Towards artificially playing the game of double pong

Combining learning and search algorithms

Master's thesis in Physics

MARCUS REMGÅRD CHRISTIAN NILSSON

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se

Master's thesis 2022

Towards artificially playing the game of double pong

Combining learning and search algorithms

MARCUS REMGÅRD
CHRISTIAN NILSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2022

Towards artificially playing the game of double pong
Combining learning and search algorithms
MARCUS REMGÅRD, CHRISTIAN NILSSON

© MARCUS REMGÅRD, CHRISTIAN NILSSON, 2022.

Supervisor: Jonas Hellgren, Volvo Autonomous Solutions
Examiner: Andreas Ekström, Dept. of Physics, Chalmers

Master's Thesis 2022
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Game frame of double pong constructed in Java.

Typeset in LaTeX

Towards artificially playing the game of double pong
Combining learning and search algorithms
MARCUS REMGÅRD, CHRISTIAN NILSSON
Department of Phyics
Chalmers University of Technology

Abstract

This thesis work investigates ways of enhancing search methods used in reinforcement learning by utilizing neural networks. The environment in which the methods are tested is the classical game of pong. Two primary networks were used called Deep value network (DVN) and Fail-state network (F-Network). The first network aids the search by estimating state-values, the second network is used to detect search paths that lead to certain losses. Regarding the search methods, two algorithms were implemented, Random rollouts and Monte Carlo tree search (MCTS). It was concluded that the combination of search together with DVN drastically outperforms plain search methods, especially in environments where deep searches are unfeasible and CPU resources are restricted. The F-Network did not show any promising results in our study, however, possible improvements are discussed.

Keywords: Neural networks, Q-learning, Monte Carlo tree search, Pong, Search algorithms, Reinforcement learning, Deep Q-learning.

Acknowledgements

First and foremost, we would like to thank Volvo Autonomous Solutions and our supervisor Jonas Hellgren for the opportunity to conduct our thesis work along side them. We are grateful for the continuous support and the fruitful discussions with Jonas.

Also, we would like to thank Andreas Ekström, from the Department of Physics at Chalmers, for his academic support and feedback.

Marcus Remgård & Christian Nilsson, Gothenburg, June 2022

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

DQN	Deep Q-learning
DVN	Deep Value Network
F-Network	Fail state network
G	Greedy one step search
MCTS	Monte Carlo Tree Search
MSE	Mean squared error
RR	Random rollout
UCB	Upper confidence bound
V-learning	Value learning

Contents

List of Figures

List of Tables

1
Introduction

This chapter introduces the reader to the problem, the scope of the thesis as well as the main research objectives.

1.1 Background

Volvo Autonomous Solutions (VAS) was created as a business unit in the Volvo group, with focus on developing autonomous systems for operating vehicles in confined areas, such as mines, as well as open road. Since decisions made from one vehicle will affect the operation of others, it is crucial to make the fleet of vehicles collaborate in a well-planned manner. Furthermore, the fleet must simultaneously act cost efficiently. Adding these layers of complexity results in optimisation problems with vast state spaces, which are often impossible to solve without applying some form of heuristic or simplification [1] . The major issue with this type of problems is that they suffer from the curse of dimensionality, i.e the state space grows exponentially with the number of state variables used to describe the problem [2].

The curse of dimensionality was first introduced by Bellman [3]. It is a recurrent difficulty in many areas of physics and statistics. For example, when describing a many-body quantum state, the information required for complete description of the system grows exponentially [4]. The curse of dimensionality also affect traditional nearest-neighbour searches [5], as well as more practical logistics problem encountered in every day life, for example flight-route planning [6].

In recent years, a lot of progress has been made in developing algorithms that efficiently combat the problem of large state spaces, and they are usually applied to different types of board games, such as the game of Go [7]. The Monte Carlo tree search (MCTS) algorithm acted as a starting point for many algorithms to come [8]. MCTS is an effective search algorithm by itself, but it was the combination of search together with reinforcement learning (RL) that eventually became the winning concept of the Dyna2 algorithm [9], which later turned into the famous Alpha Go agent developed by Deepmind [10].

Search algorithms alone can be surprisingly effective [11]. Yet, they lack the ability to learn. An RL algorithm, on the other hand, has the the ability to improve its performance with every experience it encounters. As of now, the combination of learning and search algorithms provides one of the most promising ways to combat

large state spaces in games [12].

In order to better understand the difficulties arising with vehicle autonomy, one first has to simplify the problem. As such, this thesis will revolve around the game of Pong. More specifically, deriving an RL algorithm that learns how to play Pong. Pong is a highly relevant starting environment, because it deals with coordinating objects - a fundamental concept when learning to control autonomous vehicles. The Pong environment is simple enough to make the main problem tangible, yet complex enough to experience the core issues of large state spaces.

1.2 Objectives

The aim of this thesis is to further investigate the synergism between RL and search algorithms. Exploring ways in which RL can enhance the search will be of special interest. The main objective is to research, implement and further improve upon state of the art methods for making the search more efficient and less computationally demanding. Moreover, ways in which the learning process itself can be accelerated by utilizing search will also be investigated.

In the Pong environment, this translates to finding a general solution to the game that can combat different degrees of difficulties, e.g., varying the number of balls, varying the speed of the rackets/balls or varying the size of the game frame. Additional features will increase the state space and be more demanding of computational efficiency, which in turn will require cost-effective searches in terms of CPU power.

1.3 Problem description

The game of double pong is played by controlling two rackets, as seen in black in figure 1.1. The game is lost whenever a ball hits the red zone underneath the rackets, or whenever two rackets collide with each other. If a ball hits a wall or one of the rackets, it will simply bounce off it. The main objective is to avoid loss for as long as possible by moving the rackets in a suitable manner. Naturally, the same rules apply to the game of single pong, with the only difference being utilizing only one racket.

Figure 1.1: The figure illustrates a snapshot of double pong. The two rackets are illustrated as two bold black surfaces.

The game frame will be represented by a rectangular shaped box, wherein the balls move without friction. The boundary conditions of the box will be modelled as unmovable walls, and the interaction between the ball and the wall will be completely elastic, i.e the initial speed of the ball will never change, only the sign of its velocity components. For example, when the ball hits the left wall in figure 1.1, the x-component of the velocity will change from $-x$ to x. The same dynamics will also govern the ball-racket interaction, i.e completely elastic bounce. When adding more balls to the game, the ball-ball interaction will be modelled as none-existing, meaning the balls will just pass through each other as objects without mass.

The governing physics of the ball dynamics may be simple and the deterministic, but the resulting motions can be surprisingly taxing to simulate searches on. Consider a simple version of single pong. With a window size of 400x400 pixels, there are roughly 160 000 possible locations for the position of the ball. At each location, the racket can be placed at approximately 400 different locations. With great simplification, the velocity vector of the ball can be assumed to have eight different directions. This results in a state space of size $160000 \cdot 400 \cdot 8 = 5.12 \cdot 10^8$. With three possible actions for every state (move the racket to the left, to the right or stand still), storing all of these states in memory would require $3 \cdot 5.12 \cdot 10^8 = 1.536 \cdot 10^9$ slots. Imagine then the state space growing even larger for double pong. Repeatedly iterating through all of these states would be time consuming. Any real life autonomous application will not only involve a much larger state space, but also demand operational speed, and so one can imagine the explosion in complexity that follows. It is therefore clear that there is a vital need for efficient algorithms that can operate in these large state spaces.

2
Theory

This chapter provides the necessary background, theory and notation for the methods presented in chapter 3. The essential concepts of RL are provided a more robust mathematical framework. Deep RL with neural networks is also discussed, as well as the idea of simulating a search.

2.1 Introduction to reinforcement learning

The idea of *learning from an environment* is a familiar concept to us humans. As children, we acquire new skills through simple trial and error. We make a guess, observe how the environment responds, and then we make a new guess slightly adjusted based on that response. We are also much aware of the concept of *delayed rewards* – giving up comfort now might result in an increased pleasure later. Balancing long-term gains with immediate satisfaction is constantly present throughout our lives.

The aforementioned two features summarize the very heart of reinforcement learning. Learning from an environment and delaying rewards are closely related to how both humans and animals learn in real life. As a matter of fact, the foundation of many reinforcement learning algorithms has sprung up from investigating biological systems. In 1927, the Russian physiologist Ivan Pavlov explored a phenomena called classical conditioning. He conducted experiments on dogs, and noticed how they started to drool whenever the person who normally fed them entered the room. Pavlov used the term "reinforce" to describe how a stimulus could strengthen certain behaviour among animals [13]. The same idea is used in reinforcement learning within the digital domain. By setting up the right rewarding system, we can make the computer drool even when there is no food around.

2.2 Terminology

Two important components of RL have already been mentioned: the *environment* and the object interacting with the environment, otherwise known as the *agent*. The basic idea of utilizing a *reward signal* has also been touched upon. In this section, an informal introduction to additional components characteristic for reinforcement learning will be given.

State, action and episode

A state is a description of the present moment. It contains all available information about the environment. For example, in the game of single pong a state could be as simple as a vector containing the positional coordinates of the ball and racket. However, it could also be a RGB color image of the game provided through raw pixel data.

Given a state, the agent can perform a number of actions. Returning to the game of single pong, an action could be moving the racket to left. Whenever the game is ended, the agent ends up in a so-called terminal state — a state where no further actions can be taken

Finally, we can informally define an episode as a sequence of states and actions that ultimately ends up in a terminal state.

Reward signal

The reward signal acts as the stimulus for the agent. It describes whether an event was good or bad by either punishing or rewarding the agent. The one and only objective for an agent is to accumulate as much reward as possible during an episode. Hence, the reward function is crucial for making the agent learn the desired behaviour. However, defining a good reward function is not entirely easy and requires caution. There is always the risk of incorporating hidden biases. For example, giving partial rewards during a game of chess to stimulate behaviour we believe is beneficial might actually hinder the agent from playing optimal in the long run [14].

Policy function

The policy function is a mapping from state to action. It tells the agent what action to perform in a certain state. The aim of reinforcement learning is to change this policy function through experience, and to finally arrive at a so-called optimal policy [15].

Value function

The reward function describes the immediate reward of an action, but the purpose of the value function is to estimate the long-term expected reward of an action or a state. In a sense, the value function describes how good or bad a particular state is in relation to the goal of the agent. Giving up an immediate reward now might result in collecting more rewards later, and the aim of the value function is to correctly capture these situations.

Summary

The concepts presented in the previous sections can be summarized in figure 2.1, which illustrates how the introduced terminology are interconnected.

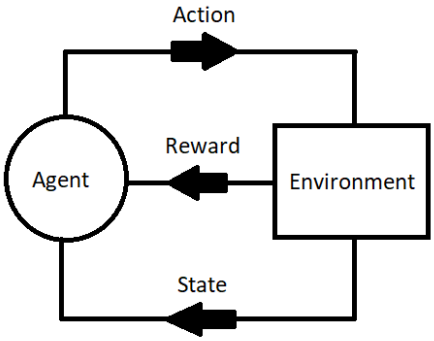

Figure 2.1: Schematic illustration expressing how actions, rewards and states connects the agent with the environment.

2.3 Mathematical framework

In this section, a robust mathematical framework for dealing with reinforcement learning will be introduced. The notation presented here will be used throughout the thesis.

Markov property

Whenever the current state of an environment fully encapsulates all historic events, the system is said to fulfill the Markov property [16]. A more formal way of interpreting this property is in terms of a memory less system, which satisfies the following condition

$$P(x_t|x_{t-1}, x_{t-2}, ..., x_0) = P(x_t|x_{t-1}) \tag{2.1}$$

with t being the timestep.

In other words, no additional information will improve the prediction of the future. This property is an important simplification utilized in many reinforcement learning problems. In order for it to be accurate, it is essential that the chosen state representation mimics this property well. For example, given the position, velocity and acceleration of an object in space, its future position can be accurately predicted without knowing its previous trajectory.

Markov Decision Processes

A Markov Decision Process (MDP) is defined by a 4-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ denotes a set of states, $\mathcal{A}$ denotes a set of actions, $\mathcal{P}$ is a set of transitional probabilities and $\mathcal{R}$ is a set of rewards [17]. Given a state s and action a, the

probability of ending up in state s' and receiving reward r is given by the transition probability

$$p(s', r|s, a) \tag{2.2}$$

As time progresses, the agent interacts with its environment. Assume that at timestep t the agent is in a state s_t and performs an action a_t according to some policy $\pi(a_t|s_t)$, i.e the probability of selecting action a_t when in state s_t. As the agent performs action a_t, it will receive a reward r_t and transition to a next state s_{t+1} according to eq.(2.2), otherwise known as the *model* of the environment. An episode is defined as a trajectory of a set of states, actions, rewards and next states that eventually end up in a terminal state s_T [15],

$$\text{episode} = \{s_1, a_1, r_1, s_2, ..., s_t, a_t, r_t, s_{t+1}, ..., s_T\} \tag{2.3}$$

with s_T being the terminal state in an episode.

Return

The return at time t is formally defined as

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.4}$$

where $\gamma \in (0, 1]$ is the discount factor, which deducts future rewards. Roughly speaking, it tells the agent how much weight it should give to rewards obtained far into the future [14]. If γ equals one, no discount is applied, meaning rewards collected at the end of the game is considered equally important as rewards collected near the present. In some circumstances, this is an undesired trait. Hence, fine tuning of this hyperparameter is usually necessary.

Bellman equation

Previously, the value function was introduced as the long-term expected future reward of a state. Now, the value $v_\pi(s)$ of a state s, based on following a certain policy π, can be properly defined as

$$v_\pi(s) = \mathbb{E}[R_t|s_t = s] \tag{2.5}$$

which using eq.(2.2) and eq.(2.4) can be reformulated as the Bellman equation [3],

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi\left(s'\right)\right] \tag{2.6}$$

with γ being the discount factor from equation 2.4.

Reinforcement learning revolves around finding an optimal policy $v_*(s)$ for a given problem [15], or more specifically, finding

$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S} \tag{2.7}$$

Policy evaluation and policy improvement

The Bellman equation, eq.(2.6), can be converted to an updating rule,

$$v_{k+1}(s) \leftarrow \sum_{a} \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_k\left(s'\right)\right]. \tag{2.8}$$

Eq.(2.8) provides an iterative updating scheme for evaluating an arbitrarily policy π, also known as policy evaluation. The sequence of $\{v_k\}$ can be shown to converge to v_π as k $\rightarrow \infty$ [15]. However, besides having a method for evaluating a policy, there is also a need for improving upon a policy, i.e finding ways of accumulating more rewards. As mentioned in section 2.2, the goal of reinforcement learning is to find an optimal policy. It can be shown that by following a so-called greedy policy π_{greedy} defined in eq.(2.9) , a new policy is obtained which is at least as good as the former policy π [15].

$$\pi_{greedy} = \arg\max_{a} \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right]. \tag{2.9}$$

Eq.(2.9) roughly implies that the agent should look one step into the future by simulating actions. Then, by evaluating all next states s' according to the current policy π, the agent should take the action that looks the most promising, i.e provides the highest value. This process is called policy improvement.

Policy iteration

By combining policy evaluation and policy improvement, an algorithm called policy iteration is obtained. Initially, an evaluation of an arbitrary policy is performed using eq.(2.8). Then, this policy is improved by acting greedy with respect to this evaluation. Similarly, this improved policy is evaluated and then improved again by acting greedy with respect to its evaluation. By repeating this pattern according to figure 2.2, it can be proven that the algorithm eventually converges to an optimal policy [15].

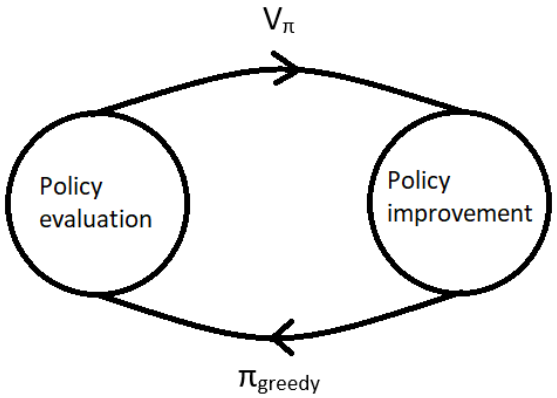

Figure 2.2: The figure illustrates a schematic view of policy iteration, a process which converges to an optimal policy by combing policy evaluation with policy improvement in an iterative manner.

These two iterations are the founding block of most RL algorithms. However, there are loads of variants and they can be intertwined in efficient ways, with one of those ways being Q-learning, which will be discussed next.

2.4 Q-learning

Up until now, a *model* of the environment has been required in order to find the optimal value function via policy iteration. More precisely, the model has provided the transition probabilities in eq.(2.2). However, in many situations, the transition probabilities are unknown, i.e a model of the environment is not given beforehand. Instead, the agent will have to rely on experience and learn how the environment works by trial and error. This is where Q-learning comes in handy [18].

Previously, a state-value function $v(s)$ was used to find the optimal policy for a given problem. Q-learning, on the other hand, deals with action-value functions $Q(s, a)$, i.e the expected long-term future rewards for taking a certain action in a given state. The Bellman update equation from eq.(2.8) now takes the form

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{2.10}$$

where γ is the discount factor from Eq.(2.4), s' and a' is the succeeding state and action, and α is a hyperparameter called learning rate, which determines the step size when moving towards a local minimum [19].

A simple Q-learning problem

In order to better understand all the concepts discussed so far, a simple Q-learning problem consisting of connected nodes presented in figure 2.3 will be solved. The nodes represent states, the arrows represent actions and the grey states are terminal states. For each action, a reward can be seen. For example, transitioning from state (5) to state (6) results in a reward of 10. On the other hand, a transition from state (8) to (9) results in a reward of -10. Also, for state (1),(2),(4),(5),(7) and (8), the action that returns to the same state will be defined as $a = 0$ giving a reward of $r = -1$, while the action that transitions to a new state will be defined as $a = 1$ giving a reward of $r = 0$. Furthermore, in state (3), a transition to state (4) will be defined as $a = 0$, while a transition to state (7) will be defined as $a = 1$.

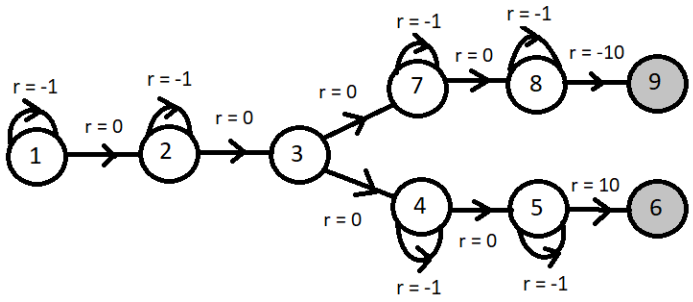

Figure 2.3: The figure illustrates the environment for a simple node problem. The circles represent states, the arrows represent the possible actions from each state and the grey states represent terminal states. For each action, a reward r can be seen.

By setting both the discount factor γ and the learning rate α to one, the Q-learning algorithm in eq.(2.10) simplifies to

$$Q(s,a) \leftarrow r + \max_{a'} Q(s',a'). \tag{2.11}$$

Since a particular action-value is updated by looking one step forward, it is convenient (yet not necessary) to start near a terminal state, because all action-values in a terminal state are zero by definition [15]. All other action-values will initially be set to zero as well. Applying eq.(2.11) to $Q(s=8,a=1)$ results in,

$$Q(s=8,a=1) \leftarrow -10 + 0 = -10. \tag{2.12}$$

However, when evaluating $a=0$ from state (8), the first iteration results in

$$Q(s=8,a=0) \leftarrow -1 + \max_{a'} Q(s'=8,a') = -1 + max(0,-10) = -1 \tag{2.13}$$

The second iteration results in

$$Q(s=8,a=0) \leftarrow -1 + \max_{a'} Q(s'=8,a') = -1 + max(-1,-10) = -2 \tag{2.14}$$

And by the 10th iteration, the following action-value is obtained

$$Q(s=8,a=0) \leftarrow -1 + \max_{a'} Q(s'=8,a') = -1 + max(-10,-10) = -11 \tag{2.15}$$

This action-value has now converged, because during an additional 11th iteration, the following action-value is obtained,

$$Q(s=8,a=0) \leftarrow -1 + \max_{a'} Q(s'=8,a') = -1 + max(-10,-11) = -11 \tag{2.16}$$

By now, the action-value table looks as in table 2.1

	s = 1	s = 2	s = 3	s = 4	s = 5	s = 6	s = 7	s = 8	s = 9
a = 0	0	0	0	0	0	0	0	-11	0
a = 1	0	0	0	0	0	0	0	-10	0

Table 2.1: Action-values for the simple node problem.

Analogously, the action-values for state (7) will be updated to

$$Q(s = 7, a = 1) \leftarrow 0 + \max_{a'} Q(s' = 8, a') = max(-10, -11) = -10 \tag{2.17}$$

and

$$Q(s = 7, a = 0) \leftarrow -1 + \max_{a'} Q(s' = 7, a') = -1 + max(0, -10) = -1 \tag{2.18}$$

Once again, by the 10th iteration, the action-value $Q(s = 7, a = 0)$ will have converged to -11, resulting in table 2.3.

	s = 1	s = 2	s = 3	s = 4	s = 5	s = 6	s = 7	s = 8	s = 9
a = 0	0	0	0	0	0	0	-11	-11	0
a = 1	0	0	0	0	0	0	-10	-10	0

Table 2.2: Action-values for the simple node problem.

Similarly, the action-values for state (5) will be updated according to

$$Q(s = 5, a = 1) \leftarrow 10 + 0 = 10 \tag{2.19}$$

and

$$Q(s = 5, a = 0) \leftarrow -1 + \max_{a'} Q(s' = 5, a') = -1 + max(0, 10) = 9 \tag{2.20}$$

Eventually, by continuing in this manner, the following action-value table will be obtained,

	s = 1	s = 2	s = 3	s = 4	s = 5	s = 6	s = 7	s = 8	s = 9
a = 0	9	9	10	9	0	-11	-11	-11	0
a = 1	10	10	-10	10	0	-10	-10	-10	0

Table 2.3: Action-values for the simple Q-value problem.

Exploration and exploitation

The simple node problem presented in section 2.4 is much easier solved by utilizing a computer. An agent will take steps in the environment, initially randomly, but as the Q-table is updated with values, the agent will receive better and better estimates of its action-values. The agent always has two choices when faced an arbitrary state. Either take an action randomly, or perform a table look up and act greedy

with respect to the current action-values. The agent will have to balance exploiting already obtained knowledge with exploring new paths. Usually, this balance is regulated by a term called epsilon (ϵ), which is a percentage point controlling how often actions are taken randomly. As long as epsilon is greater than zero, the action-values are guaranteed to converge [15].

2.5 Value learning

Value learning (V-learning) is very much similar to Q-learning. The algorithm is based on the same updating algorithm, see eq.(2.21), but instead of learning action-values the algorithm learns state-values.

$$V(s) \leftarrow V(s) + \alpha[r(a) + \gamma \max_a V(T(s,a)) - V(s)] \tag{2.21}$$

where $T(s,a)$ is the transition operator, which transitions a state s to s' through action a.

V-learning on the simple node problem

V-learning can also be applied to the simple node problem in section 2.4. As before, the value of terminal states (6) and (9) are zero be definition. The values of all other states will be initialized to -1 to demonstrate the fact that initial values can be set arbitrarily. All other assumptions holds. Once again, for convenience, the updating algorithm in eq.(2.21) will be applied initially to state (5).

$$V(s=8) \leftarrow r(a) + \max_a V(T(s,a)) = r(a) + max(-1,0) = -10 + 0 = -10 \tag{2.22}$$

which results in the following V-table,

	s = 1	s = 2	s= 3	s = 4	s = 5	s = 6	s = 7	s = 8	s = 9
V(s)	-1	-1	-1	-1	-1	0	-1	-10	0

Table 2.4: State-values for the simple node problem.

In similar fashion, state (6) and (7) will also converge to the value of -10. Proceeding to state (5), the following update is obtained,

$$V(s=5) \leftarrow r(a) + \max_a V(T(s,a)) = r(a) + max(-1,0) = 10 + 0 = 10 \tag{2.23}$$

The final V-table can be seen in table 2.5.

	s = 1	s = 2	s= 3	s = 4	s = 5	s = 6	s = 7	s = 8	s = 9
V(s)	10	10	10	10	10	0	-10	-10	0

Table 2.5: State-values for the simple node problem.

An interesting comment on table 2.5 is the value of state (1), which is 10, even though none of its actions actually results in a reward of 10. However, from this state, there is an available path that leads to state (6), where the actual reward of 10 is received. Another comment is the value of state (7), which is -10. This is due to the fact that no matter what action is taken in this state, state (6) can never be reached.

2.6 Function approximation

So far, the Q-learning algorithm has been relying on table look ups for updating and storing action-values. This worked great on the simple node problem in section 2.4, because the state space was small. However, as the state space grows, utilizing tables for storing action-values will simply not be feasible, as discussed in section 1.3. Therefore, there is a vital need for some type of approximation. A common way of approximating action-values is to implement a neural network. In this way, the agent can be trained to recognize similar situations in the state space and thereby approximate the action-values. This type of generalization makes the Q-learning procedure much less time consuming and suitable for more complex problems. The combination of Q-learning and neural networks is often referred to as Deep Q-learning [20].

2.6.1 Feedforward neural networks

Deep Q-learning uses neural networks as a function approximator for the table look up. The simplest and also the first type of artificial neural networks are so called feedforward neural networks. The main principle of this type of neural networks is that information is only travelling in one direction, forward, thereby its name feedforward neural network. The network architecture includes an input layer, an arbitrary amount of hidden layers with hidden neurons and an output layer. The idea is that the input data travel through the network layers and its neurons with different activation functions and weights in order to produce a final output from a output layer with a specific activation function. The activation functions of the network aim to transform the weighted sum input to each neuron in order to produce a final output that is feasible with the task at hand, for more complex problems they are often chosen to be nonlinear functions [21].

When the final output is produced the loss is computed by applying a so called loss function. A common choice of loss function for a neural network that uses several samples of input data at once, i.e batches, is the mean squared error (MSE). On the other hand, a binary classification neural network utilizes the binary crossentropy loss function in order to compute the loss. Furthermore, the loss is propagated backwards through the neural network in order to update the weights of the neurons by using so called gradient descent. The goal of the backpropagation is to calculate the partial derivatives of the loss function with respect to the weights and then move in negative direction of these gradients in order to approach the minima of the loss function. After a sufficient amount of iterations, the weights of the network should

have converged, and the training of the network is complete [21].

2.6.2 Deep Q-learning

In classical Q-learning, the Q-values are updated using the Bellman update equation (2.10). For deep Q-learning, the same principle can be utilized. The difference is that the Q-values are approximated by a neural network. Since the idea is to get the Bellman error, $r + \gamma \max_{a'} Q(s', a') - Q(s, a)$, to converge to zero the weights of the network is updated by minimizing the Bellman error. A common choice of loss function, when dealing with neural networks, is the MSE. By using this loss function it is possible to update the weights of the network with respect to several input data each time, since the average is taken. Furthermore, the gradient descent algorithm is a common method for updating the weights of the network. In this method, the weights are updated in the negative direction of the gradient of the loss function with respect to the weights, in order to move towards the minima of the loss function. Then, the update procedure of the weights can be mathematically formulated as

$$w' = w - \alpha \nabla_w \frac{1}{N} \sum_{i}^{N} \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)^2 \tag{2.24}$$

where the mean squared Bellman error is used. N is the number of data inputs used when fitting the model and $\alpha \in (0, 1)$ is the learning rate of the neural network [20].

Target network

An important aspect of the DQN algorithm is the incorporation of a second network, with the same architecture as the original network, called target network. If one were to update both $Q(s', a')$ and $Q(s, a)$ at the same training iteration with one network it could lead to diverging Q-values. By utilizing a target network, in order to freeze the values $Q(s', a')$ during a number of iterations, one can stabilize the training procedure. The incorporation of the target network can be compared to freezing a moving target when trying to hit bullseye [14].

Hyperparameters

Hyperparameters that need to be determined before initializing the training algorithm is the learning rate α, the number of training episodes, the size of the replay buffer where the transitions are stored and the update frequency of the target network. Also the action policy of the training agent needs to be determined. A common approach is the epsilon greedy algorithm where at each iteration a random action is chosen with some probability p and the optimal action, suggested by the network, is chosen with probability $1 - p$. However, since the network is gradually improving for each gradient descent it is common practice to use some sort of epsilon decay during the training procedure. The principle is simple, in the beginning of the training when the network is somewhat useless a random action is chosen with a high probability and in the end when the network is well-trained a random action is

chosen with a low probability. One can, for example, choose to linearly decrease the epsilon value, i.e. the probability for a random action, with the number of training episodes [14].

Algorithm

In order to initialize the deep Q-learning network (DQN) the network is initialized with random weights. Also, the target network is initialized with random weights. First, only random actions is taken, in the environment, in order to fill up the replay buffer with transitions. Then one can initialize the first training iteration by choosing a random state in the state space. Then an action a is chosen with epsilon greedy and a reward r for that action is collected from the environment model. Then the transition, consisting of the previous state s, the action a, the reward r and the new state s', is saved in the replay buffer. Then a minibatch of transitions, of some predetermined size, is randomly selected from the replay buffer in order to perform gradient descent and update the weights of the network. Eventually, after some specific number of iterations, the target network will be updated as well [14].

Pseudo code: DQN

Algorithm 1: DQN

Initialize replay memory D
Initialize Q with random weights θ
Initialize Q_{target} with weights $\theta' = \theta$
while *i<nrTrainingEpisodes* **do**
  s $\leftarrow$ $state_{start}$
  while *isNotFailure(s)* **do**
    $a_{opt} \leftarrow \max_a Q(s, a)$
    p=Random(0,1);
    a=$\begin{cases} a_{rand} & \text{if } p < \epsilon \\ a_{opt} & \text{else} \end{cases}$
    $s' = T(s, a)$
    $r = R(s, a)$
    Store transition (s,a,r,s') in replay memory D
    $s = s'$
    EXPERIENCE REPLAY
    Sample random minibatch of transitions (s,a,r,s') from D
    set t =$\begin{cases} r & \text{if episode terminates at s'} \\ r+\gamma \max_{a'} Q_{target}(s', a'; \theta') & \text{otherwise} \end{cases}$
    Perform gradient descent step on $(t - Q(s, a; \theta))^2$ w.r.t θ
    //Periodic update of target network (for stability)
    Every C steps reset $Q_{target} = Q$
  end
end

Experience replay buffer

It is of outermost importance to get the training procedure to result in an agent which is not overfitted to the training data and is able to generalize to new data. One reason for overfitting is that there exists some bias in the training data. E.g. the training data is not uniformly distributed over the state space.

However, in reinforcement learning, a common issue is so called sequential bias/correlation when training on sequential data. In order to reduce the instability that emerges from the training on highly correlated sequential data one can make use of a so called experience replay buffer. One can significantly reduce the sequential correlation by storing each transition in this replay buffer and then randomly sample a minibatch, from the replay buffer, for each gradient descent step. By initializing the replay buffer with some specific size, often very large, and constantly adding a new transition for each iteration the older transitions will be pushed further back in the replay buffer and eventually be removed [22].

2.7 Binary classification

A binary classification problem describes a situation where one seeks determine which of two categories an example object belongs to. A binary classification model should, given some features, be able to predict which class or category an example belongs to. For example, if a well-performing binary classification model is given a dataset of images of different humans and categories (labels) man and woman it should be able to determine if an image shows a man or a woman. It is common practice for a classification model to assign probabilities to each class instead of outputing a definite decision. In machine learning this is achieved by applying the Sigmoid activation function in the output layer [21]. It is highly relevant to ponder over the final decision threshold when developing a binary classification model.

A binary classification neural network utilizes the same principles as discussed in 2.6.1. The final decision threshold for the neural network can be obtained by studying the so called Receiver operating characteristic curve, often referred to as ROC curve. More precisely, the ROC curve is the graph obtained when plotting the true positive rate versus the false positive rate for a specific threshold. These measures are obtained by applying the trained binary classification network to a batch of input data and then comparing the output of the network to the actual output. Then the idea is to choose the positive class threshold that corresponds to the maximum difference between the true positive rate and the false positive rate. This threshold should correspond to the optimal threshold for the binary classification task at hand [23].

2.8 Search algorithms

A search algorithm is an algorithm which explores the state space of an environment by performing simulations. The search can implement many different strategies, ranging from a simple rollout (section 2.8.2) to conducting a complete search tree (section 2.8.4). The choice of search algorithm is largely dependent on the complexity of the problem and the computational resources available. The search is dependent on both the depth of the search, i.e how many steps from the current state one simulates, and the number of overall simulations one wishes to perform during a search. A deeper search will result in more accurate value estimations, but it will also require heftier computational power [11].

2.8.1 Greedy one step search

Naturally, the most simple form of search is when the agent only simulates one step ahead. Hereinafter, this will be called the greedy one-step search algorithm. The algorithm simply chooses the most optimal action in each state, suggested by a trained agent, i.e a neural network. This kind of approach requires a well-trained network in order to be effective, since the state-value approximations obtained by the network have to accurate. It is useful since it demonstrates the pure strength of the trained agent.

Pseudo code: greedy one step search

Algorithm 2: Greedy one step search

$N(s)$: Pre-trained neural network

bestAction$\leftarrow$N(s);
s'=environment.step(bestAction,s);

2.8.2 Rollout

A rollout is an important concept which refers to rolling out simulations from the current state [15]. A rollout performs repeated simulations starting from a so-called root node by following a certain policy π. During every simulation, the accumulated reward is collected. By averaging over many simulations, an action value is obtained for each action at the root node [15]. Once the simulations are done and the action values have been utilized to perform a real action, they are immediately discarded. Despite the simplistic nature of a rollout, it is surprisingly effective [11]. Naturally, a rollout can also estimate state-values following the same principles, i.e by rolling out simulations from all successive root node states. Figure 2.4 describes the rollout in a schematic way, with a search horizon D.

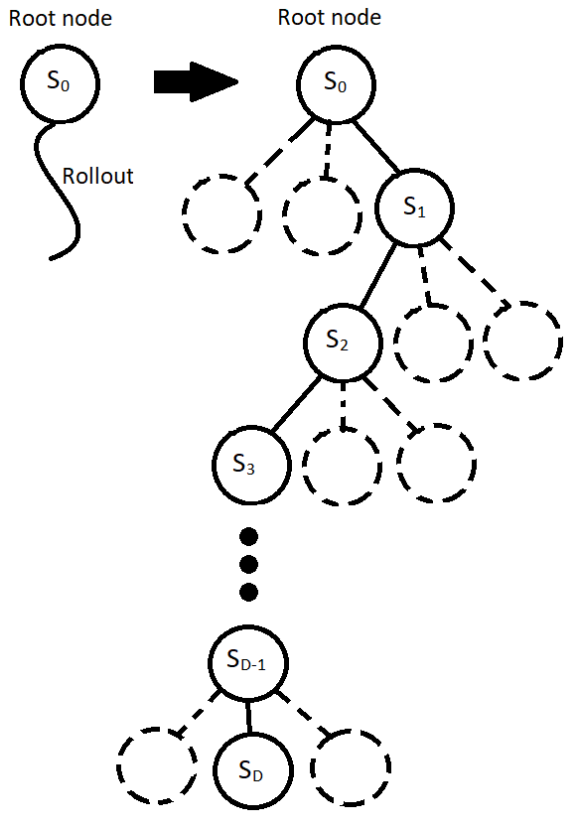

Figure 2.4: The figure illustrates the rollout process. In this case, the action space is three and the rollout search horizon is D. The rollout policy determines the action choice at every state S. By performing repeated simulations and collecting the rewards along the way, value estimates of the root node actions are obtained.

2.8.3 Random rollout

A Random rollout is from here on now defined as a rollout following a random policy. This algorithm simulates random actions in order to make a final real decision from the root node. The width of the search tree depends on the size of the action space. The search depth depends on the search horizon, i.e how far down, in the tree, the algorithm should perform random actions before switching to a new simulation. The search horizon depends on the problem at hand, e.g if there is any time limit on when to make the final decision. In order to favour more exploration one should be careful about choosing a too large search horizon. When the time is up the algorithm chooses the action from the root node that led to the most favourable end result. The end result can, for example, be seen as the cumulative reward for a path.

Pseudo code: random rollout

Algorithm 3: Random Forest search

while *timeNotFinished* **do**
  $s \leftarrow s_0$;
  return = 0;
  step = 0;
  a_0;
  while *step < searchHorizon AND isNotFailure(s)* **do**
    a_{rand} = getRandomAction(s);
    If (step==0) **then** $\{a_0 = a_{rand}\}$ //store first action
    $s', r \leftarrow$ environment.step(s, a_{rand});
    return $\leftarrow$ return + r; //add reward
    $s \leftarrow s'$
    step++;
  end
  If (return>bestReturn) **then** $\{a_{opt} = a_0\}$
end
$a_{final} = a_{opt}$

2.8.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a search algorithm where a search tree is built and expanded node-by-node according to the subsequent predicted outcomes for the sequentially created and newly found leaf nodes. Usually, a specific node corresponds to a state in the state space that is being considered. When a sufficient amount of searches in the tree/iterations have been performed from the root node, trade off between computational cost and the quality of the choices made by the agent, 1600 in AlphaGo, you make a choice of action from the root node. For competitive play one should make the final choice deterministically and for exploratory play one should make the final choice stochastically (based on the number of times the different actions have been performed) [24].

In the MCTS algorithm there will be 4 basic steps in each iteration.

1. Selection.
2. Expansion.
3. Prediction, e.g. an output from a pre-trained neural network.
4. Back propagation.

In order to describe these steps a set of parameters, that is saved in each edge between two nodes, has to be defined.

$$\{N(s,a), W(s,a), Q(s,a), P(s,a)\} \tag{2.25}$$

Here, $N(s,a)$ corresponds to the number of times MCTS has made the choice to perform the action a from state s. The parameter $W(s,a)$ is the total value of the action, i.e. the sum of the previously obtained action values v, as obtained in the prediction phase and $Q(s,a) = \frac{W}{N}$ is its mean value and is a measure of how good

the action is. The variable $P(s, a)$ specifies the prior probability of an action a from a specific state s and is obtained as an output from the neural network.

The algorithm starts off with the selection phase. Here, actions are chosen sequentially, starting from the root node s_0, until a leaf node s_L is reached. Different policies can be used in order to select an action in the tree. However, a common choice of selection policy is to choose the action that maximizes the upper confidence bound value (UCB)

$$\text{UCB} = Q_i + U = Q_i + c\sqrt{\frac{\log N_p}{n_i}} \tag{2.26}$$

where n_i is the number of times the child node i has been visited and N_p is the number of times the parent node has been visited. The constant c can here be seen as an exploration parameter and is often chosen to be $\sqrt{2}$. This can of course be changed depending on personal preference regarding exploration vs exploitation. Early in the simulation the second term in UCB dominates (more exploration) but later on the first term dominates (less exploration). If a neural network is utilized, the prior probability of a move P can also be incorporated in the second term of the UCB. Then, an intuitive interpretation of the second term U is that it increases if an action has not been explored much, relative to the other actions, or if the prior probability of the action is high. A special case is when two or more actions happens to have the same UCB, which for example is the case when two or more child nodes have not been visited before. Then, these possible actions have an infinite UCB, since $n_i = 0$ for these child nodes. In order to keep things simple one can simply choose to perform the first action in line (i.e. first from the left in the tree) [8].

When a leaf node s_L has been reached in the selection phase an expansion of this node, by adding the number of edges that correspond to the size of the action space, is performed. Also, for each newly created edge, the parameters $N(s_L, a)$, $W(s_L, a)$ and $Q(s_L, a)$ are set to zero. Simultaneously, the game state of the leaf node is passed into the neural network, which outputs the action value v for the current state s_L and the move probabilities P which are attached to the possible actions (egdes) from the leaf node. The action value v is an estimate of how good the current state is in terms of the probability of avoiding a game loss, in the near future, from the current state.

The final step is then to update each edge that was traversed in order to get to the leaf node s_L i.e. so called back propagation of the action value v. Each edge is updated in the following way

$$\begin{aligned} N &\to N + 1 \\ W &\to W + v \\ Q &= W/N. \end{aligned}$$

That is the end of an iteration. This is repeated for a large amount of iterations in order to obtain good estimates, or until the time is up. Finally a choice of action from the root node s_0 is made. One can either choose the action with the largest

value Q or simply the action with the largest value N, if we decide to make our choice based on competitive play [24]. After the actual move has been made the following is done before the next series of iterations

1. Let the new state, that was attained by choosing the preferred action above, become the new root node s_0.
2. The sub-tree from the chosen move is retained for calculating subsequent moves.
3. The rest of the tree is discarded.

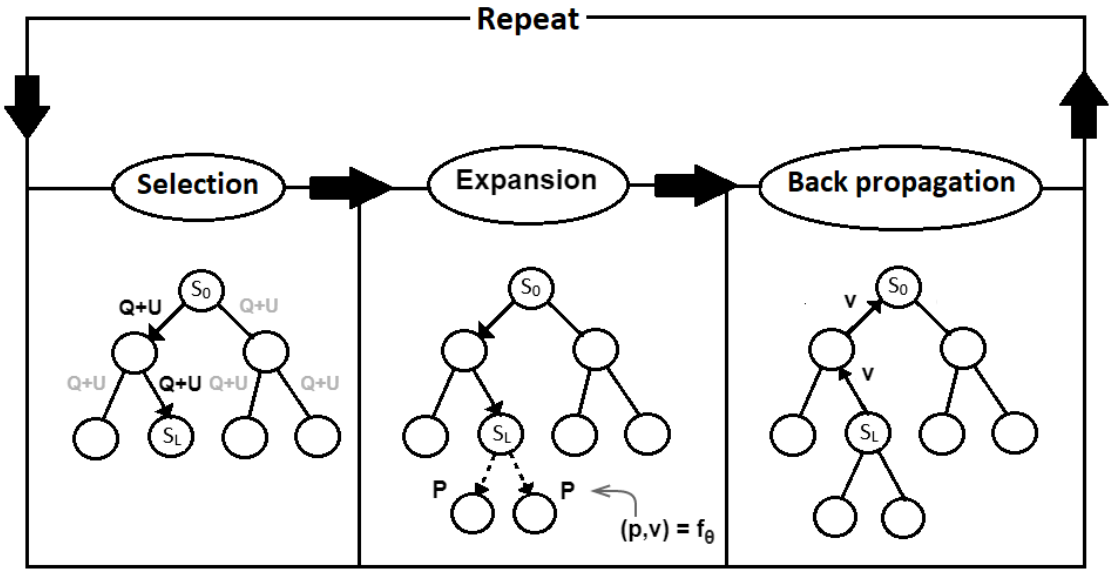

Figure 2.5: The figure illustrates the MCTS algorithm. For simplicity only two actions per state are included in this figure. Here, f_θ represents the output of the neural network. This whole procedure is repeated until a terminal state of the game is reached.

3
Methods

This chapter describes and motivates the specific implementations used when artificially learning to play the game of pong. Initially, a brief description of the pong environment will be given. Then, the implementation of learning and search algorithms will be explained in detail.

3.1 Overview

Finding the optimal combination of learning and search algorithms, in the Pong environment, is the main purpose of this thesis. The following learning and search methods will be considered

LEARNING\SEARCH	Greedy one step search (G)	Random rollout (RR)	Monte Carlo tree search (MCTS)
Deep Value Network (DVN)	DVN + G	DVN + RR	DVN + MCTS

3.2 State and actions in pong

A state in the game of pong is represented by a vector containing the x- and y-coordinates of the ball, the x- and y-velocities of the ball as well as the x-coordinate of the racket. This choice of state representation is motivated by the desired Markov Property described in section 2.3

The actions available in all states of the game are either moving the racket to left, to the right or stand still. Movement can occur in every frame of the game, with the speed presented in table 3.1.

3.3 The pong environment

Two different types of environments for the game of pong is used, differing only in the size of the game frame and the width of the racket. The reason for this is to examine whether the larger state space is influencing the training time of the learning algorithms. Forthcoming, the two environments will be referred to as the small and large environment. The specifics of the environments can be seen in table 3.1.

	Small environment	Large environment
Frame size	380x180 pixels	1000x800 pixels
Racket width	50 pixels	120 pixels
Speed of racket	10 pixels/frame	10 pixels/frame

Table 3.1: Environmental specifics for the two different pong environments; small and large.

The speed of the ball is constant during a game but initially randomized between a fixed set of velocity vectors; the x-component ranges from [-5,5] pixels/frame and the y-component ranges from [-1,-5] pixels/frame, i.e the ball is always moving away from the racket in the beginning of a game (see figure 1.1 for definition of coordinate system). Also, the initial position of the ball is randomly generated.

3.4 Learning algorithms

In this thesis, the main purpose of the learning algorithm is to guide the search algorithm. The guidance can be done in a number of different ways depending on the specific network and search algorithm used. Two different neural networks is presented here, the F-network and the Deep Value network (DVN). The F-network recognizes states that lead to loss and can therefore abort search simulations early, thereby saving valuable CPU resources. DVN, on the other hand, guides the search in the following two ways:

1. DVN estimates state-values during the final step of a rollout, i.e when the simulation reaches the search horizon.
2. DVN aids MCTS by presenting which part of the search tree that looks the most promising to expand on.

Learning algorithms, as opposed to search algorithms, are improved by experience.

3.4.1 Deep Value network (DVN)

DVN is a modified version of the classical Deep Q-Network described in section 2.6. The main difference is that DVN is trained to learn state-values rather than action-values, much like V-learning presented in section 2.5. DVN provides a more flexible learning algorithm. For example, DVN can be used in problems where the number of actions varies between different states.

Reward function

The self-made reward function is defined as,

$$r = \begin{cases} 100 & \text{if hit} \\ -d(x_r, x_b)100/L & \text{if miss} \\ 0 & \text{Otherwise} \end{cases} \tag{3.1}$$

where d is the distance between two points, x_r is the x-coordinate of the racket center, x_b is the x-coordinate of the ball center and L is the width of the game frame. A hit is defined by the ball hitting the racket, and a miss is defined by a game loss. Thus, the reward for missing is proportional to how large the miss is.

Network architecture and hyperparameters

The neural network consist of one hidden layer with 64 hidden neurons, presented in figure 3.1.

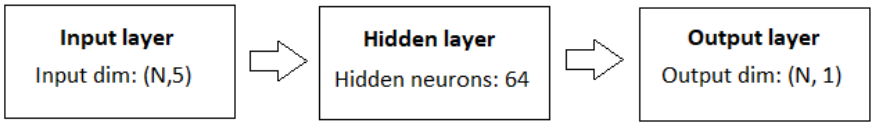

Figure 3.1: Schematic figure illustrating the architecture of the network, where N is the variable batch size.

Table 3.2 presents the hyperparameters, motivated by running a hyperparameter optimization, see Appendix A. The learning rate (α), batch size, synchronization frequency and experience replay size was defined in section 2.6.2, whereas the discount factor was introduced in Eq.(2.4).

Hyperparameter	
Learning rate (α)	0.001
Batch size	500
Synchronization frequency	100
Discount factor (γ)	0.95
Experience replay size	100000

Table 3.2: DVN hyperparameters for the small environment.

Table 3.3 presents the hyperparameters used for the large environment.

Hyperparameter	
Learning rate (α)	0.0001
Batch size	500
Synchronization frequency	8
Discount factor (γ)	0.95
Experience replay size	100000

Table 3.3: DVN hyperparameters for the large environment

The method used for normalizing the input data to the network is the common Min-max technique [25]. During training, whenever the agent manages to strike 15 hits in a row, the episode is ended. This reduces the training time as well as limits long sessions of repeated ball trajectories from filling up the replay buffer (section 2.6.2) with only successful runs. Lastly, a linear epsilon decay scheme is implemented (section 2.6.2),

$$\epsilon_{k+1} = \begin{cases} \epsilon_k - 1.5/E & \text{if } \epsilon > 0.05 \\ \epsilon_k & \text{otherwise} \end{cases} \tag{3.2}$$

where k is the current episode, $\epsilon_0 = 1$ and E is the total number of training episodes.

Pseudo code: DVN

Below is the complete pseudo code for training the DVN.

Algorithm 4: DVN

Initialize replay memory D
Initialize V with random weights θ
Initialize V_{target} with weights $\theta' = \theta$
while *i<nrTrainingEpisodes* **do**
  s $\leftarrow$ $state_{start}$
  while *isNotFailure(s)* **do**
    $action_{opt} \leftarrow$ ArbitrarySearchMethod(s_0)
    p=Random(0,1);
    a=$\begin{cases} a_{rand} & \text{if } p < \epsilon \\ a_{opt} & \text{else} \end{cases}$
    $s' = T(s, a)$
    $r = R(s, a)$
    Store transition (s,a,r,s') in replay memory D
    $s = s'$
    EXPERIENCE REPLAY
    Sample random minibatch of transitions (s,a,r,s') from D
    set t =$\begin{cases} r & \text{if episode terminates at s'} \\ r+\gamma \max_a V_{target}(s'; \theta') & \text{otherwise} \end{cases}$
    Perform gradient descent step on $(t - V(s; \theta))^2$ w.r.t θ
    //Periodic update of target network (for stability)
    Every C steps reset $V_{target} = V$
  end
end

3.4.2 F-Network

In order to speed up the search in the pong environment, a binary classifier is implemented. Throughout this report, this binary classifier is referred to as the F-Network. The purpose of this binary classifier is to remove actions that lead to non-feasible states, i.e fail states. Hence, the F-Network limits the search to areas of interest by removing search tree branches of certain failure. A fail state could both

be a direct terminal state, i.e. the ball hits the ground, or an eventual inevitable fail state. The fact that a state must be an inevitable fail state, i.e. it should not be possible even for an optimal policy to save the ball, is what makes the implementation problematic. Without this restriction, of it being an inevitable fail state, the binary classifier would start to classify states that are possible to save as fail states and that is not desirable.

The training algorithm will be based on the DVN architecture and thereby utilize the principles presented in 2.6 and 2.7. However, there are a few key differences. In order to explore as many states as possible, a random action will be taken with 80% probability during training. For the remaining 20 % of the time, an already trained DVN is used to pick action. Also, in order to produce labels t, the targets, the following logic is used

$$\text{set t} = \begin{cases} \text{false} & \text{if } \exists a : T(s,a)! = TS \text{ and } F_{\text{target}}\left(T(s,a)\right)! = 1 \\ \text{true} & \text{Otherwise} \end{cases} \tag{3.3}$$

where true means fail state. Eq. (3.3) states that if there exists an action a in the current state s that does not lead to a terminal or eventual fail state, it should not be classified as a fail state.

The same neural network and hyperparameters as in section 3.4.1 is used when training the F-Network.

Pseudo code: F-Network

The training algorithm in its entirety can be seen below.

Algorithm 5: F-Network

Initialize replay memory D
Initialize F with random weights θ
Initialize F_{target} with weights $\theta' = \theta$
while *i<nrTrainingEpisodes* **do**
  s $\leftarrow state_{start}$
  while *isNotFailure(s)* **do**
    $action_{opt} \leftarrow$ GreedyOneStepSearch(s_0)
    p=Random(0,1);
    a=$\begin{cases} a_{rand} & \text{if } p < 0.8 \\ a_{opt} & \text{else} \end{cases}$
    $s' = T(s, a)$
    $r = R(s, a)$
    Store transition s in replay memory D
    $s = s'$
    EXPERIENCE REPLAY F-NETWORK
    Sample random minibatch of states s from D
    set t =$\begin{cases} false & if\ \exists a : T(s,a)! = TS\ and\ F_{target}(T(s,a))! = 1 \\ true & \text{Otherwise} \end{cases}$

    loss=binaryCrossEntropy
    Perform gradient descent step on loss$(t, F(s; \theta))$ w.r.t θ
    //Periodic update of target network (for stability)
    Every C steps reset F_{target} = F
  end
end

3.5 Search

In this section, the pong implementation of the Random rollout presented in section 2.8.2 is described. The search is not aided by a learning algorithm, and will serve as a base line when evaluating potential improvements. The primary difference from the pseudo code in section 2.8.2 is that a value estimate for each root node action is continuously updated, instead of simply storing the optimal action. Besides giving additional information, it also enables frictionless integrating of learning algorithms

later on.

Algorithm 6: Random rollout

$q_S(s_0, a) = 0 \; \forall \; a$
while *cpuTimeNotFinished* **do**
  $s \leftarrow s_0$;
  return = 0;
  step = 0;
  a_0;
  while *step* $<$ *searchHorizon AND isNotFailure(s)* **do**
    a_{rand} = getRandomAction(s, **a**);
    If (step==0) **then** $\{a_0 = a_{rand}\}$ //store first action
    $s', r \leftarrow$ environment.step(s, a_{rand});
    return $\leftarrow$ return + $\gamma^{step} \cdot r$;
    $s \leftarrow s'$
    step++;
  end
  $q_S(s_0, a_0) = q_S(s_0, a_0) + \alpha \cdot (return - q_S(s_0, a_0))$; where $\alpha = \frac{1}{\text{step}}$
end
$a_{optimal} = \max_a(q_S(s_0, a))$

3.6 Combining learning and search

During a search, one has to take into account that there is a finite amount of CPU time available, since the ball will inevitably move to a new position. Therefore, it is crucial that the search is able to perform under restricted time and limited search depth. This can be achieved by combining the learning algorithms together with the search algorithms. Both MCTS and the Random rollout will be considered in combination with the learning algorithms presented earlier in section 3.4.

3.6.1 DVN with greedy one step search

As discussed in section 2.8.1, the greedy one step search is the simplest form of search, and it will serve as the evaluation baseline for the DVN. The algorithm simply performs a one step simulation and evaluates each state with DVN. The action that leads to the highest state-value is then chosen.

Pseudo code

Algorithm 7: DVN with greedy one step search

$V(s)$: pre-trained trained DVN
$F(s)$: pre-trained F-Network

bestAction=getActionMaximizingValue(s);
s', r=environment.step(bestAction,s);

Function *getActionMaximizingValue(s)*
    $a \leftarrow F(s)$ //remove actions that lead to non-feasible states (optional)
    $a_{\text{best}} \leftarrow \max_a (V(s'))$; $//V(s') = V(T(s,a))$
    return a_{best};
end

3.6.2 Random rollout with DVN

The disadvantage of a Random rollout is that it often requires a deep search horizon in order to get valuable insights. Simultaneously, there also has to be enough CPU resources available to repeat the simulation many times, so that reasonable value estimates are obtained. With speed demands and limited computational resources, this trade off usually results in having to choose a short search horizon. However, by utilizing a DVN, the information deficiency of a short search horizon can be complemented.

During a Random rollout, there is no evaluation of the end state, i.e the state at the search horizon. In other words, the collected rewards gets truncated. If the search is very deep, this has little effect on the final value estimates. However, for short searches, this can result in very skewed assessments. As discussed in section 2.2, the reward of an action tells little about the long term future rewards of that action. Similarly, a too short search horizon gives little information on the long term values. However, by utilizing a DVN, the final state-value can be approximated at the search horizon.

Pseudo code: Random rollout with DVN

Algorithm 8: Random rollout with DVN

$V_L(s)$-pre-trained/Continuously trained DVN/NN //long term memory
$q_S(s_0, a) = 0 \ \forall \ a$ //clear short term memory
while *cpuTimeNotFinished* **do**
  $s \leftarrow s_0$;
  return = 0;
  step = 0;
  a_0;
  while *step < searchHorizon AND isNotFailure(s)* **do**
    a $\leftarrow F(s)$ //remove actions that lead to non-feasbile states
    a_{rand} = getRandomAction(s, **a**);
    If (step==0) **then** $\{a_0 = a_{rand}\}$ //store first action
    $s', r \leftarrow$ environment.step(s, a_{rand});
    return $\leftarrow$ return + $\gamma^{step} \cdot r$;
    $s \leftarrow s'$
    step++;
    If (step==searchHorizon) **then** {return $\leftarrow$ return+$\gamma^{step} \cdot V(s)$}
  end
  $q_S(s_0, a_0) = q_S(s_0, a_0) + \alpha \cdot (return - q_S(s_0, a_0))$; where $\alpha = \frac{1}{\text{step}}$
end
p=Random(0,1);

$$\text{a}=\begin{cases} \max\limits_{a}(q_S(s_0, a)) & \text{if p}<\epsilon \\ \max\limits_{a}(q_S(s_0, a) + V(T(s_0, a))) & \text{else} \end{cases}$$

// ϵ decaying linearly with number of episodes. Trust the agent when it has seen more episodes.

3.6.3 Monte Carlo tree search

The core elements of section 2.8.4 is used when implementing MCTS in the Pong environment. However, there are a few adjustments that one has to make. For simplicity there will be no probabilities P present in the algorithm. Also, the agent that is being trained during the MCTS simulations will also be the one guiding the MCTS. Since the agent will need some exploration of the environment a stochastic approach will be taken to the final move for each MCTS simulation in the earlier stages. In the beginning episodes of the training procedure the final move will be chosen with respect to the number of times N_s a state has been visited with a higher probability than in the end of the training procedure. Also, for these exploration steps to be efficient the exploration parameter c will be increased temporarily to 100, in order to properly match the exploitation term in the UCB policy. Moreover, not only the state value v will be back propagated when a leaf node has been reached, but also the reward r produced by the reward function in the Pong environment. The combination of $r + v$ will make sure that both short and long term value of an action is taken into account when evaluating the consequent state of an action, since r only have a significant impact when the search is approaching the racket. It is

important to mention that both a racket hit and miss is, permanently, considered as a leaf node. Also, for simplicity, the tree will not be saved between each actual step in the algorithm. The same network architecture and hyperparameters as before are used for the network being trained and guiding the simulations.

3.6.3.1 Pseudo code

Algorithm 9: Monte Carlo Tree Search

$V(s)$-continuously trained DVN
p=Random(0,1);
$\begin{cases} \text{c=100} & \text{if p}<\epsilon \\ \text{c=}\sqrt{2} & \text{else} \end{cases}$
j=0;
while *cpuTimeNotFinished and j<maxIter* **do**
    j++;
    $s \leftarrow s_0$; //root node
    while *isNotLeafNode(s)* **do**
        a = getUCBPolicy(s);
        $s', r \leftarrow$ simulatedEnvironment.step(s, a);
        tree.traverse(a) $s \leftarrow s'$
    end
    EXPANSION
    v=V(s) //Prediction
    Back propagate $v + r$ upwards in the tree
end
a=$\begin{cases} \underset{a}{max}\ \text{tree.}N(s_0, a) & \text{if p}<\epsilon \\ \underset{a}{max}\ \text{tree.}Q(s_0, a) & \text{else} \end{cases}$
// ϵ decaying linearly with number of episodes. Trust the agent to exploit when it has seen more episodes.
$s_0 \leftarrow$ environment.step(s_0, a);
Discard tree for next simulation

4

Results

In this chapter, obtained results are presented, both for the small and large pong environment defined in table 3.1. The different implementations are then compared with each other in a stress test.

4.1 DVN

The training plots of the DVN together with the greedy one step search applied to the large and small environment can be seen in figure 4.1 and 4.2, respectively. Performance on the y-axis is measured as the number of hits during the episode, meaning the number of times the racket manages to hit the ball.

Small environment

Figure 4.1 shows how the training evolves with the number of episodes played, using the reward function presented in section 3.4.1 as well as the network architecture and hyperparameters presented in section 3.4.1. The network was trained 400 episodes with a limit of 15 hits in a row, in order to diminish the influence of repeated trajectories, as discussed in section 3.4.1. These repeated cycles are specific to the pong environment and of little relevance when analyzing the physical implications of the algorithms and their performance in a real-world application.

The moving average uses the last 50 episodes. The network utilizes the epsilon decay scheme of eq.(3.2), and after approximately 300 episodes ϵ reaches its minimum 0.05.

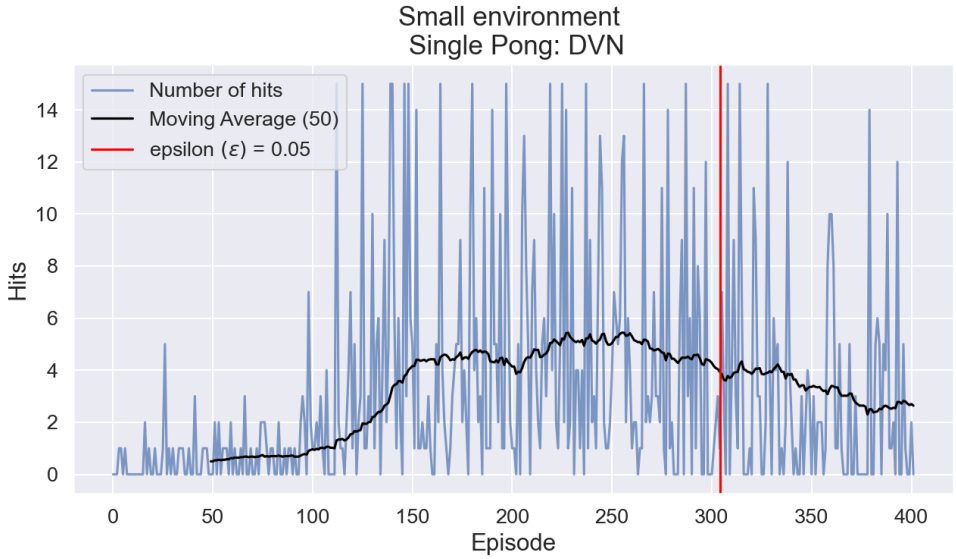

Figure 4.1: The plot illustrates how the training evolves with the number of episodes for DVN with the greedy one step search. The red line marks the episode where ϵ reaches its minimum 0.05. The moving average is based upon the last 50 episodes. The network was trained for 400 episodes. Training time: 1848 seconds.

Large environment

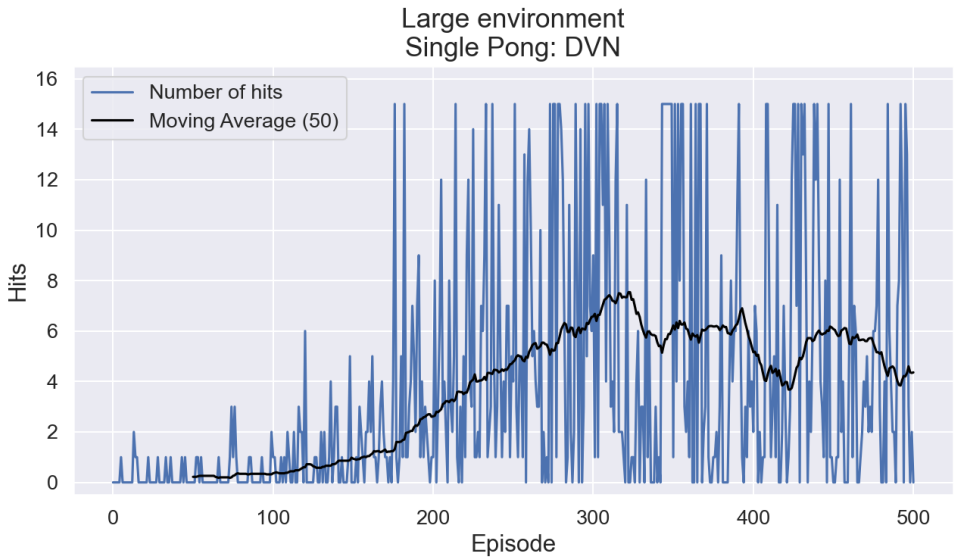

Figure 4.2: The plot illustrates how the performance, racket hits for each episode, of the DVN improves with number of episodes played. Here epsilon decreases linearly with the number of episodes in the range (0.8,0.05).

Here, in the large environment, the learning rate is set to 0.0001 in order to achieve optimal performance. In figure 4.2, one can see that the performance is steadily increasing but also limited.

4.2 Search: Random rollout

Small environment

The results from the Random rollout search (section 3.5) can be seen in figure 4.3. The search horizon is set to 5, with a time restriction of 10 ms for each search. This implies that in every state of the game, a maximum of 10 ms is dedicated to perform the search. It is clear from the flat averaging curve in figure 4.3 that no learning is present. Instead, the performance of the search is tightly bounded to its search horizon.

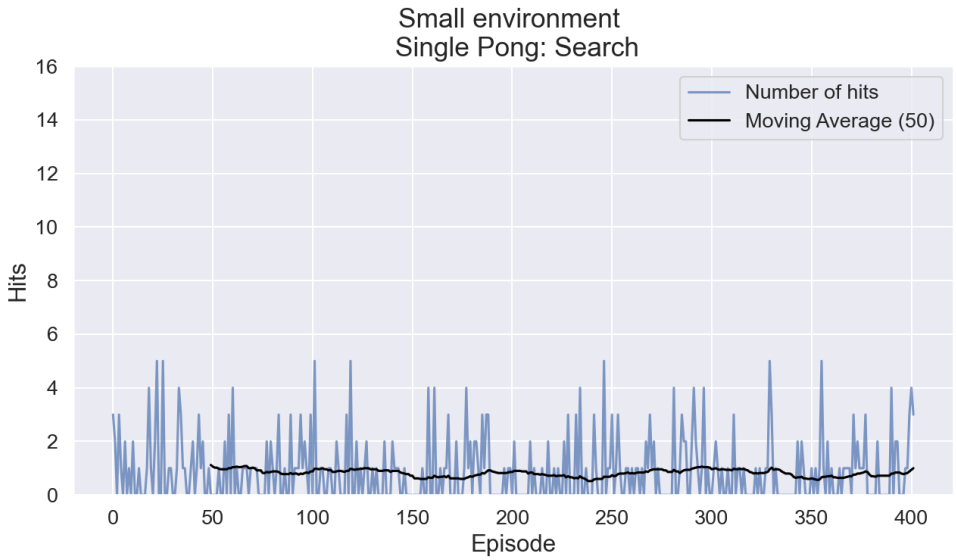

Figure 4.3: The plot illustrates the number of hits per episode for the Random rollout search. The search horizon is set to 5.

Large environment

The results from the Random rollout search, for the larger environment, can be seen in figure 4.4. In this particular plot, the search depth is set to 20, with a time restriction of 10 ms for each search, i.e in every state of the game, a maximum of 10 ms is dedicated to perform the search. It can be noticed that a search horizon of 20 is required in order to get a similar average as in figure 4.3. This is expected, since a larger environment also implies a larger state space and more potential positions for the racket.

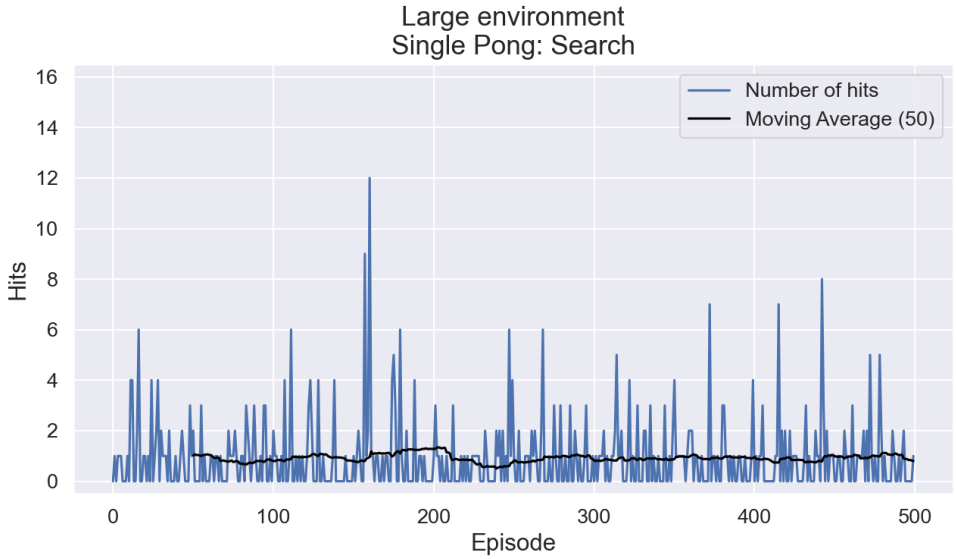

Figure 4.4: The plot illustrates the number of hits per episode for the Random rollout search in the large environment. The search horizon is 20 in this particular example.

4.3 Search + DVN: Random rollout with DVN

Small environment

The training plot from combining the Random rollout search with the DVN (section 3.6.2) can be seen in figure 4.5. The same network architecture and hyperparameters as in figure 4.1 were used. The same epsilon decay was also implemented. Similar to figure 4.3, the time restriction for each search is 10 ms and the search horizon is 5.

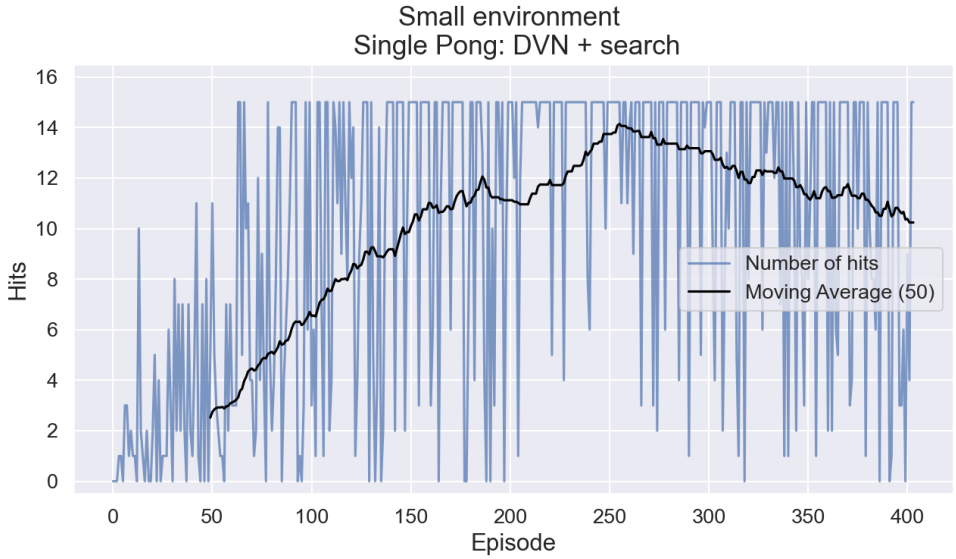

Figure 4.5: The plot illustrates the training evolution during 400 episodes of training a DVN together with the Random rollout search algorithm. The moving average is based on the result of the last 50 episodes. Training time: 8800 seconds.

Large environment

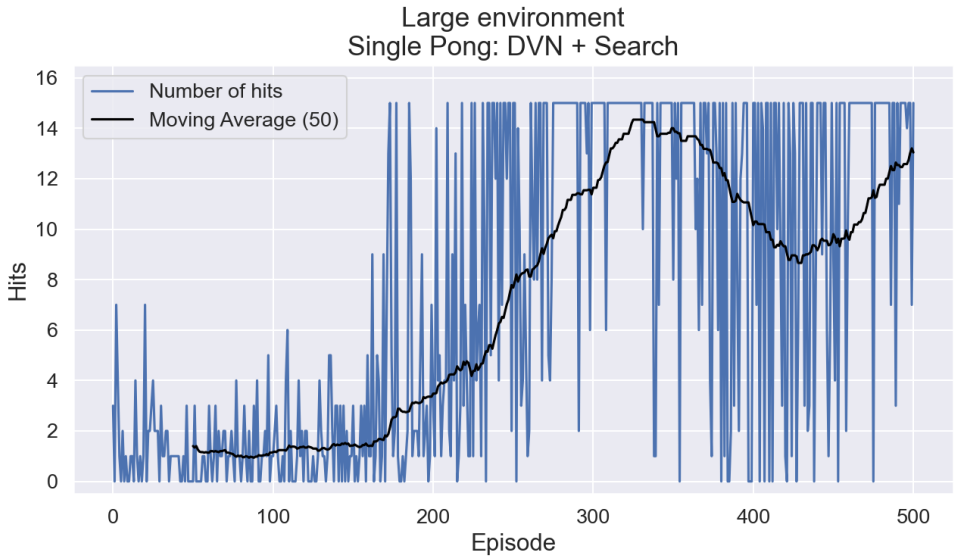

Figure 4.6: The plot illustrates the training evolution during 500 episodes when training a DVN together with the Random rollout search algorithm with a search depth of 20. The moving average is based on the result of the last 50 episodes.

In figure 4.6 one can see that optimal performance is reached after about 300 episodes. Then some catastrophic forgetting appears and after that the algorithm approaches the optimum for the weights again.

4.4 Evaluation plots

In this section, evaluation plots for different search depths are presented. In figure 4.7, the performance difference between utilizing only search (Random rollout) and DVN + search (Random rollout together with DVN) is shown. All averages was obtained from 25 samples. There is a clear improvement in performance for low search depths. A very interesting result, since computationally taxing real life applications usually prohibits deep searches.

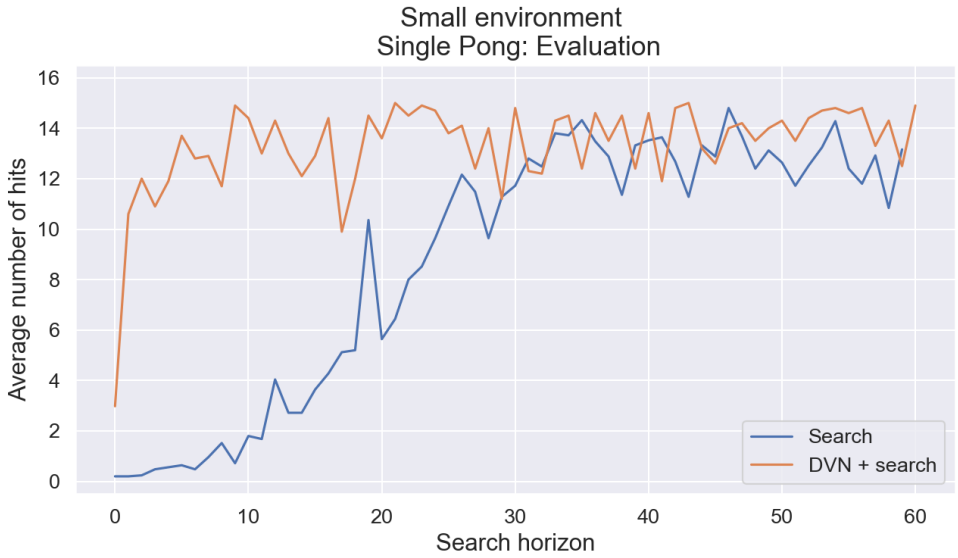

Figure 4.7: The plot illustrates the average number of hits for a given search depth. It shows the performance difference between the Random rollout search algorithm versus combining it with a pre-trained DVN. The number of hits for each episode is an average over 25 samples.

4.5 Stress test

In order to more clearly compare the efficiency of the different methods, a so-called stress test was conducted. The purpose of the stress test is to simulate demanding conditions that resembles the environment in a real-world application better. It demonstrates how the methods perform when available CPU is very scarce. During every simulation step, a CPU heavy time consuming dummy loop was implemented. The x-axis shows the time restriction, i.e the total number of milliseconds allowed during every search. When the time limit is up, a real (non-simulating) action is taken, and a new search begins. The average number of hits are obtained from 15 samples in each time restriction, and the plot shows the moving average from the last 5 time restrictions so that the trends are clearly visible.

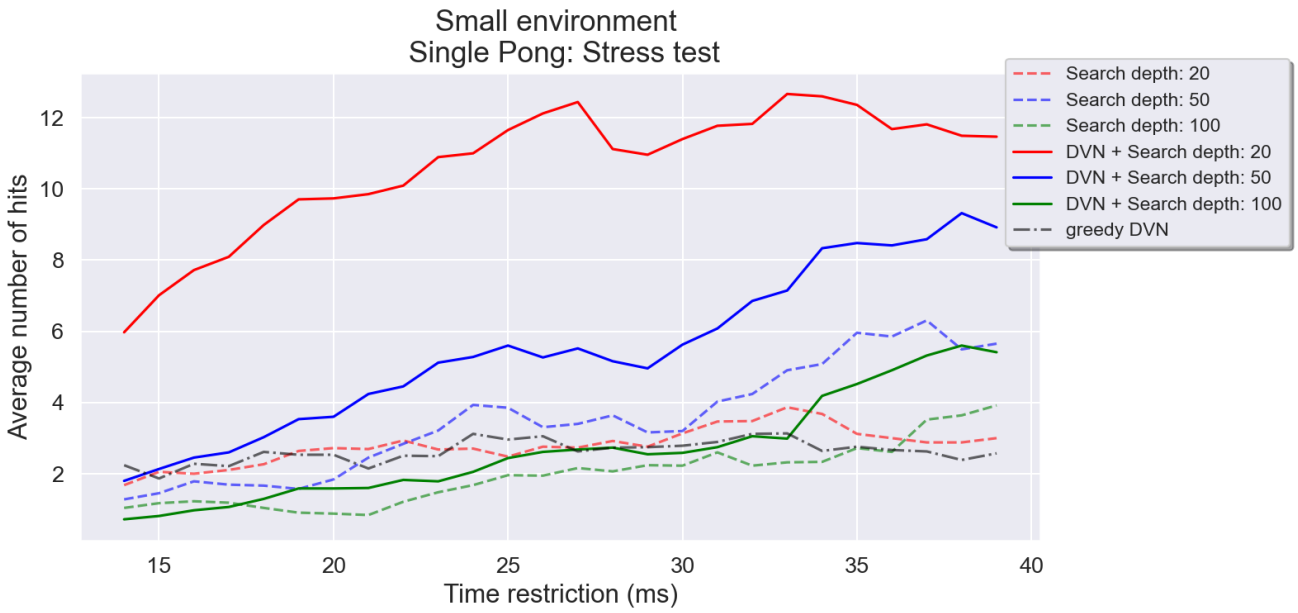

Figure 4.8: The plot illustrates a stress test conducted for three different methods: search (Random rollout), DVN + search (Random rollout with DVN) and a greedy DVN (DVN with a greedy one step search). The methods were tested for three different search horizons: 20, 50 and 100.

The performance superiority of combining a Random rollout with DVN is evident in figure 4.8. The combination results in a higher hit rate for every search horizon (compare the dashed lines with the normal lines) for almost every time restriction. Moreover, a DVN with a search horizon of 20 (red line) manages to drastically outperform all other methods, even for very low time restrictions. Also, it is clear that Random rollouts (searches without learning) is heavily underperforming in the stress test. In figure 4.7, where the time restriction is 10 ms, a search horizon of roughly 30 is enough to perform optimally. In the stress test, however, there is simply not enough time to get reasonable value estimates from searches alone, seen by the dashed lines in figure 4.8.

Another aspect of interest when examining the dashed curves in figure 4.8 is that during very taxing circumstances, i.e low time restrictions, a short search horizon (depth 20) naturally outperforms the deeper searches (depth 50 and 100). As the time restriction eases though, a search depth of 50 begins to outperform both a search depth of 20 and a search depth of 100. This is expected, as there is an intrinsic trade off between selecting a deep search horizon while simultaneously having enough time run many simulations at that depth. Finally, it is worth mentioning the performance of the greedy DVN, i.e DVN with the greedy one step search. This demonstrates the base line strength of the DVN alone, which is quite weak.

4.6 MCTS

Using the algorithm described in 3.6.3, with a restricted number of simulations, the number of times the search reaches a leaf node, of 60.

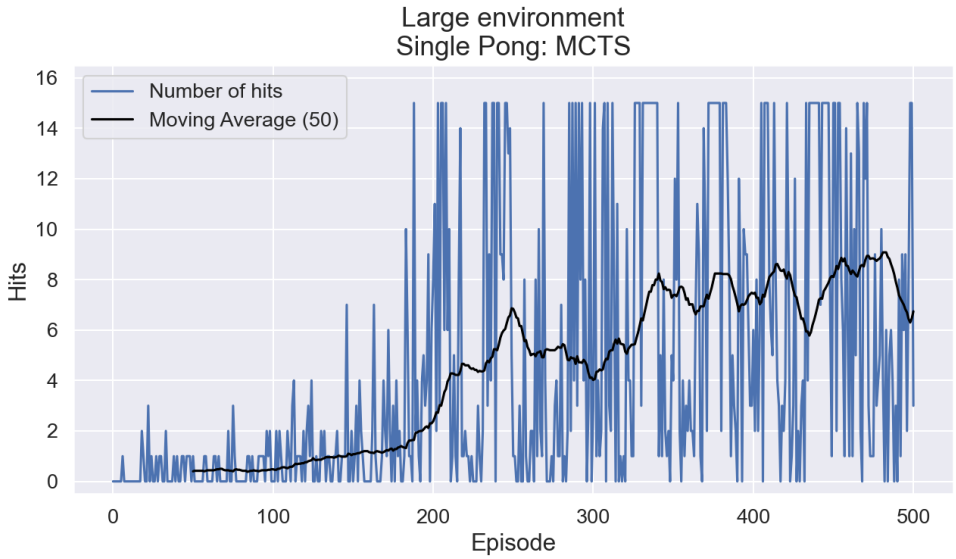

Figure 4.9: The plot illustrates the training episodes for DVN using MCTS.

By studying figure 4.9 one can realize that the trend is pointing upwards but also that there is a lot of catastrophic forgetting, which affects the training. There are several factors that one has to take into account when comparing this result to the previous one. This will be discussed in the discussion part.

4.7 F-Network

The implementation of the F-Network was not very successful, or at least not successful enough to make an impact. The agent learns that states closer to ground is more likely to be a fail state but does not seem to be able to differentiate between a state where the racket is further away from the ball and one where the racket is closer to the ball. It is important to take the racket position into account when deciding whether a state is a fail state or not, since a state where the racket is further away from the ball, in the x-direction, is more likely to be a fail state. This is further discussed in the upcoming discussion part.

5
Discussion

In this chapter, a thorough discussion of the obtained results and their implications are given.

First and foremost, it is evident from figure 4.1 that the learning algorithm can perform on its own. However, its performance seems restricted. The DVN learns and improves over time, but there are still episodes with fierce misses. This could potentially be due to to catastrophic forgetting, a common phenomenon where the agent abruptly seems to "forget" what it recently has learnt [26]. This can potentially be combated with prioritized replay [27]. A simplified version of the prioritized replay was implemented and tested without any major improvements though.

Moreover, another important aspect which reduced the tendencies of catastrophic forgetting was keeping the experience replay buffer, 2.6.2, size very large (100 000) and limit the number of hits at a maximum of 15 hits in a row. A possible problem with the smaller buffer sizes (500-10000) is that they quickly get filled up with only good examples the moment the agent learns to play decently, and so the exposure to wrong action choices gets drastically diminished. The larger buffer size reduced this tendency, as well as restricting the number of hits in a row to only 15. However, it could be more beneficial to reduce this cap even more, at least during training, in order to not fill up the replay buffer with only good examples. Another way to combat catastrophic forgetting could be to stop the training when a certain predetermined goal has been reached, for example 15 hits for several episodes in a row, in order to save the weights of the network when they are somewhat optimal. Furthermore, one could also try to stop the decay of epsilon, the probability of a random action, at an earlier stage than 0.05 in order to experience diversity in the state space when the agent is well-performing.

However, a perfectly performing DVN was never the main objective. On the contrary, the interesting aspect is the enhancement it provides to a search. A non-optimal DVN gives a more fair representation of any real life engineering problem, which is rarely solved by simply utilizing a neural network alone.

Looking at the results in general, it is clear that there are a few differences between the small and the large pong environment. First of all, looking at the figures 4.3 and 4.4 the larger pong environment needed a smaller learning rate in order for the loss function to not get trapped in a local minima and consequently a larger amount of episodes, 500 instead of 400, was needed for the training. Also, as one should expect,

a deeper search depth was needed in order for the random search algorithm to more or less replicate the same average as in the small environment. More specifically, it was found that a search depth of 20 in the large environment corresponded to a search depth of 5 in the small environment. The same conclusions can be drawn from figures 4.5 and 4.4 where the training results from the DVN + Search are shown. Here it is important to note that the search depth for the small environment was set to 5 during training and the search depth for the large environment was set to 20 during training.

When it comes to the Random rollout search algorithm, it is clear from the moving average in figure 4.3 that there is no improvement over time. This means that, as expected, the algorithm does not learn. In this particular example, the search depth is set to 5 with a time restriction of 10 ms, and the average number of hits fluctuates around 1. By increasing the search depth and keeping the time restriction at 10 ms, it can be seen in figure 4.7 that optimal performance is achieved already at a search depth of around 30. This is of course heavily dependent on the specifics of the environment (section 3.3) as well as the action space. What is interesting though is the general trend. When given large enough CPU time and deep enough search depth, the Random rollout search manages well on its own, as it hits almost perfect scores between the search depth of 30 and 60 in figure 4.7. However, in practise, CPU time will always be a scarce resource when making decisions in live applications, for example when steering a truck autonomously. Therefore, it is interesting to examine figure 4.8, where the computationally heavy dummy loop was implemented - a more likely scenario in any practical problem. It is evident that more demanding time restrictions impacts the search algorithms negatively.

When combining the search and learning part, it is evident from figure 4.5 that both the amount of training episodes required and training performance is greatly improved. The sequences with zero hits are almost completely gone. Since the search requires time to simulate (10 ms in each state), the training time was almost four times greater than for the DVN with the greedy one step search in figure 4.1. When training though, the search could potentially be run on a separate thread, to speed up the training time. Another interesting aspect is that the Random rollout search combined with DVN can also be understood as making more out of the available training data than the DVN alone, which means that in cases where data gathering is costly and time consuming, the additional training time might not be a problem.

In figure 4.7, one can notice a great improvement already at a search depth of three when combining the DVN with the Random rollout search. At a search depth of around 10 the performance is roughly optimal. This demonstrates the real strength of combining learning with search, and the results are further validated in figure 4.8. The time restriction is often fixed from the hardware settings in a practical problem, and so it will impose the need of a shallower search in order to get reasonable value estimates. As already discussed, deep searches with low time restrictions are ideal settings that seldom can be afforded.

The comparison between the MCTS algorithm and the DVN+Search algorithm is problematic in many different ways. First of all, it is hard to equate the search depth of the random rollout algorithm with the maximum number of iterations of a simulation in MCTS, since the search depth will depend on the actual performance of the DVN. However, it is necessary to restrict the search algorithms in some way since otherwise they would be too good for the environment of single pong. Also, for MCTS there is a lot of catastrophic forgetting. This is because of the fact that the search is highly dependent on the performance of the network guiding it. Moreover, the training could potentially be improved by continuously saving the tree after each actual step in the algorithm. In order to really showcase the strength of MCTS, one would need to compare its performance, after training, in the double pong environment where the state space is large enough. Also, during evaluation catastrophic forgetting, which is detremental for MCTS, will not be present.

The F-Network, which was intended to speed up the search by cutting the search when a failure state is found, was not very successful. There are several reasons for this. The logic behind the algorithm is intuitive and well-performing for simple environments. However, when dealing with larger state spaces there seems to be a a big problem regarding the learning pattern and convergence of the weights. The agent learns that states closer to ground is more likely to be a fail state but does not seem to be able to differentiate between a state where the racket is further away from the ball and one where the racket is closer to the ball. In order for the F-Network to serve its purpose, it is vital to take the racket position into account when deciding whether a state is a fail state or not. In conclusion, when dealing with a state space and input of this size, an alternative method or algorithm should be considered. Another apparent problem is that, for single pong at least, there are not much fail states in the first place. The racket is able to save most situations and therefore, even if the F-Network was optimally categorizing fail states, the benefit would not be significant enough to motivate this kind of approach. To combat this type of problem, one could for example try to reduce the speed of the racket.

5.1 Further research

There is much room for further research and improvements in the search for optimal learning and search methods in the pong environment. One first obvious thing to tackle would be to, more rigorously, study different combinations of hyperparameters and network architectures in order to test their effect on the overall performance. One particularly interesting aspect is the type of neural networks that are being utilized. One could for example also try to implement a convolutional neural network instead of a feedforward neural network. By implementing a convolutional neural network one could process an image of the current state of the game as input instead of having different attributes of objects in the game as input.

The next step in this topic would be to expand the environment to include two rackets and test the different methods implemented in the single pong environment in the double pong environment. This would imply a bigger state space and really

put the algorithms developed in this thesis to the test. First and foremost, it would be interesting to study the coordination and cooperation of the rackets, since this was one of the main reasons VAS announced this thesis. A quick implementation of MCTS in the double pong environment was made in the very end of the thesis, and the agent seemed to be able to play in an effective manner. However, this was by no means studied extensively and has to be researched further. Also, an alternative version of the F-Network could be studied and implemented in the double pong environment.

Furthermore, by the time one feels satisfied with the implementations in the double pong environment there are a lot of interesting add ons that could contribute to a higher complexity of the environment. For example, more balls could be added to the game in order to make it harder for the rackets to navigate the state space. Also, from a physics standpoint, it would be interesting to make the environment more challenging by adding friction for the rackets, air resistance or stochastic winds to make the movement of the balls even more unpredictable.

6
Conclusion

The purpose of this thesis work was to investigate ways of enhancing search methods used in reinforcement learning by utilizing neural networks. More specifically, the goal was to study the performance of different learning and search algorithms in the game of pong. The result shows that the combination of search algorithms together with DVN drastically outperforms plain search methods, especially in environments where deep searches are unfeasible and CPU resources are restricted. In order to draw further conclusions regarding the performance differences between MCTS and DVN+Search, a more complex environment must be more rigorously studied. To expand the environment to double pong, possibly with more than one ball, would lead to more and deeper insights in this regard. Furthermore, the F-Network did not show any promising results. However, it is possible that a different approach to the binary classification algorithm in the double pong environment could lead to a decrease in search time.

Bibliography

[1] Wikipedia. *State space search*. https://en.wikipedia.org/wiki/State_space_search. Accessed: 2022-04-09.

[2] Wikipedia. *Curse of dimensionality*. https://en.wikipedia.org/wiki/Curse_of_dimensionality. Accessed: 2022-04-09.

[3] Richard Ernest Bellman. *The Theory of Dynamic Programming*. RAND Corporation, 1954.

[4] Sankar Das Sarma, Dong-Ling Deng, and Lu-Ming Duan. "Machine learning meets quantum physics". In: *Physics Today* 72.3 (Mar. 2019), pp. 48–54. DOI: 10.1063/pt.3.4164. URL: https://doi.org/10.1063%2Fpt.3.4164.

[5] R. B. MARIMONT and M. B. SHAPIRO. "Nearest Neighbour Searches and the Curse of Dimensionality". In: *IMA Journal of Applied Mathematics* 24.1 (Aug. 1979), pp. 59–70. ISSN: 0272-4960. DOI: 10.1093/imamat/24.1.59. eprint: https://academic.oup.com/imamat/article-pdf/24/1/59/1941049/24-1-59.pdf. URL: https://doi.org/10.1093/imamat/24.1.59.

[6] Hamed Mohammadbagherpoor et al. *Exploring Airline Gate-Scheduling Optimization Using Quantum Computers*. 2021. DOI: 10.48550/ARXIV.2111.09472. URL: https://arxiv.org/abs/2111.09472.

[7] Wikipedia. *Go*. o(https://en.wikipedia.org/wiki/Go_(game). Accessed: 2022-04-09.

[8] Levente Kocis and Csaba Szepesvari. "Bandit based Monte-Carlo Planning". In: (2006). URL: http://ggp.stanford.edu/readings/uct.pdf.

[9] Muller M. Silver D. Sutton R. "Sample-Based Learning and Search with Permanent and Transient Memories". In: (2008). URL: https://www.davidsilver.uk/wp-content/uploads/2020/03/dyna2_compressed.pdf.

[10] Maddison C. Silver D. Huang A. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* (2016). URL: https://doi.org/10.1038/nature16961.

[11] Gerald Tesauro and Gregory R. Galperin. "On-line Policy Improvement using Monte-Carlo Search". In: (1997).

[12] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038%2Fs41586-020-03051-4.

[13] I.P. Pavlov. "Conditioned reflexes: An investigation of the physiological activity of the cerebral cortex". In: (1927). URL: https://antilogicalism.com/wp-content/uploads/2019/04/conditioned-reflexes.pdf.

[14] Yuxi Li. "Deep Reinforcement Learning". In: 2018. Chap. 5.

[15] Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2014. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[16] R.K. Getoor and M.J. Sharpe. "Markov Properties of a Markov Process". In: *Springer-Verlag* (1981). URL: https://link.springer.com/content/pdf/10.1007/BF00532123.pdf.

[17] RICHARD BELLMAN. "A Markovian Decision Process". In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684. ISSN: 00959057, 19435274. URL: http://www.jstor.org/stable/24900506 (visited on 04/21/2022).

[18] Christopher Watkins. "Learning From Delayed Rewards". In: (Jan. 1989).

[19] Kevin P. Murphy. *Machine learning : a probabilistic perspective*. Cambridge, Mass. [u.a.]: MIT Press.

[20] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: http://arxiv.org/abs/1312.5602.

[21] Zhang et al. *Dive into Deep Learning*. 2022.

[22] William Fedus et al. *Revisiting Fundamentals of Experience Replay*. 2020. DOI: 10.48550/ARXIV.2007.06700. URL: https://arxiv.org/abs/2007.06700.

[23] Tilmann Gneiting and Peter Vogel. *Receiver Operating Characteristic (ROC) Curves*. 2018. DOI: 10.48550/ARXIV.1809.04808. URL: https://arxiv.org/abs/1809.04808.

[24] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.

[25] *Feature scaling — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-May-2022]. 2022. URL: https://en.wikipedia.org/wiki/Feature_scaling.

[26] Michael McCloskey and Neal J. Cohen. "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". In: ed. by Gordon H. Bower. Vol. 24. Psychology of Learning and Motivation. Academic Press, 1989, pp. 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8. URL: https://www.sciencedirect.com/science/article/pii/S0079742108605368.

[27] Tom Schaul et al. *Prioritized Experience Replay*. 2015. DOI: 10.48550/ARXIV.1511.05952. URL: https://arxiv.org/abs/1511.05952.

A

Appendix 1

A.1 Hyperparameter optimization for DVN with the greedy one step search

In order to find reasonable values for the hyperparameters in section 3.4.1, a so-called hyperparameter optimization was performed. For three different learning rates: 0.01, 0.001 and 0.0001 in fig A.1, fig A.2 respectively fig A.3, the total number of hits collected during 500 episodes of training is plotted for different batch sizes and different number of hidden neurons. From the figures, it is evident that the number of hidden neurons has the largest influence on the learning; in all figures, the accumulated hits is increased with increased number of hidden neurons. The batch size, on the other hand, can be seen to have a smaller impact. A clear winning candidate can be spotted in figure A.2, with a learning rate of 0.001, batch size of 500 and 64 hidden neurons. This is also the hyperparameters used for the DVN.

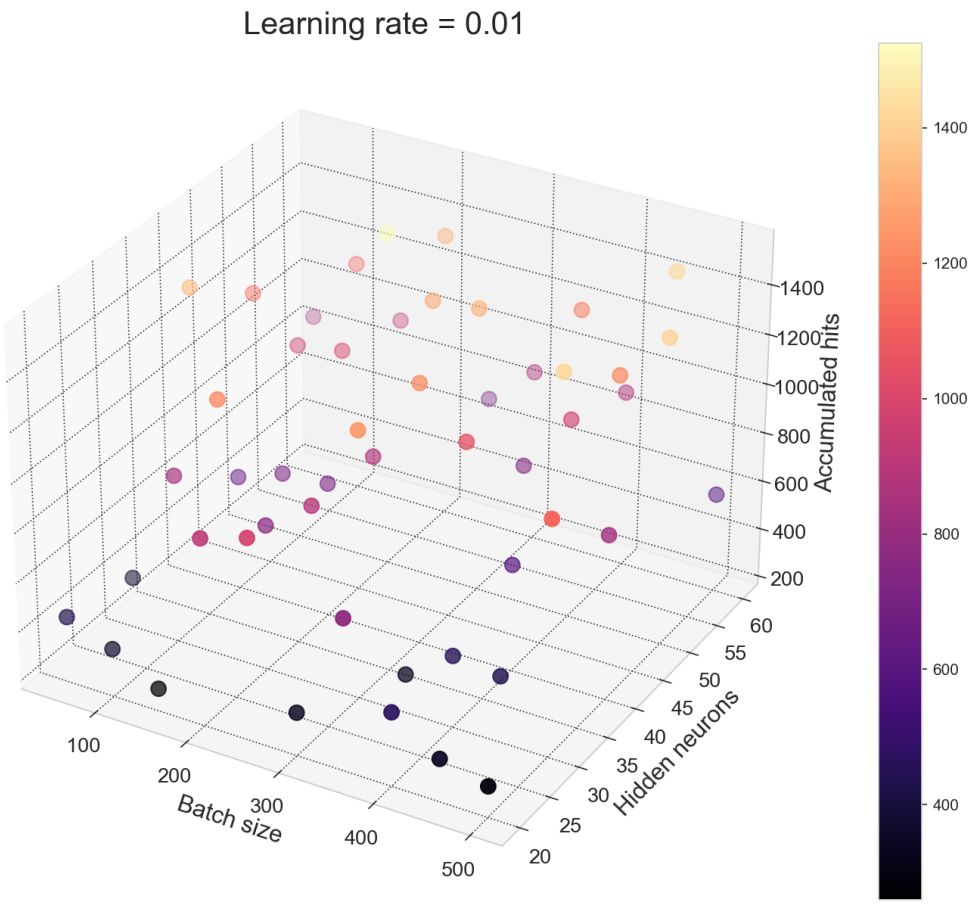

Figure A.1: The plot illustrates the hyperoptimization with learning rate 0.01.

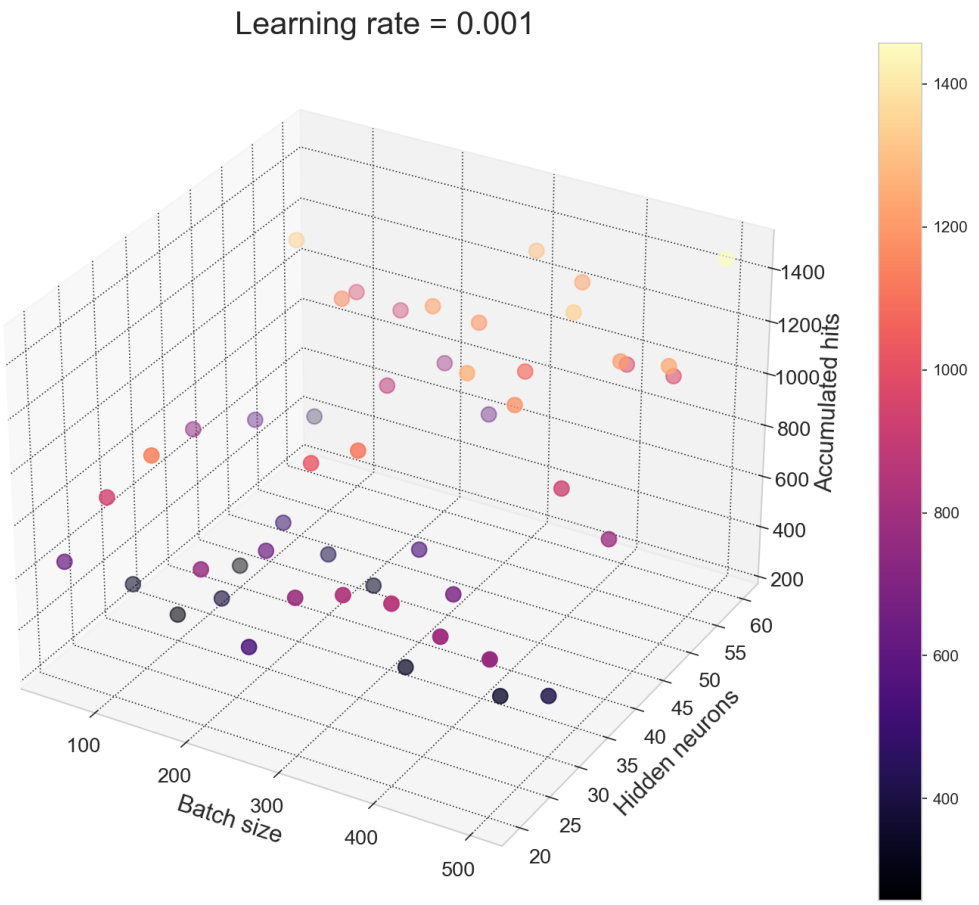

Figure A.2: The plot illustrates the hyperoptimization with learning rate 0.001.

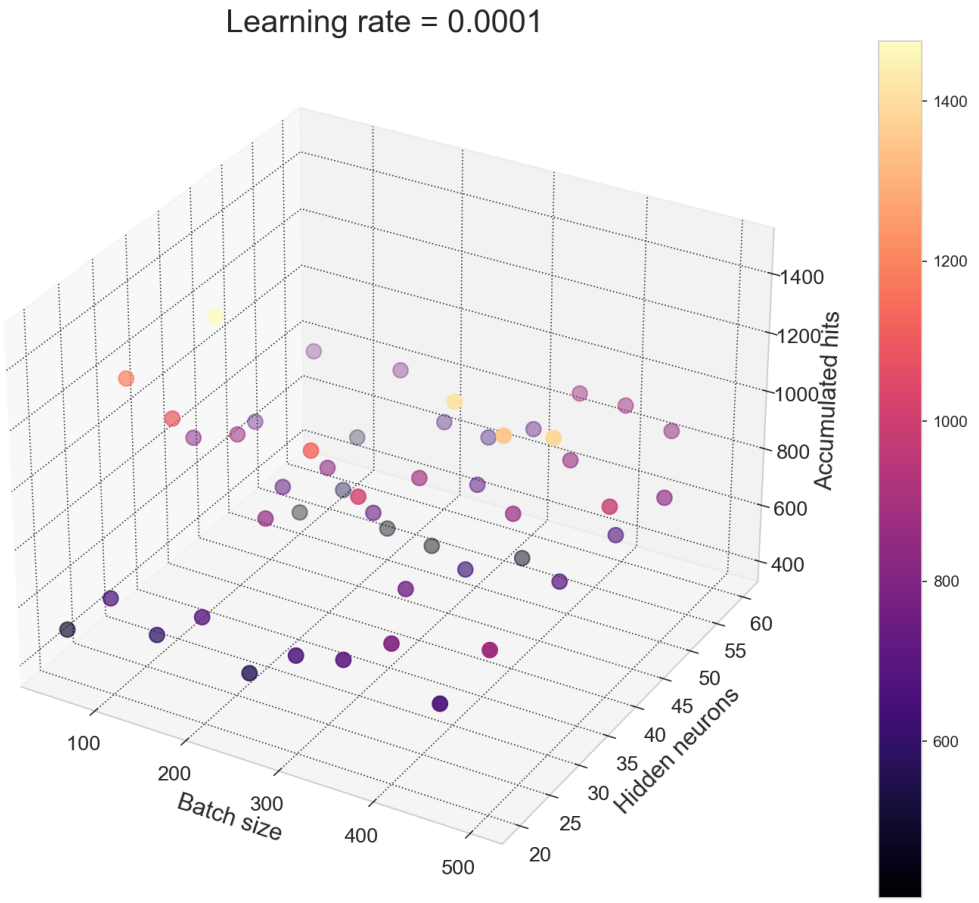

Figure A.3: The plot illustrates the hyperoptimization with learning rate 0.0001.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY