs/1706.04208

12. Watkins, C.J.: Learning from delayed rewards. Ph.D. Thesis, University of Cambridge, England (1989)
13. Wiering, M., van Otterlo, M.: Reinforcement Learning: State of the Art. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27645-3
14. Wiering, M.A., Van Hasselt, H.: Two novel on-policy reinforcement learning algorithms based on TD(λ)-methods. In: 2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning, pp. 280–287 (2007)