Poznań University of Technology
Institute of Computing Science

Mateusz Kurek

Deep Reinforcement Learning in Keepaway Soccer

Master's thesis

Supervisor: Wojciech Jaśkowski, Ph.D.

Poznań, 2015

Abstract

This thesis focuses on deep neural networks and reinforcement learning (a.k.a deep reinforcement learning) in the context of the keepaway soccer problem.

Deep learning is a new area in machine learning, but it already produced numerous successful applications. It is capable of learning high-level abstractions and it is known to be especially useful for problems with highly-dimensional input, such as image and speech recognition. A recent study combines deep learning (using deep neural networks) with reinforcement learning for general video game playing (GVGP), leading to an agent outperforming a human player.

The keepaway soccer problem, in which the keepers try to maintain possession of the ball away from the takers for as long as possible, is widely used multi-agent reinforcement learning benchmark. It poses many challenges to machine learning methods, such as a large state space and uncertainty. Although, its computational requirements make it feasible and convenient to experiment with it, it is complex enough so that it cannot be solved trivially. In contrast to GVGP, a state in the keepaway soccer is characterized by just a dozen or so variables.

The main objective of this thesis was to verify whether deep reinforcement learning, and the recently proposed Deep Q-Learning in particular, is suitable for such a low-dimensional environment. To this aim various deep reinforcement learning techniques were compared on the keepaway soccer benchmark. To alleviate the problems of correlated data between consecutive samples, an experience replay mechanism was used, which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors. Another used techniques, which increases the stability of Deep Q-Network were RMSProp and target network freezing.

The results of extensive computational experiments proved that deep learning methods have a great potential also when applied to such a low-dimensional state space problem as keepaway soccer. It outperformed all previous approaches to this task using only a fraction of the computational expense of the runner-up, Symbiotic Bid-based (SBB) GP.

Among of the different techniques used, experience replay and minibatches turned out to be especially effective, significantly increasing the agent's performance. A minor improvement can be also obtained by using heterogeneous team learning in place of the homogeneous one.

Surprisingly, the experiments showed, that there is no advantage in using deep neural networks over shallow neural networks in keepaway soccer problem. Similarly, freezing the target network and using meta-states does not improve the overall results and sometimes decrease them. There is also no clear winner between the classical RMSProp and the DeepMind's version.

The results obtained in this thesis indicate that Deep Q-Learning has a large potential also for low-dimensional reinforcement learning problems.

Acknowledgements

I am very grateful for the advice and support of my supervisor, Wojciech Jaśkowski, who have shown a large interest in my work. His stimulating motivation and valuable ideas helped me to complete this thesis. Our numerous discussions, either face-to-face or through the email, have greatly improved this work. I thank him especially for constructive criticism and careful revision of the text.

I want to thank my beloved fiancée Monika and my family for their love, support and, most importantly, patience they showed me during writing this thesis.

Contents

Chapter 1

Introduction

> You have to learn the rules of the game. And then you have to play better than anyone else.
>
> *Albert Einstein*

1.1 Motivation

This study focuses on the fields of reinforcement learning and deep neural networks.

Machine learning is a field in computer science that explores the construction of algorithms that can learn from and make predictions based on data. In contrast to the traditional expert-based approach, in which a computer is told exactly what to do in response to a given data, learning-from-data approaches are not told how to behave in certain situations – they can learn from the observations of the data and figure its own solution to the problem. One of the subfields of machine learning are artificial neural networks, which are one of the simplest, and yet one of the most powerful learning models that was ever invented. Artificial neural networks are inspired by biological neural networks and are used to estimate value of functions that can depend on a large number of inputs and are generally unknown.

Deep learning is a new research track within the field of machine learning. Nowadays it is one of the hottest trends in artificial intelligence. The main idea behind deep learning is to create multiple-layer architectures, learning hierarchies of concepts, and high-level abstractions. Deep neural networks, which are basically neural networks with many hidden layers, are well-known for their successful results in the fields of image processing [22], speech recognition [7], and natural language processing [48].

Reinforcement learning is one of the most promising artificial intelligence paradigms. It is successfully used in fields such as robotics [21], operations research [39], or economics [34]. One of the most successful application of reinforcement learning is playing board games, especially world-famous TD-Gammon, which uses temporal different learning and self-play to achieve super-human level in backgammon. Another application of reinforcement learning is robotics, where it offers a framework and set of tools for the design of sophisticated and hard-to-engineer behaviors. The robot learns from trial-and-error interactions with the environment. By observing the results of its actions it can determine the optimal behaviour — the actions to take to reach defined goal or the highest reward. Examples of such robots include playing air hockey [2], learning ball acquisition [20], and even playing robotic soccer [42].

A recent work which brings together deep learning and reinforcement learning is „Playing Atari with Deep Reinforcement Learning" [33] published by Google DeepMind. The paper describes a system that combines deep learning methods and reinforcement learning in order to create a system that is able to learn how to play a large set of simple games for Atari 2600. The system has access only to the screen of the game and the scores received during games. The system learns to understand which moves are good and which are bad basing solely on the visual data, which is exactly the same as a human player evaluates his performance and adapts his playing strategy. According to the reported results, the system was able to master a number of different games and play most of them better than a human player.

Recently, a combination of deep learning and reinforcement learning in the robotic field was also proposed [3]. This combination allows a learning agent to control a system based only on visual inputs using a deep neural network to extract relevant features from the images and it can be seen as a step towards truly intelligent machines.

Combination of deep learning and reinforcement learning is still a new area of research, with a lot of unknowns and open questions, thus it is motivating to further investigate quality of certain deep learning methods, and assess how do they perform in problems that not involve image, speech and natural language processing.

Keepaway soccer is a simplification of the RoboCup simulated soccer task, in which the objective is for the keepers to maintain possession of the ball away from the takers, for as long as possible. It presents many challenges to machine learning methods, including a large state space, uncertainty, multiple agents, and long and variable delays in the effects of actions. It is complex enough that it can not be solved trivially, but, at the same time, simple enough that complete machine learning approaches are feasible. Keepaway soccer has been put forth as a testbed and, currently, is widely used to compare different machine learning algorithms like Q-learning, neuro-evolution and genetic programming.

Deep learning (and especially deep reinforcement learning) is an effective method in general video game playing (GVGP) problems, in which the state space (visual input) is highly-dimensional . Here we ask the question, if deep reinforcement learning is also effective in low-dimensional state space problems, with a small number of input variables such as keepaway soccer?

1.2 Scope and Objectives

The main objective of this thesis is to experimentally verify whether deep reinforcement learning is a suitable technique for the keepaway soccer task. Specific research questions include:

1. Is deep reinforcement learning competitive to other methods used to approach this low-dimensional problem, such as Q-learning or SARSA, neuro-evolution or genetic programming?
2. Which of the techniques proposed by DeepMind [33] (experience replay, minibatch learning, target network freezing) are effective in keepaway soccer problem?
3. What are the best values of learning parameters?
4. Is deep reinforcement learning suitable for multi-agent environments?

The secondary objectives are:

1. To review of reinforcement learning, neural networks and deep learning methods and their applicability in the keepaway soccer task.
2. To design and implement efficient and flexible software framework to experiment with Deep Q-Learning for keepaway soccer.
3. To compare homogeneous and heterogeneous team learning in keepaway soccer.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 is an overview of reinforcement learning with emphasis on the Q-learning algorithm and ϵ-greedy policy. In Chapter 3, artificial neural networks and deep learning methods are described. Chapter 4 contains the review of keepaway soccer problem and recent studies in this domain. The implementation of the Deep Q-Learning algorithm and the architecture of the whole learning system is presented in Chapter 5. Chapter 6 is devoted to various experiments and results for keepaway soccer. Finally, Chapter 7 summarizes achieved results, draws conclusions, and proposes directions for further research.

Chapter 2

Reinforcement Learning

Reinforcement learning (RL) is the one of the most researched areas in artificial intelligence and machine learning, nowadays. In this approach an agent (also called a learner) is learning by interactions with a (possibly uncertain) environment. The agent learns what to do (how to map observed percepts to actions) to maximize (possibly discounted) cumulative reward, a.k.a reinforcement signal. The agent does not know, *a priori*, what action to make, but has to explore (by trial-and-error) to understand which action provides the highest reward over the time by trial and error. Reinforcement learning consists in learning from a sequence of observed states, performed actions and received rewards. [40] [67].

There are few differences when comparing reinforcement learning to supervised learning – the most popular learning paradigm studied in machine learning:

- Supervised learning consists in learning from examples provided by an external supervisor, whereas in reinforcement learning an agent learns from its own experience (the task is defined in such way that a predefined dataset of examples is hard or even impossible to get, as opposed to the supervised learning). In other words, when it comes to committed error, a supervised learning agent knows exactly what it should have done, while a reinforcement learning agent only knows that an action was incorrect (and eventually to what extent it was incorrect).
- Reinforcement learning has to deal with exploration-exploitation dilemma (described later in this section), while there is no such dilemma in supervised learning.
- Reinforcement learning involves delayed rewards, while in supervised learning rewards are known immediately.
- In supervised learning, the evaluation of the system is separated from the learning phase, whereas in reinforcement learning evaluation phase is usually done simultaneously with learning – reinforcement learning is, usually, on-line learning, while supervised learning is, typically, off-line learning.

Reinforcement learning can be formulated as a class of Markov Decision Process (MDP). An MDP is a discrete-time, stochastic control process, defined as a tuple (S, A, P, R, γ), in which:

- S is a finite set of states ($s_t \in S$, where s_t is state observed at (discrete) time t),
- $A(s_t)$ is a finite set of possible actions in state s_t ($a_t \in A(s_t),\ A(s) \subseteq A$, where a_t is action executed at time t),
- $P(s, a, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability, that executing action a in state s at time t will lead to state s' at time $t + 1$,
- $R(s, a)$ is function describing the immediate (or expected immediate) reward for taking action a in state s,
- γ is the discount factor, which denotes the difference in priorities between the immediate

reward and future rewards. A reward received n time steps ahead is worth γ^{n-1} times less than the same reward obtained immediately ($\gamma \in (0, 1]$).

The reinforcement learning problem is such MDP where R or P are unknown – the agent has to learn how an action is good (the reward function), and where his actions lead (the transition probabilities).

The agent follows policy $\pi(s)$ which is, in general, a mapping from states to probabilities of executing each (possible) action ($\pi : S \to A$). The expected cumulative discounted reward of an agent from state s following policy π is defined as

$$U^{\pi}(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_{t+1}]$$

The agent's goal is to maximize (expected and discounted) future rewards function, thus finding an optimal policy $\pi^*(s)$ such that:

$$\pi^*(s) = argmax_{\pi} U^{\pi}(s)$$

for every $s \in S$.

It is unusual that an agent receives accurate information about reward directly after performing each action – most of the time a future cumulative reward is delayed. For example, in any board game, it is hard to estimate how a single move will affect the overall game state. In such case, an agent may not receive any reward information until the end of the game. Thus, the learner must propagate the delayed reinforcement to previous states and actions that (implicitly) caused this reinforcement. This is known as *temporal credit assignment* problem [57].

The agent is not told in advance what action to perform in a given state, but based on past experiences and new options (called accordingly exploitation and exploration) it selects the action to execute. This process is known as *trial-and-error learning*. This leads to the *exploration-exploitation dilemma* – how long an agent should explore and when it should start exploiting its knowledge. In other words, what should be the trade-off between exploration and exploitation?

Reinforcement learning methods are especially useful when precise domain knowledge is costly to obtain (or even it is not available at all). In particular, reinforcement learning is a promising approach to learn to play games. One of the most successful approaches to use reinforcement learning in game playing is Tesauro's TD-Gammon [61] – a program taught to play Backgammon entirely by reinforcement learning and self-play, which achieved super-human level of play. There have been also numerous other notable successes of reinforcement learning in learning to play games, such as checkers [46], Othello [58][17], chess [62] and 2048 [59]. Besides board games, reinforcement learning is useful in controlling mobile robots [30][27], improving elevator performance [5][6], optimizing factories operation [28], and many others real-world problems [18].

2.1 Q-Learning

Q-learning is a model-free, temporal difference [55][57] algorithm to learn action-value function for any Markov decision process. In model-free learning, the environment model (state transition probability function $P(s, a, s')$ and the reward function $R(s, a)$) is not used during the learning process – the agent has to learn the model basing on its experience. Q-learning is an off-policy method. It means that it learns the optimal strategy independently of the agent's behavior, in contrast to on-policy methods, which learn the policy being followed by the agent (including non-greedy, exploration steps). An exemplary on-policy algorithm is SARSA (State-Action-Reward-State-Action) [45][40].

Action-value function returns the expected cumulative reward of taking action a in state s when optimal policy will be followed in the subsequent steps. Q-function can also be written in a recursive form, known as Bellman equation:

$$Q(s,a) = R(s,a) + \gamma \max_{a' \in A(s')} Q(s',a'), \tag{2.1}$$

where $R(s,a)$ is an immediate reward for taking action a in state s, γ is a discount factor ($\gamma \in (0,1]$) and s' is a new state after making action a.

When a function fulfilling the Bellman equation is found, optimal policy could be created by taking in each step the action with the highest Q-value [67].

In the learning process, it is expected that Eq. (2.1) will be satisfied. When it is not, Q-value function for state s and action a should be modified according to the following rule:

$$Q(s,a) \leftarrow Q(s,a) + \eta[R(s,a) + \max_{a' \in A(s)} Q(s',a') - Q(s,a)], \tag{2.2}$$

where η ($\eta \in (0,1]$) is the learning rate and the rest of the parameters have the same meaning as in Eq. (2.1).

In the most straightforward implementation of Q-learning, the state-action values are stored in a look-up table (LUT). Q-learning uses Eq. 2.1 to iteratively update the value of $Q(s,a)$ by updating appropriate values in table $Q[S,A]$ where S is a set of possible states and A is a set of all possible actions. During the learning $Q[s,a]$ represents current estimate of $Q^*(s,a)$, where $Q*$ is the optimal state-action value.

Q-learning with LUT algorithm is presented in Listing 2.1.

Listing 2.1: Q-Learning algorithm

1. For each possible state-action pair initialize table $Q[s,a]$ to zero
2. Observe current state s
3. Unless stop condition is satisfied:
    3.1 Select action a (according to utility function for state s):
        $a = argmax_{a'} Q(s,a')$
    3.2 Execute action a
    3.2 Receive immediate reward r.
    3.3 Observe new state s'.
    3.4 Update table entry $Q[s,a]$ using the update rule:
        $Q[s,a] = Q(s,a) + \eta[r + max_{a'} Q(s',a') - Q(s,a)]$
    3.5 Assume s' as a new state s.

In real world problems, keeping table $Q[S,A]$ for each action in each state is not possible, because the state space is too large to consider explicitly each of the states. One has to use approximation function ($\hat{Q}(s,a)$) instead, which advantage is that it also generalizes the acquired knowledge over non-visited states. Such a function approximator is also a compromise between the quality of approximation and the time necessary to learn.

The state is represented as a set of features $f_1, f_2, ..., f_n$, where the number of features is significantly lower than the number of possible states. The Q-function is parametrized by the vectors of parameters θ. During the learning process, only $\theta = (\theta_1, \theta_2, ..., \theta_n)$ parameters are subjected to learning. According to the Widrow-Hoff's [56] rule, parameters should change along the gradient in order to minimize the overall loss L_θ:

$$L_\theta(s,a) = \frac{1}{2}(\hat{Q}(s,a) - Q(s,a))^2$$

The update rule for θ parameters in Q-learning is as follows:

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial L_\theta(s,a)}{\partial \theta_i} = \theta_i - \eta(\hat{Q}(s,a) - Q(s,a)) \frac{\partial \hat{Q}(s,a)}{\partial \theta_i}$$

$$\theta_i \leftarrow \theta_i - \eta(\hat{Q}(s,a) - (R(s,a) + \gamma \max_{a'} \hat{Q}(s',a'))) \frac{\partial \hat{Q}(s,a)}{\partial \theta_i}$$

The approximator $\hat{Q}(s,a)$ could be for example linear combination of state features with factors $\theta_1, \theta_2, ..., \theta_n$:

$$\hat{Q}(s,a) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s)$$

One of possible (nonlinear) approximation functions is an artificial neural network, used successfully in many previous studies [16][10][63]. Input to such a neural network is encoded state and output is Q-value for particular action. Actually, there are at least two options for neural network outputs: neural network could have number of outputs equal to the number of overall possible actions or there could be separated neural network for each action, where each neural network have only one output. This topic will be broadly described in Section 3.3.

2.2 ϵ-greedy Policy

ϵ-greedy policy handles the exploration-exploitation dilemma. A greedy algorithm is an algorithm that always takes whatever action seems to be the best according to the action-value function Q. Greedy agent might fall into a local optima. The ϵ-greedy algorithm is almost a greedy algorithm, because it exploits the best available option, but once in a while, it explores the other available options. Using this strategy, either a random action is selected with probability ϵ or an action defined by agent current policy π is selected with probability $1 - \epsilon$.

Epsilon can be fixed during the learning, but annealing it often leads to better results. For example, the agent can initially start with $\epsilon = 1$ (full exploration – always select a random action), then, progressively with each game played, ϵ is decreased linearly towards ϵ_f value, reaching it after K played episodes. By using linear annealing of ϵ, with each successive episode, the agent uses an action provided by Q-function more frequently (agent is moving forwards to almost full exploitation), ending with only few percent of random actions (e.g., $e_f = 0.01$, which means that one in one hundred actions is selected by random) after a predefined number of episodes.

Chapter 3

Deep Learning

In recent years, the popularity of deep learning has increased enormously in the machine learning community. Deep learning tries to model and extract high-level abstraction features in data using complex, multi-layer models and non-linear transformations. Deep learning methods are used in both supervised and unsupervised learning. In contrast to shallow learning algorithms, deep learning algorithms are distinctive by the number of transformations each signal encounter during its propagation from the input to the output [47]. There are many motivations to explore deep algorithms:

- the ability to learn complex functions,
- the ability to learn high-level, hierarchical abstractions,
- the ability to learn from a very large set of training data, and
- the ability to learn from unlabeled data.

Deep learning was successfully applied in multiple fields such as speech recognition [7][13], image recognition and classification [22][23], natural language processing [48] and many other tasks in machine learning [8]. In all of these studies, the performance of the resulting algorithm was better than other machine learning approaches to these problems.

3.1 Neural Networks

One of the most common approaches in deep learning involves using artificial neural networks (ANN). Artificial neural networks are inspired by biological neural networks (especially model of a human brain) and are used to estimate the value of complex functions, which could depend on numerous inputs.

Artificial neural network, illustrated in Fig. 3.1, is a system of connected artificial neurons, which are mathematical functions imitating biological neurons. Artificial neuron receives one or more signals x as an input (referencing biological neuron's dendrites) and transforms them to produce an output (representing biological neuron's axon). Usually, this transformation is a weighted sum of all inputs processed at the end by a non-linear activation function φ. Neural output for a given input vector x is defined by the following formula:

$$f(x) = \varphi(\Sigma_i w_i x_i)$$

where w_i is weight of input i and x_i is the actual value (of signal) at input i. There are many possible and widely used activation functions, such as hyperbolic tangent ($\varphi(v_i) = tanh(v_i)$), logistic function ($\varphi(v_i) = (1 + e^{-v_i})^{-1}$), rectifier ($\varphi(v_i) = max(0, vi)$), softplus ($\varphi(v_i) = ln(1 + e^{v_i})$; this is smooth approximation of the rectifier) and many others. The main purpose of an

activation function is to introduce non-linearity into the neural network (without non-linearity, neural network, even with infinite number of layers, would behave like a single layer, because sum of linear functions is another linear function as well). Typically, activation functions have a „squashing" effect. Usually, the value +1 is assigned to the x_0 input, which makes it a bias input with $w_0 = b$. A bias value allows to shift the activation function horizontally, which may be critical for successull learning.

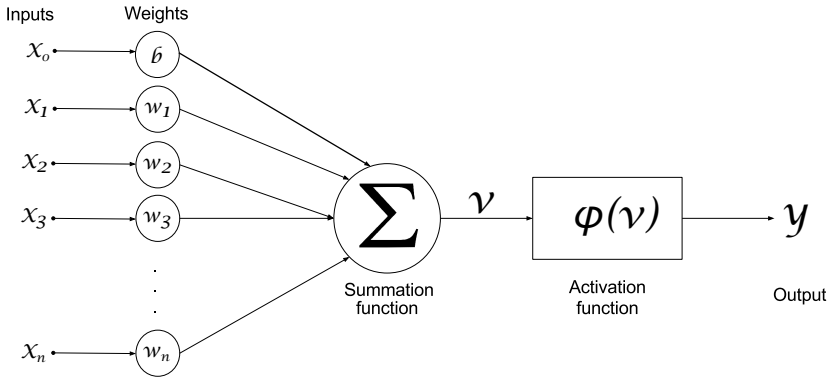

Figure 3.1: Artificial neuron.

The word 'network' refers to the connections between the neurons in different layers of the system. The most popular non-linear feedforward neural networks consists of one input layer, at least one hidden layer, and an output layer. An example network, shown in Fig. 3.2, has three layers. The first layer contains input neurons which send data to the second layer of neurons, and then second layer (hidden layer) send data to the third layer of output neurons. Note that, in this case, subsequent layers are fully connected, which means that output of each neuron in layer X is (one of) the inputs to each neuron in layer $X + 1$.

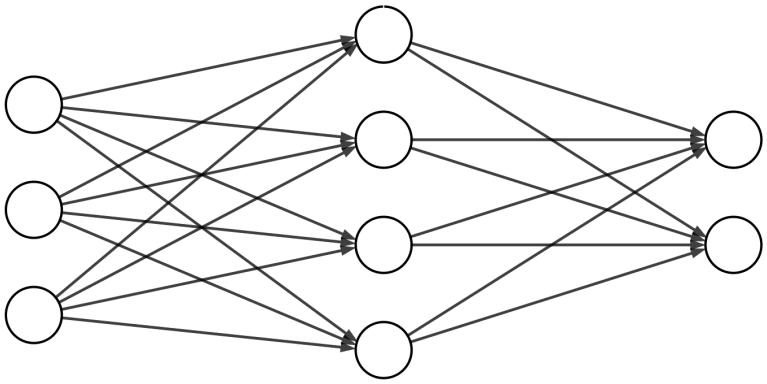

Figure 3.2: Shallow artificial neural network with 1 hidden layer.

The feedforward neural network is ANN in which connection between units form directed acyclic graph (DAG) – there are no (directed) cycles between neurons. In such a network information moves only in one direction – from the input nodes to the output nodes. In this thesis, only feedforward networks are considered.

There are numerous algorithms for training artificial neural networks. Training of a ANN is a process in which ANN's parameters are updated in order to minimize overall error committed by a network. One of the common methods of training is backpropagation, which stands for the backward propagation of errors used together with optimization method, such as gradient descent.

In this method, the derivative of the loss function (also called the cost function) with respect to the network parameters is calculated. Those parameters are, then, updated in a gradient direction. The gradient descent algorithm works by taking the gradient of the weights to find the path of steepest descent. By following the steepest descent at each iteration, it will find a minimum (error), or diverge if the weight space is infinitely decreasing. However, when a minimum is found, there is no guarantee that it is a global minimum – algorithm could get stuck in a local minimum. Note that backpropagation with gradient descent can be applied only to networks with differentiable activation functions [43].

3.2 Deep Neural Networks

Traditional feedforward neural networks are considered to have the depth equal to the number of layers (i.e. the number of hidden layers plus 1, for the output layer). In many simple problems neural networks with depth 2 were successfully used, but there are many cases when single hidden layer is not enough. Solution to this problem is a deep neural network (DNN) – an artificial neural network with at least two hidden layers. The extra layers enable composition of features from lower layers, giving the potential of modeling complex, hierarchial features with a fewer units than in a shallow network. By providing sufficient amount of data to deep neural networks, it is often possible to learn better models than hand-coded features [22].

Before 2006 attempts at training deep architectures failed: training a deep feedforward neural network usually returns worse results than when using shallow architectures. Three papers [14][1][38], published in 2006 and 2007, revolutionized deep architectures. The key principles in all of these papers were:

- unsupervised learning of representations is used to pre-train each layer,
- unsupervised training one layer at a time, using previously trained ones,
- supervised training to tune all layers together.

Deep neural networks are usually designed as basic feedforward networks, but in many recent studies deep learning architecture was created using convolutional deep neural networks (especially in image recognition and processing) [32] [33].

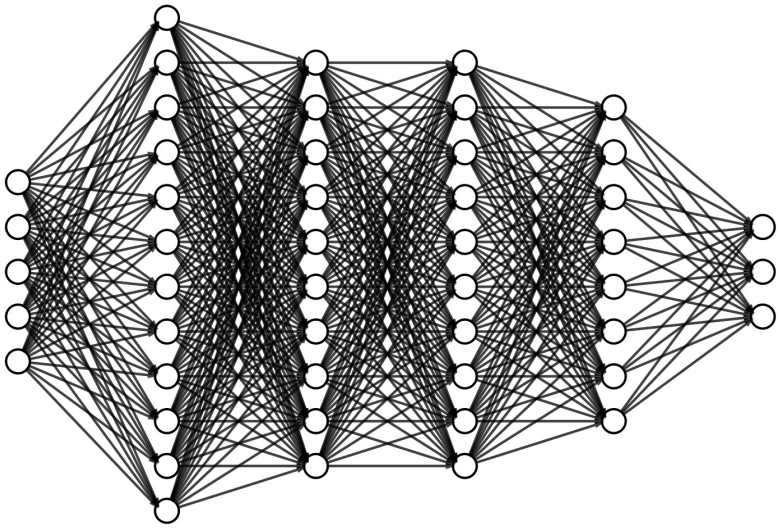

Figure 3.3: Deep artificial neural network with 4 hidden layers.

3.3 Deep Q-Networks

Deep Q-Network is a neural network that approximates Q-value function in reinforcement learning context. In such architecture, the deep neural network with weights w approximate Q-function ($Q(s, a, w) \approx Q^*(s, a)$). The input to this network is a state and the outputs are Q-values for each action (the number of the outputs in the network should be the same as the number of possible actions). Deep Q-Network was introduced by Google DeepMind to successfully learn to play Atari 2600 games [33] and will be the subject of further studies in Chapters 5 and 6.

Reinforcement learning is known to be unstable (or even to diverge) when a nonlinear function approximator such as a neural network is used to represent the Q-function [64]. This instability is caused by correlations between succeeding observations, the fact, that small updates of Q-function may substantially change policy, and the correlations between the Q-value and target values.

3.3.1 Experience Replay and Minibatches

To inhibit first of this issues, biologically-inspired mechanism called *experience replay* is used [26]. In this idea, the agent can experience the effects of its actions without actually executing them. Data used to learning is randomly sampled at each step from memory of agent's previous transitions, what removes the correlation in the sequence of training examples and smooths the training distribution over many past behaviors, thus reducing oscillations and divergence of the learning process. Another advantage of this mechanism is reusing single experience in many weights updates, which allows for greater data efficiency [33] [25]. Note that using experience replay implies off-policy learning (see Section 2.1), because current parameters are different than the ones used to generate experience sample.

To perform experience replay, agent's transitions experience at each time step t are stored in dataset D as a tuple $e_t = (s_t, a_t, r_t, s_{t+1}, i_{t+1})$, where s_t is state observed at time t, a_t is action performed after observing state s_t, r_t is reward given for executing action a_t, s_{t+1} is resulting state after taking action a_t, and i_{t+1} stores information if state s_{t+1} is terminal. The dataset (also called, replay memory) $D_N = \{e_1, e_2, ..., e_n\}$ stores last N experience tuples.

Another technique, used tightly together with experience replay, is *minibatch learning*, which consists in, basically, learning more than one training example at each step. It can perform significantly more efficient than standard, single-sample stochastic gradient descent methods because the code can make use of vectorization libraries or GPU rather than computing each step separately. It makes the learning process also less prone to outliers and noises, as the gradient computed at each step uses more training examples. Generally, determining the gradient of a batch involves computing cost function over each training example in the batch and then, at the end, summing the results of these functions. When planning the size of the minibatch, some trade-offs between efficiency and noisiness have to be made: small size of the minibatch is more susceptible to noise (which may lead to stagnation in local optimum), whereas with large size of the minibatch, the update of single parameter will last longer. Moreover, too large minibatch may end up with bouncing around a local optimum instead of converging to one.

At each step of the training, dataset D is sampled uniformly at random to get minibatch of experiences of size M ($(s, a, r, s', i) \sim U(D)$) and perform learning (weights updates) on them. This solution is, however, limited because the memory does not differentiate important experiences from insignificant ones and always overwrites the oldest transition with the most recent one. Similarly, the uniform sampling gives equal importance to all of the transitions.

Full pseudocode of Deep Q-Learning algorithm with experience replay and minibatches is presented in Listing 3.1.

Listing 3.1: Deep Q-Learning with experience replay and minibatches

1. Initialize replay memory D to capacity N
2. Initialize neural network Q with random weights w_0
3. For each episode e, unless stop condition is satisfied:
    3.1 Get initial state s_0
    3.2 For each time step t in current episode:
        3.2.1 With probability ϵ select random action a_t,
              otherwise select $a_t = argmax_{a'} Q(s_t, a')$
        3.2.2 Execute action a_t
        3.2.3 Receive immediate reward r_t
        3.2.4 Observe following state s_{t+1} and if it is terminal (i_{t+1})
        3.2.5 Store transition $(s_t, r_t, a_t, s_{t+1}, i_{t+1})$ in D
        3.2.6 Sample random minibatch of transitions $(s_j, r_j, a_j, s_{j+1}, i_{j+1})$ of size M from D
        3.2.7 For each sample in minibatch:
            3.2.7.1 Set $y_j = \begin{cases} r_j & \text{if } i_{j+1} \text{ is true} \\ r_j + \gamma max_{a'} Q(s_{j+1}, a', w) & \text{otherwise} \end{cases}$
        3.2.8 Perform backpropagation algorithm using gradient of weights w.r.t loss (calculated error):
              $L = \sum_{j=1}^{M} (y_j - Q(s_j, a_j))^2$

3.3.2 Target Q-Network Freezing

Another modification of the standard Q-learning aimed to further improve its when using neural networks is freezing weights of the target Q-network [33]. More precisely, there are two separated networks maintained:

1. a target network, called $\hat{Q}$ in the following, with a fixed set of old parameters w^- for generating target Q-values, used in the Q-learning process, and
2. a network for interacting with the environment (generating Q-values for each possible action in current state), with the current set of parameters w.

Target Q-values are calculated using the old set of parameters w^-:

$$R(s, a) + \gamma max_{a'} \hat{Q}(s', a', w^-)$$

At every update iteration, the current parameters w are updated to minimize the mean-squared Bellman error w.r.t the old parameters w^- by optimizing the following loss function:

$$L(w) = \mathbb{E}_{s,a,r,s',i \sim D}[(R(s, a) + \gamma max_{a'} \hat{Q}(s', a', w^-) - Q(s, a, w))^2]$$

Every C steps, parameters from the Q-network are copied to the target Q-network ($w^- \leftarrow w$). Generating the learning targets using an older set of weights adds a delay between the time an update to Q is made and the time the update affects the targets, counteracting oscillations and divergence.

3.4 Root Mean Squared Gradient (RMSProp)

Backward propagation of errors (*backpropagation*) [44] is the most commonly used algorithm for multi-layered feed-forward networks learning. The basic idea behind this algorithm is the repeated application of the chain rule to compute the influence of each weight in the network with respect to an arbitrary loss function L. Once the partial derivative for each weight w_i is known, the loss function is minimized by performing the gradient descent w.r.t the learning rate η:

$$w_i = w_i - \eta \frac{\partial L}{\partial w_i} \qquad \text{backpropagation algorithm}$$

The gradient descent method, described in Section 3.1, has several drawbacks which disqualify from using it in minibatch Q-learning. First of all, gradient descent requires to propagate through the neural network all of training samples to make a single update, which slows down the learning process. This is why, nowadays, the most common learning method is stochastic gradient descent (SGD), which works only on a subset of samples at a time, which typically gives a significant performance improvement.

SGD assumes that the learning rate is the same for each parameter being learned, which combined with highly varied gradient value for each parameter does not seem reasonable since for some components it leads to huge changes and tiny changes for the others. When full-batch learning (such as gradient descent) is used, this problem could be solved by using only the sign of the gradient, where the weight updates are all of the same value (the sign of this update is based on the sign of the gradient).

A modification of SGD, called RProp (*resilient backpropagation*)[41], combines the idea of only using the sign of the gradient and the idea of adapting the step size (update value, Δ_i) separately for each weight. The step size is adjusted using η^- and η^+ parameters ($0 < \eta^- < 1 < \eta^+$) based on the consistency of the sign of the gradient over time. Whenever two following gradients of a single parameter have the same sign, the step size for that parameter increases, and whenever the signs disagree, the step size decreases.

$$\Delta_i^{(t)} = \begin{cases} \eta^+ \cdot \Delta_i^{(t-1)} & \text{if } \frac{\partial L}{\partial w_i}^{(t-1)} \cdot \frac{\partial L}{\partial w_i}^{(t)} > 0 \\ \eta^- \cdot \Delta_i^{(t-1)} & \text{if } \frac{\partial L}{\partial w_i}^{(t-1)} \cdot \frac{\partial L}{\partial w_i}^{(t)} < 0 \\ \Delta_i^{(t-1)} & \text{otherwise} \end{cases}$$

$$w_i^{(t)} = w_i^{(t-1)} - sign(\frac{\partial L}{\partial w_i}^{(t)}) \cdot \Delta_i^{(t)} \qquad \text{RProp algorithm}$$

Unfortunately, RProp does not work well with minibatches [11].

RMSProp [11] is an extension to RProp and SGD which combines the robustness of RProp, efficiency of minibatches, and effective averaging of gradients over minibatches (unlike RProp). RMSProp keeps track of previous gradients and divide updates by the average magnitude of the gradient over the last several updates, which leads, in practice, to maintaining separate learning rate η_i for each weight. This allows to modify each parameter w_i according to its previous magnitudes, preventing from taking it into account with too large or too small weight. RMSProp could be parametrized by ρ (gradient moving average decay factor; usually $\rho = 0.95$ is used), overall learning rate[1] (η) and ε (small value added for numerical stability; usually $\varepsilon = 1 \times 10^{-6}$ is used). In the RMSProp algorithm the overall weight w_i update at time step t is calculated as follows:

$$g_i^{(t)} = \rho g_i^{(t-1)} + (1-\rho)(\frac{\partial L}{\partial w_i}^{(t)})^2 \qquad \text{moving average of squared gradient from the past}$$

$$\eta_i^{(t)} = \frac{\eta}{\sqrt{g_i^{(t)} + \varepsilon}} \qquad \text{learning rate for particular weight}$$

$$w_i^{(t)} = w_i^{(t-1)} - \eta_i^{(t)} \frac{\partial L}{\partial w_i}^{(t)} \qquad \text{weight update}$$

[1] In practice, overall learning rate is linearly annealed from η_s to η_f over η_e episodes

In this thesis the classical RMSProp algorithm is compared to Google's DeepMind version of RMSProp [33], which, besides keeping squared gradients from the past, keeps track of regular gradients as well:

$$g_i^{(t)} = \rho g_i^{(t-1)} + (1-\rho)(\frac{\partial L}{\partial w_i}^{(t)})^2 \qquad \text{averaging squared gradient from the past}$$

$$h_i^{(t)} = \rho h_i^{(t-1)} + (1-\rho)\frac{\partial L}{\partial w_i}^{(t)} \qquad \text{averaging gradient from the past}$$

$$\eta_i = \frac{\eta}{\sqrt{g_i^{(t)} - (h_i^{(t)})^2 + \varepsilon}} \qquad \text{learning rate for particular parameter}$$

$$w_i^{(t)} = w_i^{(t-1)} - \eta_i^{(t)}\frac{\partial L}{\partial w_i}^{(t)} \qquad \text{weight update}$$

Chapter 4

Keepaway Soccer

Keepaway soccer is a one of many sub-tasks of robot soccer[29]. In keepaway soccer, one team, the keepers, is trying to maintain possession of the ball within a limited region (a rectangle field), while another team, the takers, is trying to gain possession of the ball. The episode ends when the takers gain possession of the ball or the ball leaves the region [50][52][53]. Keepaway soccer is a perfect candidate for machine learning benchmark problem because, on the one hand, it is simple enough that could be successfully learned without enormous time and cost effort, and, on the other hand, it is complex enough. There are many reasons why keepaway soccer is a challenging machine learning task:

- The state space is continuous, so it is infeasible to explore it completely,
- There is hidden state – each agent see only part of the environment (what sensors return),
- Multiple agents need to learn at the same time,
- There is long and variable delay in effect of each action,
- Agent have noisy sensors and actuators – neither it is perceiving the world exactly as it is nor it cannot affect the world exactly as intended,
- Perception and action are asynchronous - action is not direct response to perception trigger.

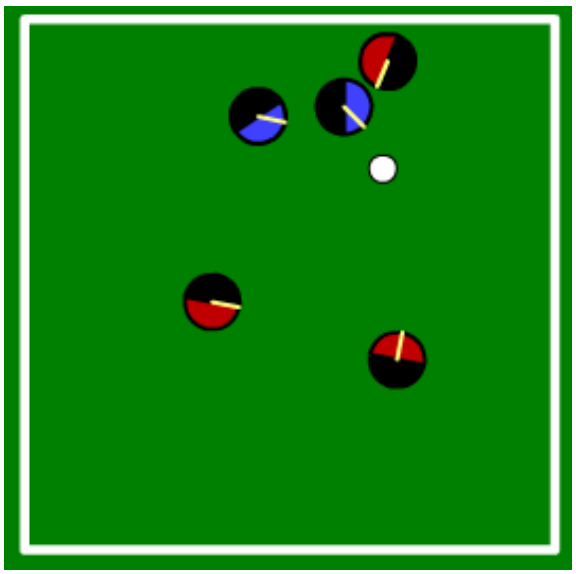

Figure 4.1: Visualization of 3 $vs.$ 2 keepaway play in $20\,\mathrm{m} \times 20\,\mathrm{m}$ region.

4.1 Soccer Server

Since late 2002, the keepaway task has been part of the official release of the open source Soccer Server used at RoboCup [29], the annual international robotics competition, which aim is to promote robotics and artificial intelligence research. Each agent in the simulator receive a visual signal every 150ms, which contains information about relative distances and angles to the other objects in the world, such as the ball and the agents. Every agent may execute one of a primitive, predefined, parametrizable actions, such as *turn(angle)*, *dash(power)* or *kick(power, angle)* every 100 ms (the RoboCup soccer simulator operates in discrete time steps, $t = 0, 1, 2, ...$, each one representing 100 ms of simulated time). Note that signal with current environment state and agent action happen asynchronously and are (relatively) independent – the action is not a direct response to the visual perception. Every state description variable and action executed by agent may contain random noise which makes the game not deterministic. A noise is injected by the simulator, without external control. Every agent must be controlled by a separate process, with no inter-agent communication permitted other than via the simulator itself. Full details of the simulator are presented in the server manual [4].

When started in a special mode, the simulator enforces the rules of the keepaway task instead of the rules of the full soccer. In particular, the simulator places the players at their initial positions at the start of each episode and ends an episode when the ball leaves the play region or is taken away. In this mode, the simulator also informs the players when an episode has ended and produces a log file with the duration of each episode. This log file is later used as a source to method analysis and rating. An example log file is presented in Listing 4.1.

Listing 4.1: Example keepaway soccer output log

```
# Keepers: 3
# Takers:  2
# Region:  20 x 20
#
# Description of Fields:
# 1) Episode number
# 2) Start time in simulator steps (100ms)
# 3) End time in simulator steps (100ms)
# 4) Duration in simulator steps (100ms)
# 5) (o)ut of bounds / (t)aken away
#
1 0 112 112 o
2 112 237 125 o
3 237 606 369 o
4 606 633 27  o
5 633 779 146 o
6 779 940 161 o
7 940 1154  214 o
8 1154  1259  105 t
9 1259  1318  59  t
10  1318  1495  177 t
...
```

Soccer server may be started in synchronous mode, which significantly speeds up game simulation by executing all actions one-after-another (without unnecessary delays between actions) instead of executing them in real time. This mode is the recommended one for learning purposes.

Since keepaway soccer is simplification of the full RoboCup soccer task, it is appropriate for direct comparison of different machine learning methods. There are several simplifications in comparison of the full task[52]:

- there are fewer players involved in the game (usually 5 instead of 22),
- players are playing in a smaller region (it's clipped section of a full-size field),

- each player is focused on single high-level goal, either offensive or defensive, but never both.

This work uses version 15.2.2 of the standard RoboCup soccer simulator [29] and version 0.6 of keepaway soccer benchmark player framework [37], adapted and extended by Tom Palmer.

4.2 Environment

Keepaway soccer game is highly parametrizable. The default (and the mostly used for research purposes) configuration is 3 keepers and 2 takers (called also 3 *vs.* 2) playing in 20 m × 20 m region. There are also other popular player configurations such as 4 *vs.* 3 and 5 *vs.* 4 (note that usually there are K keepers and $K - 1$ takers) and region sizes such as 15 m × 15 m, 25 m × 25 m and 30 m × 30 m. Sample visualization of 3 *vs.* 2 game in 20 m × 20 m region is presented in Fig. 4.1.

4.2.1 Keepaway Episode

At the beginning of each episode, the coach resets all players (keepers and takers) randomly in the set of predefined locations. All takers start in one predefined corner of the region (bottom-left). Three randomly chosen keepers are placed one in each of the three remaining corners (top-left, top-right and bottom-right). The rest of the keepers are placed in the center of the region. The ball is placed next to the keeper in the top-left corner. Keeper when with possession of the ball executes action every 100 ms (of simulator time). When ball leaves the region or any of the takers take over the ball, the coach ends the current episode and starts the new one.

4.2.2 Actions

The learning agent may choose final action from the set of mid-level actions, constructed from basic skills. Peter Stone and Rober Sutton in [54] defined these actions as follows:

- **HoldBall()**: Remain stationary while keeping possession of the ball in a position that is as far away from the opponents as possible,
- **PassBall(k)**: Kick the ball directly towards keeper k,
- **GetOpen()**: Move to a position that is free from opponents and open for a pass from the ball's current position (using SPAR [65]),
- **GoToBall()**: Intercept a moving ball or move directly towards a stationary ball,
- **BlockPass(k)**: Move to a position between the keeper with the ball and keeper k.

All of these actions, excluding PassBall(k), are primitive functions mapping state to action and usually control behavior in single time step. PassBall(k) may be, however, executed during multiple time steps (this action is composed of getting the ball into position for kicking, and then a sequence of kicks in the desired direction). Moreover, every action (even primitive one) could last more than one time step, because of the agent actuators noises – player occasionally misses the following step (which means that previous action is kept) or the simulator misses command [54].

4.3 Standardized Keepaway Task

In the RoboCup soccer simulator, agents have limited and noisy sensors: each player can see objects within a 90° view angle and the precision of an object's sensed location degrades with distance. To

simplify the problem, the standardized keepaway player (keeper or taker) has full, 360° view angle (but still with random noise) [52].

From the learning algorithm perspective, keepaway problem is presented as SMDP (*semi-Markov decision process*), thus it maps straightforward to reinforcement learning problem. From the SMDP point of view, the episode consists of a following sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, ..., s_i, a_i, r_{i+1}, ..., r_n, s_n,$$

where s_i is simulator state at time t_i, a_i is action executed by agent at time t_i, r_{i+1} and s_{i+1} are resulting reward and state after execution of action a_i. The final state of the episode, s_n is the state where the takers have possession or the ball has gone out of region bounds. Reward in time step t_i is calculated as a difference between current time step and the last agent was in the possession of the ball: $r_i = t_i - t_{i-1}$ (if takers learning process is considered, reward is inverse of the above: $r_i = t_{i-1} - t_i$).

4.3.1 State Variables

To not process raw positional information, but rather higher-level features, state representation is containing distances and angles of the keepers $K_2, ..., K_n$, the takers $T_1, ..., T_m$ and the center of the playing region. For 3 *vs.* 2 keepaway problem, there are 13 variables defining the state (see Fig. 4.2):

- distances from the players to the center of the region,
- distances from K_1 (keeper with the possession of the ball) to other keepers,
- distances from teammates to their closest opponent,
- for each keeper K_i ($i = 2, ..., n$), the minimal angle with the vertex at K_1 between K_i and an opponent.

which could be presented more formaly as:

- $dist(K1, C), dist(K2, C), dist(K3, C), dist(T1, C), dist(T2, C)$
- $dist(K1, K2), dist(K1, K3)$
- $dist(K1, T1), dist(K1, T2)$
- $Min(dist(K2, T1), dist(K2, T2))$
- $Min(dist(K3, T1), dist(K3, T2))$
- $Min(ang(K2, K1, T1), ang(K2, K1, T2))$
- $Min(ang(K3, K1, T1), ang(K3, K1, T2))$

where $dist(a, b)$ is euclidean distance between object a and b and $ang(a, b, c)$ is angle between object a and c with vertex at object b. This list could be generalized to additional players, which leads to linear growth in the number of state variables (for example there will be 19 variables in case of 4 *vs.* 3 keepaway. General formula for the number of problem variables is $4K + 2T - 3$, where K is the number of keepers and T is the number of takers).

4.4 Keepaway Agent

4.4.1 Standardized Keepaway Framework

To simplify future researches and make results of different artificial intelligence methods directly comparable, Peter Stone, Gregory Kuhlmann, Matthew E. Taylor and Yaxin Liu created in 2006

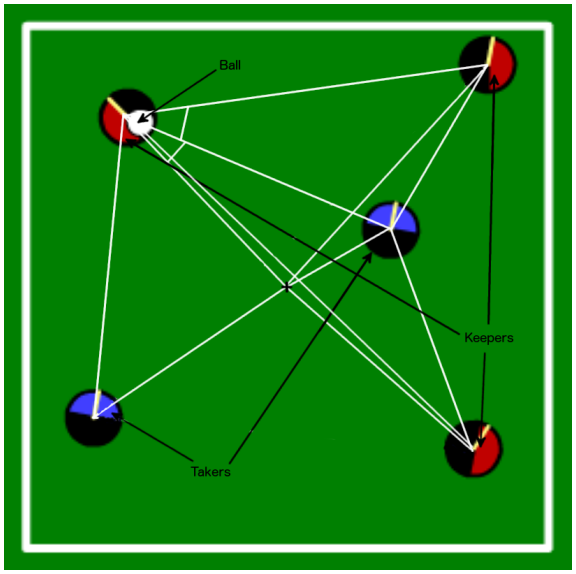

Figure 4.2: 13 state variables (marked using segments and angles in white) in 3 $vs.$ 2 keepaway problem [51].

standardized keepaway player framework [51]. This framework handles communication and synchronization with the server, world model update, localization, and low- and mid-level action. The keepaway player implementation is constructed in such a way that the details of the keepaway domain are completely abstracted from the high-level action selection. This allows new learning algorithms to be integrated with minimal effort.

Standardized keepaway player framework considers only keeper learning process when in possession of the ball (possession of the ball is not precisely defined in soccer server simulator, so there is an assumption, that player is in possession of the ball when he is close enough to kick it). Keepers not in the possession of the ball are enforced to execute predefined, *Receive* macro-action, which can be considered as action defined in section 4.2.2 under the following condition [54]:

- **Receive**: If teammate could get to the ball faster than this keeper (or optionally is in the possession of the ball), call *GetOpen()*, else call *GoToBall()*. Repeat this action until in possession of the ball or episode ends.

The keeper in possession of the ball has choice of n possible macro-actions (where n is the number of keepers): it may hold the ball or pass it to one of its teammates. These action choices are precisely defined as {Holdball, PassK_2ThenReceive, PassK_3ThenReceive, ..., PassK_nThenReceive}. Holdball action is simply *HoldBall()* executed for one or more steps, when PassK_iThenReceive action is defined as follows:

- **PassK_iThenReceive**: Call PassBall(k) to kick the ball toward teammate k. Then act as in the Receive action.

The keepers are numbered by their closeness to the keeper which is in possession of the ball: K_1 is the keeper with the ball, K_2 is the keeper which is closest to it, K_3 the next closest, and so on up to K_n, which is the keeper which is farthest from keeper K_1.

4.4.2 Learning Agent Interface

Standardized keepaway player interface was implemented in C++. New learning algorithm has to implement the SMDPAgent interface, overwriting following functions [51]:

- **int startEpisode(double state[])** – this function is called when player is in the possession of the ball for the first time in episode (it is not receiving any reward for previous actions). Notice that, if the player never obtains the ball during the episode, this function will not be called.
- **int step(double reward, double state[])** – this function is called every time when player is in the possession of the ball after the first time. The reward is accumulated reward during execution of previous macro-actions (which may be triggered by other players as well).
- **void endEpisode(double reward)** – this function is called for every agent (even if this player never touches the ball), when player receives signal from server, that the episode is ended. Reward, again, is accumulated reward after execution of last action.

startEpisode and *step* functions expect agent to return macro-action to execute, which is represented as an integer ranging from 0 to n. The state is represented as a fixed-length array of continuous, floating point values. The reward is represented as a single real value.

4.4.3 Predefined Policies

Keepaway framework does not come with any learning code, but there are few fixed policies for keepers included:

- **Random** – choose randomly from the n actions, each with probability $\frac{1}{n}$,
- **Always Hold** – always choose the Holdball() action,
- **Hand-coded** – if no taker is within 10 m, choose HoldBall(); else if teammate k is in a better location than the keeper with the ball and the other teammates, and the pass is likely to succeed, then choose PassBall(k); Else choose HoldBall(). More details could be found in [52] and [53].

Almost the same policies are defined for takers:

- **Random-T** – choose randomly from the m actions, each with probability $\frac{1}{m}$,
- **All-to-ball** – always choose the GoToBall() option,
- **Hand-coded-T** – if fastest taker to the ball, or closest or second closest taker to the ball: choose the GoToBall() option; else let k be the keeper with the largest angle with vertex at the ball that is clear of takers: choose the BlockPass(k) option.

Note that in 3 *vs.* 2 keepaway task, All-to-ball and Hand-coded-T policies are equivalent.

For all learning and testing purposes of the keepers, Hand-coded-T policy was used for the takers.

4.5 Previous Studies

Keepaway soccer has been widely used as a benchmark for learning systems of different types [51][68][19]. The majority of research in this domain has been from the perspective of reinforcement learning (function approximation) and neuroevolution, but there are other approaches like genetic programming as well. Each of the (relevant) previous studies of keepaway soccer will be briefly described in this section.

4.5.1 Reinforcement Learning: SARSA

Since the keepaway problem maps fairly directly to discrete-time, episodic, reinforcement learning problem (cf. Section 2), most of studies to date involve different kinds of function approximation

together with reinforcement learning.

Stone et al. [51] used temporal difference learning algorithm called SARSA(λ), which stands for State-Action-Reward-State-Action, with different function approximators: linear tile-coding (CMAC), radial basis functions (RBF) and neural networks. SARSA is a similar approach to Q-learning, with the only difference that SARSA is on-policy while Q-learning is an off-policy algorithm [40]. The difference lies in the way that the future reward is calculated – in Q-learning, it is the value of the best possible action that can be performed from the resulting state (s'), while in SARSA, it is the value of the actual action (a') that was taken in s':

$$Q(s,a) \leftarrow Q(s,a) + \eta[R(s) + Q(s',a') - Q(s,a)] \tag{4.1}$$

4.5.1.1 CMAC

CMAC is a form of linear tile-coding [40]. The tile-coding feature sets are formed from multiple overlapping tilings. The state variables s are used to determine the activated tile in each of the different tilings. Every activated tile adds a weighted value to the total output of the CMAC for the given state (Q). The function approximation is learned by changing how much each tile contributes to the output of the function approximator. This method allowed to discretize the continuous state space by using tilings and generalization using multiple overlapping tilings at the same time.

Using CMAC as SARSA function approximator, Stone et al. [51] achieved 15.69 s (with the standard deviation of 2.81 s) of keeper possession time averaged over 24 trials after 30 simulator hours of training. However the results of this method have a very high variation, starting at $\sim$ 11 s, ending at $\sim$ 20.5 s.

4.5.1.2 RBF

An another approach to keepaway soccer involves radial basis function (RBF) [40], which is a generalization of the CMAC to a continuous function. Stone et al. [51] used Gaussian radial basis functions, where $\phi(x) = \exp(-\frac{x^2}{2\sigma^2})$. The state-action values are computed as a sum of one-dimensional RBFs, one for each state variable, similarly to CMAC.

Using this approach, 14.23 s ($\pm$3.14) result was achieved, with 40 trials after 30 simulator hours. Comparing to CMAC, RBFs trials are learning faster and the best-case trials are comparable to CMAC results, but a lot of RBF attempts land at the worst-case end ($\sim$ 12 s).

4.5.1.3 Neural Network

Stone et al. [51] tried also neural networks with architecture $13-20-1$ as a function approximator. The inputs to the network were state variables and each network output was the action value Q (there was a separated network for each possible action). The nodes in the hidden layer have a sigmoid transfer function, while the output nodes have linear activation function. The standard backpropagation method was used to modify networks' weights.

The average possession time over 20 trials after 30 hours of learning was 10.13 s ($\pm$0.29). Neural networks do not learn as fast as CMACs and RBFs but have a significantly lower variance.

4.5.2 NEAT

NeuroEvolution of Augmenting Topologies (NEAT) [49] is a genetic algorithm that evolves both the weights and the architectures of artificial neural networks. A population of genomes is evolved by evaluating each one and selectively reproducing the fittest individuals through crossover and mutation. NEAT combines the usual search for the network weights with an evolution of a network structure. NEAT was successfully applied in keepaway soccer as a policy search method [60][68].

In this approach, the inputs to the network describe the agent's current state (like in neural network method in 4.5.1.3), but there is one output for each available action. The agent performs whichever action has the highest value on its output. A candidate policy is evaluated by allowing the corresponding network to control the agent's behavior and observing how much reward it receives. The policy's fitness is the sum of the rewards the agent accumulates while being under the network's control.

In this method, the average learned possession time over 5 runs after 840 simulator hours of training was 14.1 s (±1.75).

Several more sophisticated approaches to neuro-evolution have been applied to the keepaway task, including EANT [31] and HyperNEAT [66], which achieved respectively 14.9 s (±1.25) after 200 hours training (with mean best run equal to 16.6 s and mean worst run equal to 12.9 s) and 15.4 s (±1.31) average time possession of the ball by the keepers.

4.5.3 SBB

SBB, which stands for symbiotic bid-based genetic programming (GP) is recently proposed [24] algorithm for evolving teams of programs hierarchically using two phases of evolution: one to build a set of candidate policies and a second to learn how to deploy this set consistently. A control policy is defined by a team of simple programs that are coevolved, each specializing on a subcomponent of the task. The general idea of this algorithm is to start from simple policies in initialization phase and incrementally introduce more complexity during the progress of the task.

Kelly et al. [19] performed two experiments: with a maintenance of genotypic diversity and without it. Both experiments have two phases of evolution: the first phase was hierarchical SBB with phenotypic fitness sharing, the second one was hierarchical SBB without fitness sharing. The method with maintenance diversity outperformed the approach without genotypic diversity achieving 18.5 s (±1.9) as an average of 24 independent runs [1], where each lasts over 1700 hours of the simulator time.

4.5.4 Summary

Summarized result of previous studies[2] in keepaway soccer subject are presented in Table 4.1.

Algorithm	**Avg. keeper possession time [s] (± standard deviation)**	**Simulated hours**
Always Hold [50]	2.9 ± 1.0	-
Random [50]	5.3 ± 1.8	-
SARSA with Neural Network [51]	10.1 ± 0.3	30
Hand-coded [50]	13.3 ± 8.3	-
NEAT [60]	14.1 ± 1.8	800
SARSA with RBF [51]	14.2 ± 3.1	30
EANT [31]	14.9 ± 1.3	200
HyperNEAT [66]	15.4 ± 1.3	50-200
SARSA with CMAC [51]	15.7 ± 2.8	30
SBB with diversity [19]	18.5 ± 1.9	1739

Table 4.1: Keepaway soccer task studies and results.

[1] Average and standard deviation calculated according to the results in plots et al. [19]

[2] There were numerous other approaches to keepaway soccer problem, such as evolutionary algorithms [9], evolution of tree-structured GP, with and without ADFs [12][15], but different keepaway Soccer framework was used as a benchmark, so the results are not directly comparable.

Chapter 5

Implementation

One of the goals of this thesis was to develop efficient and robust, but at the same time flexible Deep Q-Learning algorithm. In this chapter overall system architecture and implementation details are shortly presented.

Basic keepaway soccer framework, presented in chapter 4, as well as RoboSoccer server, are implemented in C++. Due to the complicated architecture of this framework and low flexibility of the C++ language, the learning algorithm is implemented in Python[1].

5.1 System Architecture

The system consists of two programs running as two separated processes: keepaway soccer server and Deep Q-Learning agent. The processes are communicating with each other using zeromq[2] message broker in request-reply mode. Since exchanging data between programs in two different programming languages is not trivial, information send through message queue is serialized and deserialized using Google's protobuf library[3]. Overall architecture is presented in Fig. 5.1.

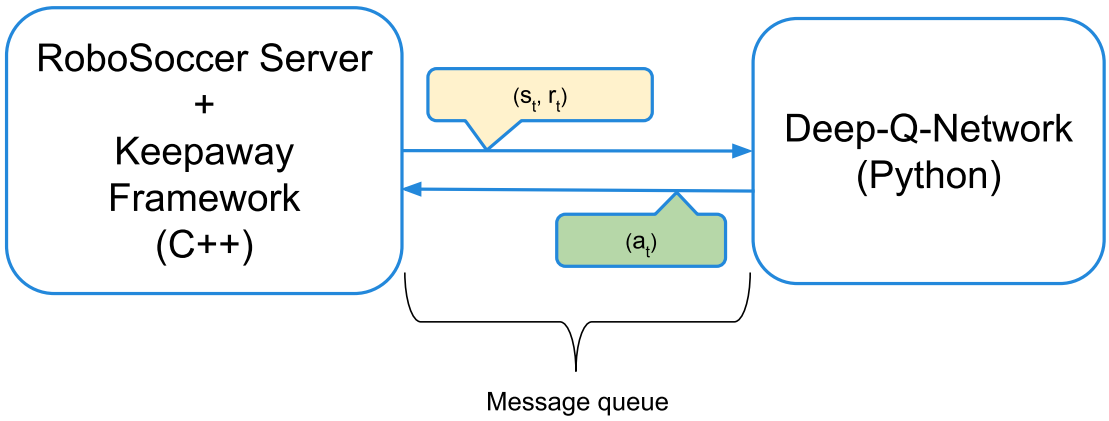

Figure 5.1: The architecture of software framework for reinforcement learning in keepaway soccer.

RoboSoccer Server and keepaway soccer framework has many external dependencies (for example C++ and Fortran compilers, QT environment, parser generator Bison, CUDA libraries and

[1] https://www.python.org/
[2] http://zeromq.org/
[3] https://developers.google.com/protocol-buffers/

many others), so to simplify installation process among many machines, a Docker[4] image was created, with all necessary libraries bundled in. This image, with keepaway environment and Deep Q-Learning agent[5], is available to download from Docker Hub[6].

5.2 Deep Q-Learning Agent

The Deep Q-Learning agent is implemented in Python using Theano[7] and Lasagne[8] libraries. Theano is a Python library that allows to conveniently define, optimize, and evaluate mathematical expressions and compile them to an efficient native code (also to GPU), while Lasagne is library built on top of Theano which allows to create and train neural networks easily.

The python agent receives signal from the keepaway framework (using zero-mq and protobuf) with information about current state, reward, current (simulator) time and call type (*startEpisode*, *step* and *endEpisode*; see 4.4) for details) and responds with the action if it is not end of the episode (using, again, zero-mq and protobuf).

The keepaway Deep Q-Learning agent has the following abilities:

- can use different weights update function (conventional RMSProp and version tweaked by DeepMind; see 3.4),
- can update target network with predefined frequency (see 3.3.2 for details),
- can utilize different neural network architectures (parametrized the number of layers and the number of neurons in each layer) – each hidden layer is dense with the rectifier activation function ($f(x) = max(0, x)$),
- can be parametrized by the size of experience memory and minibatch size.

The tuples of agent experience are not stored in raw form $(s_t, a_t, r_t, s_{t+1}, i_{t+1})$ – there are 4 separated circular arrays to store states, actions, rewards and information whether state was terminal or not. When the current index pointer reaches the end of the array, it is reset to 0 and an array is being overwritten from the beginning (overwriting the oldest experience).

The agent has also the ability to merge multiple consecutive states into one, meta-state, which could later be used to learn based on few following states instead of single one in order to cope with the partial observability.

Overall, the agent's algorithm is presented in listing 5.1. There are sometimes exceptions from this algorithm, for example, when target neural network is replaced by trained one there is additional step 3.3.

Listing 5.1: Deep Q-Learning keepaway agent

```
1. Receive current state, the reward for previous action and information if the state is terminal.
2. Save transition in experiences memory.
3. If training:
  3.1 Get random minibatch from memory.
  3.2 Calculate weights changes using RMSProp and apply them.
4. Get next action using ϵ-greedy policy.
```

[4] https://www.docker.com/
[5] Source code of Deep Q-Learning agent for keepaway soccer is available at https://github.com/mkurek/keepaway
[6] https://hub.docker.com/r/mkurek/keepaway/
[7] http://deeplearning.net/software/theano/
[8] http://lasagne.readthedocs.org/en/latest/index.html

5.3 Parameters

The full list of agent parameters is shown in Table 5.1. These parameters are usually set using environments variables (it is the simplest way to pass parameters to docker container), although runtime arguments could be used as well.

Parameter	Default value	Symbol	Description
NETWORK_ARCHITECTURE	13,30,3	-	neural network architecture (layers are separated by comma; each value is the number of neurons in each layer)
MINIBATCH_SIZE	1	M	the size of the minibatch
TRANSITIONS_HISTORY_SIZE	10000	N	experience memory size (1 means that replay memory stores only the most recent experience)
RECENT_STATES_TO_NETWORK	1	-	how many consecutive states use to form meta-state
DISCOUNT_FACTOR	0.99	γ	Q-learning discount factor
FINAL_EPSILON_GREEDY	0.01	e_f	final epsilon after linear annealing
START_LEARNING_RATE	0.0001	η_s	initial overall learning rate
FINAL_LEARNING_RATE	0.00005	η_f	final overall learning rate
LEARNING_RATE_CHANGE_EPISODES	5000	η_e	number of episodes over which learning rate is linearly annealed
RMSPROP_RHO	0.9	ρ	gradient momentum used by RMSProp
START_LEARN_AFTER	100	-	number of episodes after which learning process starts (previous episodes are played to fulfill experience replay memory)
EXPLORATION_TIME	5000	K	number of episodes over which ϵ is linearly annealed to ϵ_f
STOP_AFTER_EPISODES	10000	-	when to stop learning process
UPDATE_RULE	*rmsprop*	-	which weights update rule use (possible choices: *deepmind_rmsprop* or *rmsprop*)
SWAP_NETWORKS_EVERY	0	C	target networks update frequency (in frames; 0 means that target network update mechanism is turned off)

Table 5.1: Deep Q-Learning keepaway agent available parameters.

Chapter 6

Experiments and Results

In this chapter, computational experiments and their results are presented. These experiments were conducted to compare different reinforcement deep learning techniques and improvements on the keepaway soccer problem. The chapter is organized as follows. First, in Section 6.2, a preliminary study of learning parameters is described. Next, in Sections 6.3-6.7, different deep learning techniques (described in Chapters 2 and 3) are investigated. Finally, in Section 6.8, the best policy found in the previous chapters is presented, evaluated and used in a team learning experiment.

6.1 Experimental Setup

In all experiments, the following definitions are used:

- *episode duration* is the time of a single game (episode) in simulator time units (measured in seconds),
- *performance* (of an agent) is the average episode duration,
- *learning time* is the average time of learning process measured in real time units (hours).

For all experiments in this thesis, the following assumptions were made (unless stated otherwise):

- Keepaway setup: 3 *vs.* 2 playing on 20 m × 20 m region, with the *hand-coded* policy for takers.
- Each experimental run was repeated 6 times – the performance of a given method is the average over results of those runs.
- Each experiment was conducted using only CPU (Intel® Core™ i7-950 3.7GHz) – at the time of writing this thesis access to computing units with CUDA GPU was not possible, which does not affect overall results (the only factor which could be improved on GPU over CPU is learning time).
- Each run ends after 25000 of episodes, out of which the learning process lasts 20000 episodes. The remaining 5000 episodes were spent for evaluating the current performance of the team. To this aim, every 500 episodes, 100 evaluation episodes were conducted. At the end of the run, a final evaluation of 1000 episodes was conducted.
- 25000 episodes last 40-55 hours of the simulator time. It depends on the average length of the episode but does not from the time it takes AI to make decision or to update its parameters. Therefore, instead of reporting simulator time, approximate comparison of learning time will be conducted to test how changing the values of the algorithm parameters affects overall time it takes to perform a single run during experiment.

- Plots of trial performance will present the mean episode duration averaged over 100 episodes (and the learning runs) every 1000 episodes. Last point (at 20000 episodes) is the final method's performance, obtained by averaging the duration of 1000 episodes (and the learning runs).

6.2 Preliminary Experiments to Find Learning Parameters

The preliminary experiments were conducted to obtain the best values for learning parameters. It concerned the following parameters:

- *exploration time* and the overall *experiment time* (*EXPLORATION_TIME* and *STOP_AFTER_EPISODES*, respectively),
- *learning rate* (*START_LEARNING_RATE* and *FINAL_LEARNING_RATE*),
- *discount factor* (*DISCOUNT_FACTOR*),
- initial number of episodes to play to fulfill experience replay memory (*START_LEARN_AFTER*),
- *neural network architecture* (*NETWORK_ARCHITECTURE*).
- ϵ after exploration (*FINAL_EPSILON_GREEDY*).

The preliminary experiments were conducted by trial-and-error and by fiddling around some learning parameters. Some of the tested values are shown in Table 6.1.

Parameter	Tested values
NETWORK_ARCHITECTURE	13-30-100-50-3, 13-30-100-100-50-3, 13-50-30
DISCOUNT_FACTOR	0.99, 1.0
START \ FINAL_LEARNING_RATE	0.001, 0.0005, 0.0001, 0.00005, 0.00001
LEARNING_RATE_CHANGE_EPISODES	5000, 10000, 15000
EXPLORATION_TIME	5000, 10000, 15000, 25000
STOP_AFTER_EPISODES	10000, 20000, 25000, 35000
FINAL_EPSILON_GREEDY	0.1, 0.05, 0.01

Table 6.1: The values of the tested parameters.

The results are presented in Table 6.2. These parameters will be used in subsequent sections as the base learning parameters (unless stated otherwise). The preliminary experiments showed, for example, that increasing exploration time over 15000 and increasing *STOP_AFTER_EPISODES* over 25000 episodes does not improve the final result. Moreover, it turned out that the agent learns best when discount factor is equal to 1.0, final epsilon is 0.01 and the learning rate is constant and equal to 0.0001. The temporary arbitrary choice for the best neural network architecture is 13-30-100-50-3 – several neural network architectures will be broadly investigated in Section 6.6.

Parameter	Value
NETWORK_ARCHITECTURE	13-30-100-50-3
DISCOUNT_FACTOR	1.0
START_LEARNING_RATE	0.0001
FINAL_LEARNING_RATE	0.0001
LEARNING_RATE_CHANGE_EPISODES	15000
START_LEARN_AFTER	300
EXPLORATION_TIME	15000
STOP_AFTER_EPISODES	25000
FINAL_EPSILON_GREEDY	0.01

Table 6.2: The best parameters found in the preliminary experiments.

6.3 Experience Replay and Minibatches

6.3.1 Experiment Setup and Objective

In the first experiment, various experience memory sizes and minibatch sizes were compared. The purpose of this experiment is to investigate whether minibatch learning performs better than on-line learning (with memory size equal to 1), how the size of the minibatch affects the overall performance (both, the average score and the simulation time), and whether the claims made in Section 3.3.1 that using experience replay and minibatch learning increase the stability of the learning process are correct.

6.3.2 Results and Discussion

Memory size	Minibatch size	Agent performance [s]	Learning time [min]
1	1	8.3 (±1.5)	192
10000	1	6.2 (±0.7)	154
10000	4	14.2 (±3.3)	293
10000	8	14.3 (±3.0)	344
10000	16	15.4 (±1.5)	405
10000	32	17.8 (±1.8)	500
10000	64	18.1 (±1.3)	750
10000	128	17.7 (±1.0)	620

Table 6.3: Experience replay and minibatch size experiment results.

The results of the experiment are shown in Table 6.3. As expected, increasing the size of the minibatch increases the average score, but also increases (to minibatch of size 64 linearly) the learning time (see Fig. 6.2). The decrease of learning time between minibatch size 64 and 128 might be caused by slower progress of the learning process for minibatch size 128, thus lower average episode duration during the whole process. Initially, the experience replay with minibatches almost doubles the performance over the on-line learning (8.3 s vs. 14.2 s), but then the difference between consecutive trials with doubled minibatch size decreases (0.3 s on average between experiments with 32 and 64 minibatch size). Minibatch size 64 seems to be the point of inflection in this experiment – both 32 and 128 minibatches lead to a slightly worse performance. Note that for almost every trial, increasing the size of the minibatch decreases the standard deviation of the performance.

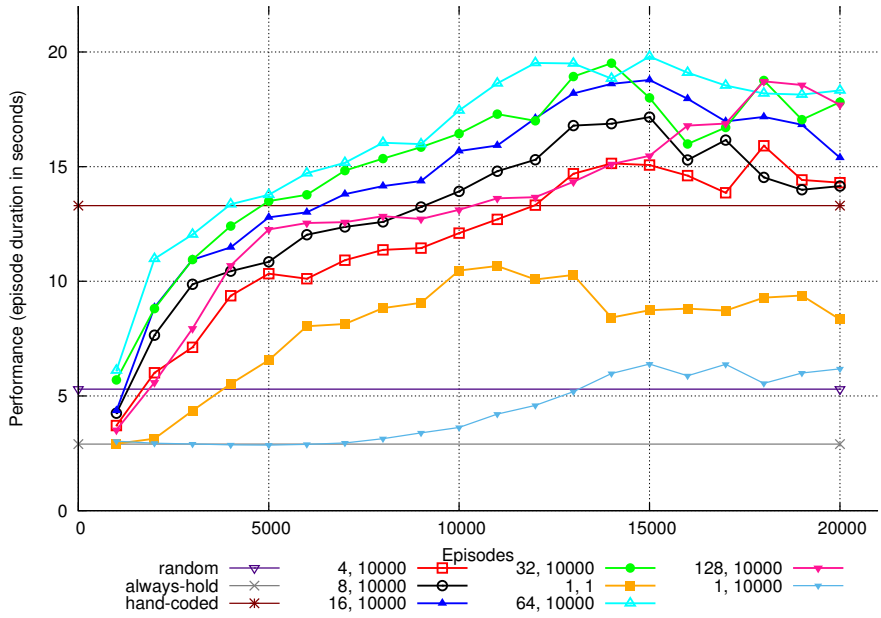

Figure 6.1: How minibatch size and experience replay influence the average episode duration.

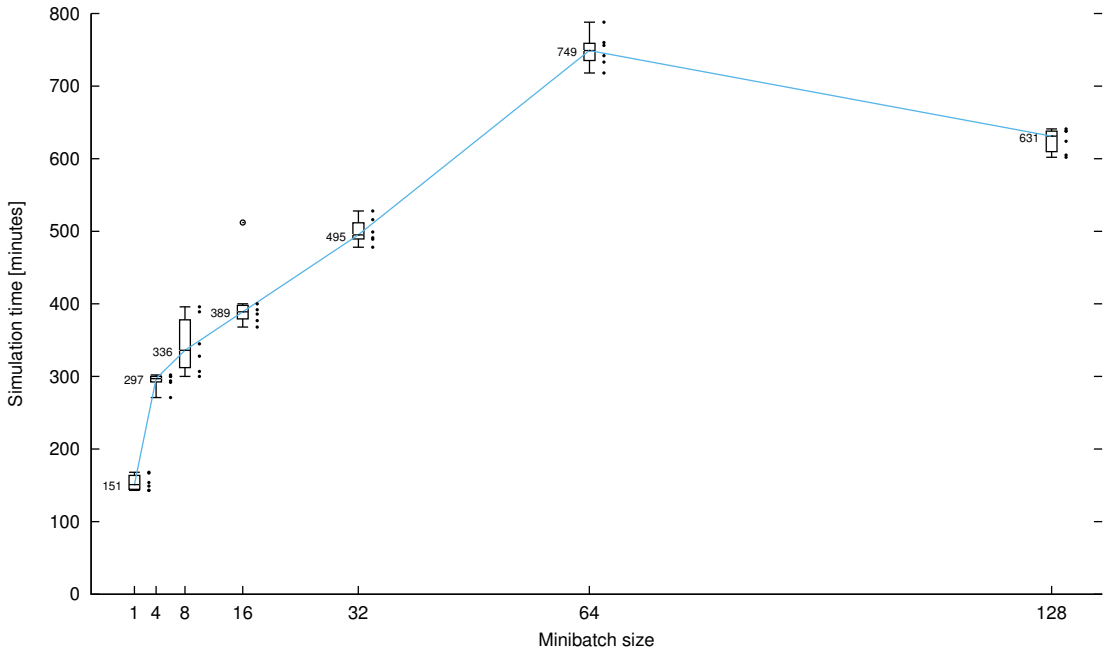

Figure 6.2: The impact of minibatch size to the learning time.

The learning dynamics is shown in Fig. 6.1. Due to time-consuming computation of each run, other sizes of experience memory were not tested.

Since the difference between results with 32 and 64 sizes of the minibatch is small and experiments with minibatch size of 64 take to run approximately 1.5 times longer than experiments with minibatch size of 32, in the subsequent experiments minibatch of size 32 will be used (minibatch of size 64 will be used to learn champion policy in Section 6.8).

6.4 Backpropagation

6.4.1 Experiment Setup and Objective

The objective of this experiment is to compare Google DeepMind's version of RMSProp algorithm with the classical RMSProp update rule.

6.4.2 Results and Discussion

Update rule	Agent performance [s]	Learning time [min]
classical RMSProp	17.8 (±1.3)	502
DeepMind RMSProp	17.9 (±2.1)	520

Table 6.4: DeepMind RMSProp vs. RMSProp.

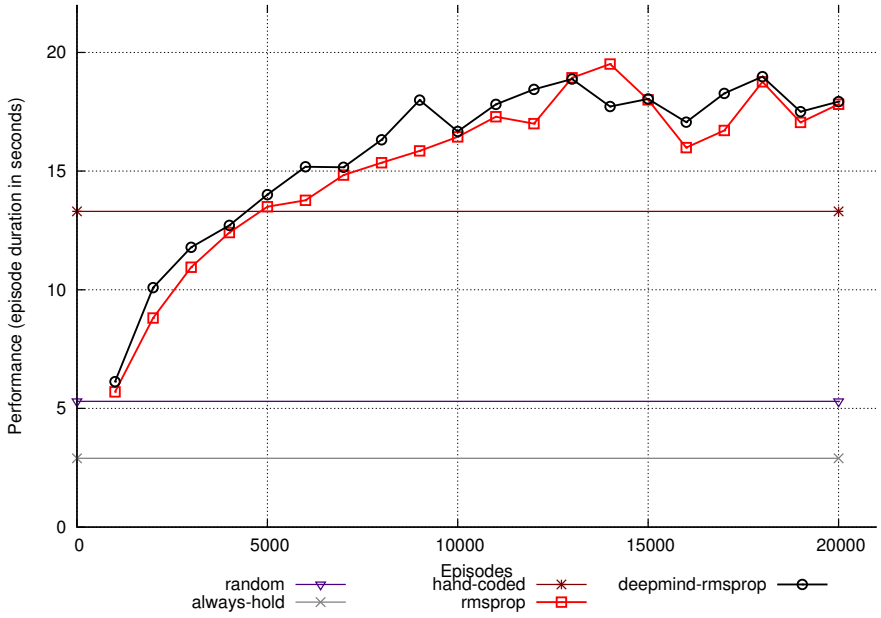

Figure 6.3: Average episode duration in seconds in DeepMind RMSProp vs. RMSProp experiment.

The comparison between classical RMSProp and DeepMind RMSProp is shown in Table 6.4 and Fig. 6.3. There is no significant difference between these two approaches on such small number of trials – to obtain notable and steady difference, dozens or hundreds of tests should be conducted. Note that the simulation time in learning runs with DeepMind RMSProp is slightly larger than

in runs with regular RMSProp algorithm. This is expected, since there is an additional factor to calculate. As depicted in Fig. 6.3, the DeepMind's version of the RMSProp algorithm is characterized by slightly less variation between consecutive evaluations, thus it will be used in subsequent experiments.

6.5 Target Network Freezing

6.5.1 Experiment Setup and Objective

In this experiment, different target network update frequencies were compared: every 100, 1000, and 10000 steps. Earlier research have shown (see Section 3.3.2) that freezing weights of the target Q-network leads to higher stability of the learning process and lower divergence between the consecutive results returned by the neural network. All experiments were performed with minibatch of size 32 and using DeepMind's version of the RMSProp algorithm.

6.5.2 Results and Discussion

Target network update frequency	Agent performance [s]	Learning time [min]
0	17.9 (±2.1)	520
100	14.5 (±2.6)	546
1000	16.9 (±3.7)	526
10000	16.1 (±1.4)	795 [1]

Table 6.5: Target network freezing results.

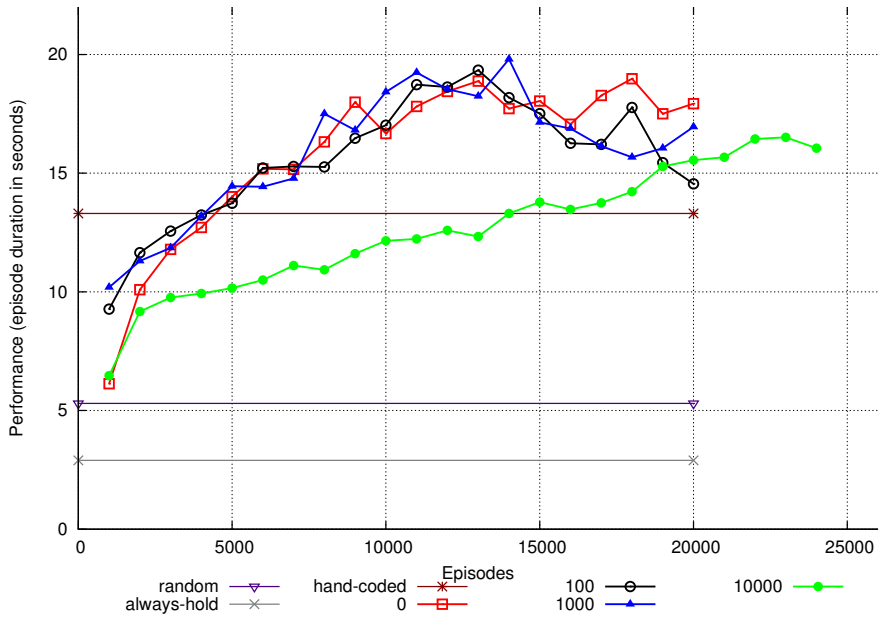

Figure 6.4: How target network update frequency influence on average episode duration.

[1] Simulation time of playing 30000 episodes.

The results of the experiment are shown in Table 6.5 and in Fig. 6.4. Contrary to the expectations, the results indicate that higher frequency does not provide any improvement over the baseline version, where only one neural network is maintained. The differences between different update schemes are insignificant. As shown in Fig. 6.4, the neural network with update frequency set to 10000 steps is learning significantly slower and with lower variance between consecutive samples. This experiment was extended to 30000 episodes, because after playing 25000 episodes, the agent's performance was not stable enough (no observable convergence) to draw conclusions.

Notice also, that the simulation time is slightly longer for the networks with update frequency 100 and 1000 steps than when using a single neural network – this is caused by relatively frequent copying values from Q-network to target network, which requires some computing power (for example, assuming 10 s as the average episode length during the whole learning process and that each Q-network update occurs every 100ms of simulator time, there are 25000, 2500, and 250 total target network updates for, respectively, 100, 1000 and 10000 update frequencies).

6.6 Neural Network Architectures

6.6.1 Experiment Setup and Objective

The goal of this experiment was to test, how different neural networks architectures perform in the keepaway soccer problem. During this experiment, neural networks with two up to five hidden layers with different numbers of nodes in the hidden layers were tested. Overall, six neural networks were considered in this experiment:

- 13-200-50-3,
- 13-30-100-30-3,
- 13-100-200-50-3,
- 13-200-100-50-3,
- 13-50-150-150-50-3, and
- 13-400-200-100-50-25-3.

All of these networks have rectifier activation function in hidden layers and no activation function in the output layer. The input layer of the neural network has always 13 nodes (the size of keepaway's state) and the output layer has always 3 nodes (the number of possible actions in 3 *vs.* 2 keepaway problem). All experiments were performed with minibatch of size 32, using DeepMind's version of the RMSProp algorithm and without the target network freezing.

6.6.2 Results and Discussion

Neural network architecture	Agent performance [s]	Learning time [min]
13-400-200-100-50-25-3	16.2 (±5.0)	673
13-50-150-150-50-3	15.8 (±0.5)	600
13-200-100-50-3	17.1 (±1.0)	582
13-100-200-50-3	16.9 (±2.0)	583
13-30-100-30-3	17.9 (±2.1)	520
13-200-50-3	18.1 (±2.2)	502

Table 6.6: Neural network architecture experiment results.

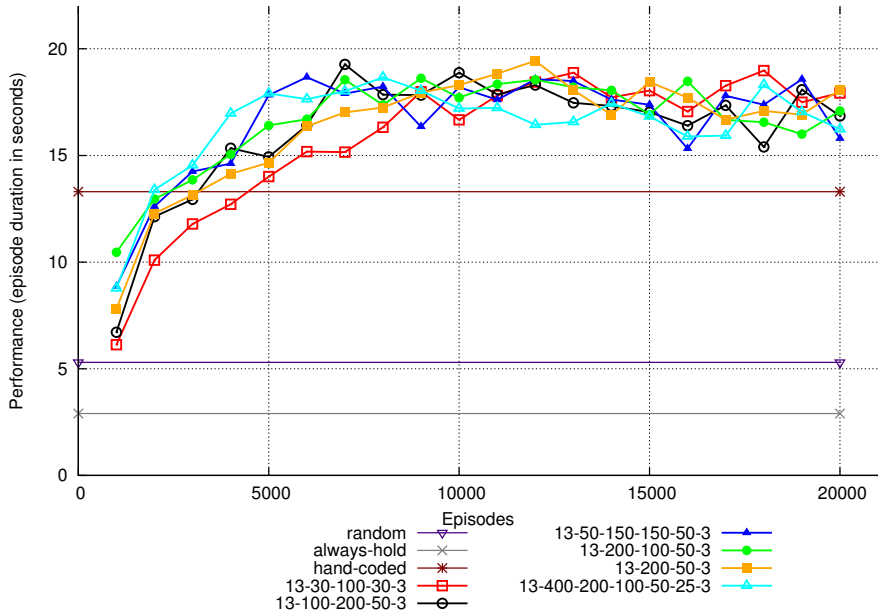

Figure 6.5: Agent performance during neural network architecture experiment.

The comparison between different neural networks architectures is shown in Table 6.6 and Fig. 6.5. The differences between various architectures of neural networks are relatively small. As presented in Fig. 6.5, in the initial phase of learning, deep neural networks have slightly better performance than shallow neural networks, but the overall trend shows that there is no advantage in using a deep neural network over a shallow neural network – neural networks with 4 and 5 hidden layers achieved the worst performance and the largest variance. The best result was made by the smallest neural network, with only two hidden layers. This might be caused by the small size of the input state. The results suggest, that there is no need to model complex non-linear relationships in a problem with such simple input as keepaway soccer or the learning algorithms used are not able to effectively make use of it.

The learning time is longer for larger neural networks, what is completely natural – there are more weights, thus more parameters need to be updated in a single step. Since the neural network of architecture 13-200-50-3 achieved the best result (both performance and learning time), it will be used in the subsequent experiments.

6.7 Meta-State Learning

6.7.1 Experiment Setup and Objective

The objective of the last experiment was to check whether combining multiple consecutive states into single *meta-state* leads to better agent's performance. A similar technique was effective for deep reinforcement in the video game playing domain [33].

To this aim, consecutive states are merged into a single vector of size $13n$, where n is the size of the meta-state. This should, theoretically, provide the agent more information about the actual state of the environment (recall that the environment is only partially observable by the agent). For example, the agent could reason whether takers are moving towards him.

During this experiment, meta-states of sizes 2 and 4 were tested. All trials were performed with minibatch of size 32, using DeepMind's version of the RMSProp algorithm, without freezing the target network and with the 13-200-50-3 neural network architecture.

6.7.2 Results and Discussion

Meta-state size	Agent performance [s]	Learning time [min]
1	18.1 (±2.2)	502
2	12.0 (±2.2)	425
4	9.4 (±2.1)	260

Table 6.7: How meta-state influence on average episode duration.

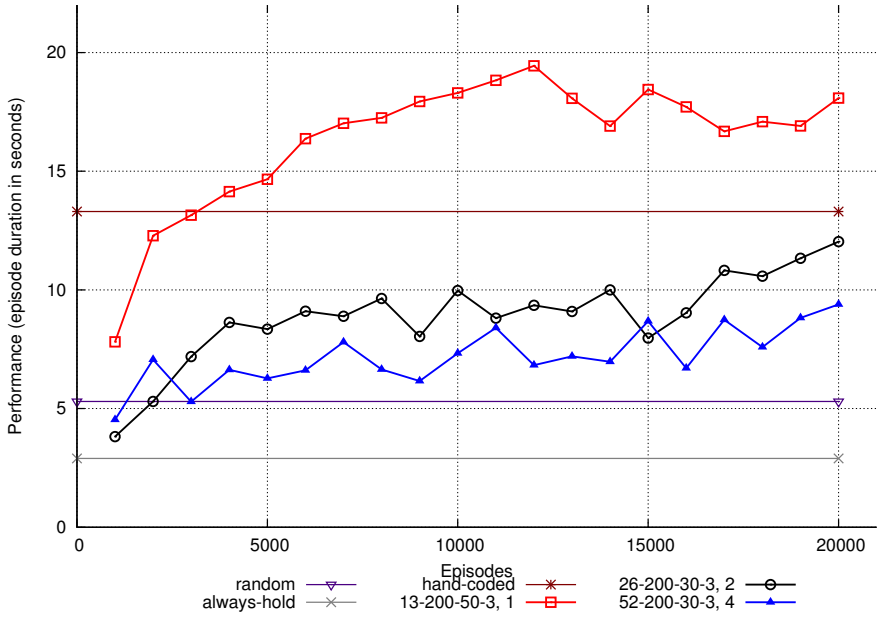

Figure 6.6: Average episode length in seconds during meta-state size experiment.

The results of the experiment are shown in Table 6.7 and in Fig. 6.6. Clearly, combining consecutive states into a single meta-state significantly decreases the agent's performance compared to using a single state as the input. This might be caused by the agent's inability either to extract valuable information from the meta-state or to relate consecutive values of variables. However, it is possible that using meta-state might leads to better agent's performance, but this will require significantly more learning time and could be the subject of future research.

6.8 Champion Agent

Based on the results of the experiments conducted in Sections 6.2-6.7, in this Chapter, the champion policy is explored and evaluated. It involves the best set of methods and parameters found in the previous experiments: the minibatch of size 64, the DeepMind RMSProp backpropagation algorithm, no target network freezing and no meta-state.

Comparing to the results of neural networks architectures experiments (see Section 6.6), there is a slight change in the last hidden layer – the champion agent uses 13-200-30-3 neural network architecture. Modification of neural network architecture is, again, result of trial-and-error A full list of parameters used by the champion agent is shown in Table 6.8. To better evaluate the champion policy 25 individual runs were performed.

Parameter	Value
NETWORK_ARCHITECTURE	13-200-30-3
UPDATE_RULE	*deepmind_rmsprop*
MINIBATCH_SIZE	64
TRANSITION_HISTORY_SIZE	10000
SWAP_NETWORKS_EVERY	0
RECENT_STATES_TO_NETWORK	1
FINAL_EPSILON_GREEDY	0.005
DISCOUNT_FACTOR	1.0
LEARNING_RATE	0.0001
LEARNING_RATE_CHANGE_EPISODES	15000
START_LEARN_AFTER	300
EXPLORATION_TIME	15000
STOP_AFTER_EPISODES	25000

Table 6.8: Champion agent parameters.

According to Fig. 6.7, the agent learns quickly and achieves the performance of over-17 s after just 6000 episodes of learning. After 8000 episodes the agent exceeded the value of 18 s of median episode duration and oscillates between values 18 and 20 to the end of the experiment. The final performance of the agent is 18.43 s (with the standard deviation equal to 1.3 s). The median of the method's results in the final evaluation is 18.2 s (with the best result for the single 100-episode evaluation run equal to 23.6 s).

The histogram of episode duration of the learned agents is shown in Fig. 6.8. Despite the fact that the agent's (median) performance is 18.2 s, almost 35% of it's results are below 10 seconds, which is caused by high randomness of the game.

6.8.1 Team Learning

6.8.1.1 Experiment Setup and Objective

During all the previous experiments, homogeneous team of the keepers was learned – they were using the same experience replay memory and were modifying weights of a single neural network. In this experiment, this approach will be compared to heterogeneous team learning [36], in which each agent has its own experience replay memory and its own neural network. Theoretically, the results should be the same as when using homogeneous team (at the beginning of each episode, keepers are randomly placed in each corner, so each agent has to learn how to start episode and how to play in other corners as well), and the theoretically optimal policy is the same for each agent. Nevertheless, when function approximation is used, the theoretical optimal policy is not achievable, thus the heterogeneous team might, in practice, improve the performance. The agents have no possibility of (direct) communication except observing behaviour of other keepers (thus, it is similar to real soccer game).

To perform the experiment, agent with the same set of parameters as the champion agent is used – the only difference is the heterogeneous idea of learning (turned on by set MULTI_AGENT to 1).

6.8.1.2 Results and Discussion

The results of comparison between homogeneous and heterogeneous team learning are shown in Table 6.9, in Fig. 6.9 and in Fig. 6.10. As expected, the final results of both approaches are close to each other, but the heterogeneous team learning is performing significantly better than

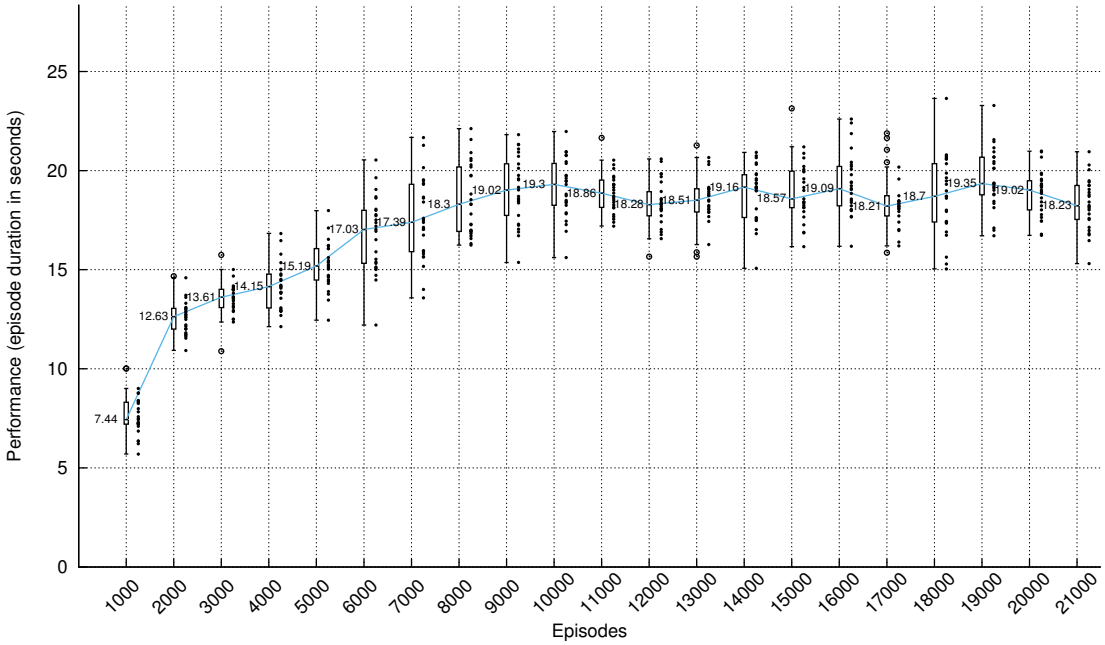

Figure 6.7: Median of agent's performance during champion agent's learning. Box plot reflects the quartile distribution and scatter plot the actual performance points from 25 runs.

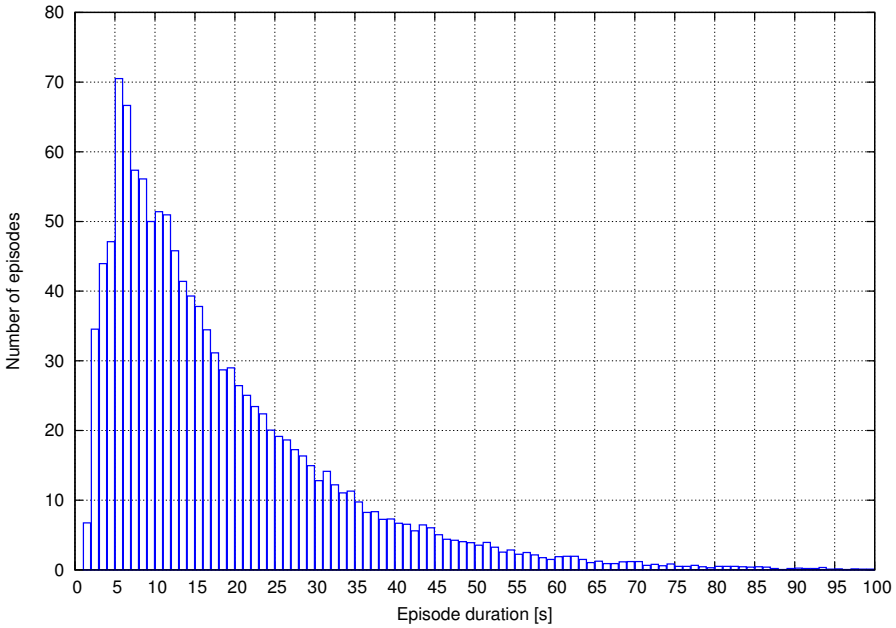

Figure 6.8: Histogram of episode duration during champion policy evaluation.

homogeneous team learning (t-test, $\alpha = 0.05$), achieving result of 19.6 s on average from 25 runs (the median of 19.5 s) with only 1.1 s of standard deviation. What is worth to notice is the learning speed (see Fig. 6.9). In heterogeneous setting the learning curve grows significantly slower than in homogeneous one. This is because each network's weights are updated 3 times less frequent than in homogeneous learning.

Multi-agent learning	Agent performance [s]	Learning time [min]
homogeneous	18.4 (±1.3)	860
heterogeneous	19.6 (±1.1)	813

Table 6.9: How heterogeneous learning influence on average episode duration.

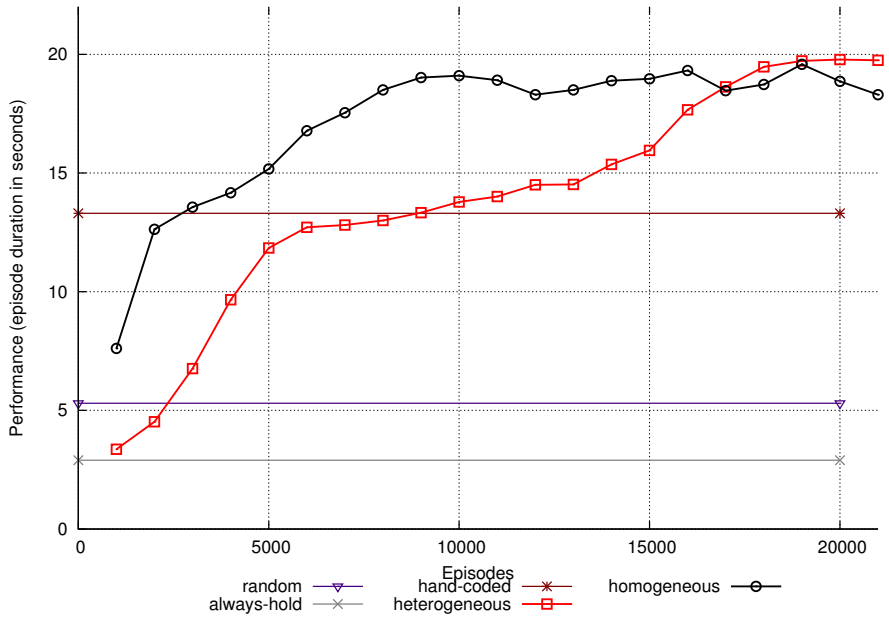

Figure 6.9: Average episode length in seconds during heterogeneous team learning experiment.

6.8.2 Comparison to Previous Studies

The comparison of all the previous results in the keepaway soccer domain together with the Deep Q-Learning approach is shown in Table 6.10. It indicates that the Deep Q-Learning method outperforms each of previously published results not only in terms of the agent's performance, but also in terms of standard deviation and the required simulated time.

Comparing Deep Q-Learning to the runner up, the best published to date method (SBB with diversity [19]), the presented approach is only slightly better in terms of the average game duration, but, importantly, needs 30 times less simulation hours to achieve this score. What is more, SBB methods is far more sophisticated than deep reinforcement learning and requires, unlike deep reinforcement learning, some domain knowledge (for example what genetic programming operators should be used).

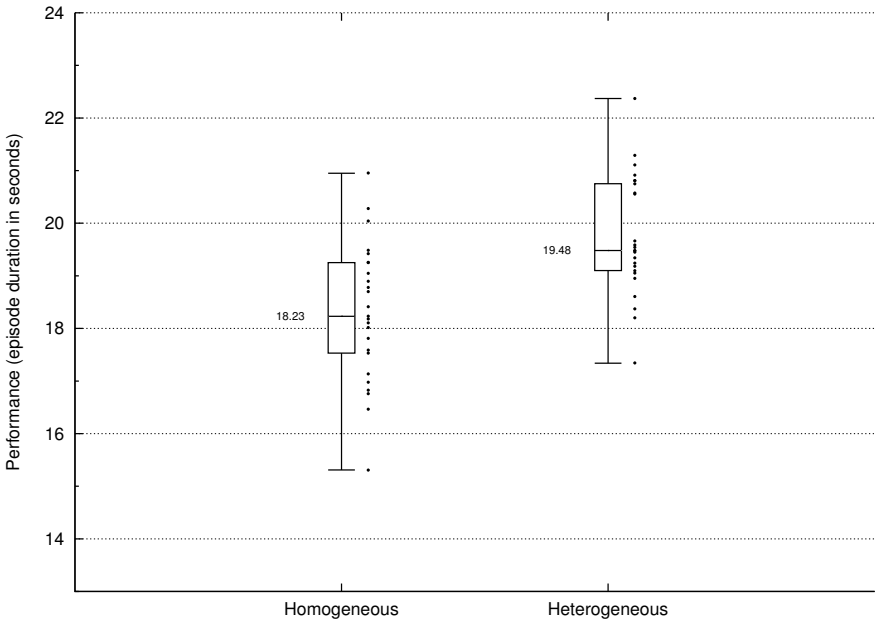

Figure 6.10: Median of homogeneous and heterogeneous agents performance during the final evaluation.

Algorithm	**Avg. keeper possession time [s] (± standard deviation)**	**Simulated hours**
Always Hold [50]	2.9 ± 1.0	-
Random [50]	5.3 ± 1.8	-
SARSA with Neural Network [51]	10.1 ± 0.3	30
Hand-coded [50]	13.3 ± 8.3	-
NEAT [60]	14.1 ± 1.8	800
SARSA with RBF [51]	14.2 ± 3.1	30
EANT [31]	14.9 ± 1.3	200
HyperNEAT [66]	15.4 ± 1.3	50-200
SARSA with CMAC [51]	15.7 ± 2.8	30
SBB with diversity [19]	18.5 ± 1.9	1739
(heterogeneous) Deep Q-Learning	19.6 ± 1.1	56.5

Table 6.10: Keepaway soccer task studies and results.

Chapter 7

Summary and Conclusions

Deep learning is a relatively new method in the artificial intelligence field, but it already produced numerous successful applications. One of them is very influencing work on deep reinforcement learning in general video game playing (GVGP) [33]. Keepaway soccer is a challenging machine learning task, with multiple agents and a large space state with uncertainty, in which the best to date agent has been produced using genetic programming. The main aim of this thesis was to verify if deep reinforcement learning is effective method for the most popular version of the keepaway soccer task (3 *vs* 2), which involves, unlike GVGP, multi-agent learning and low-dimensional states.

The particular attention was paid to Deep Q-Learning and a number of deep learning techniques presented by DeepMind in their recent paper „Playing Atari with Deep Reinforcement Learning" [33]. Specific studied methods include experience replay with minibatch learning, RMSProp back-propagation rule with DeepMind's modification of this algorithm, and target network freezing. Moreover, in contrast to the DeepMind's paper, certain learning parameters were optimized. This include the neural network architecture, learning rate, discount factor, exploration time, and ϵ-greedy parameters. Finally, two variants of team learning were examined: homogeneous team learning and heterogeneous team learning.

For the purpose of performing the experiments a general and efficient implementation of Deep Q-Learning agent was developed. It was designed to be flexible and easily deployable, so it might be adapted and used in further experiments in this field, especially concerning experience memory mechanism.

The results of the extensive experiments showed that some of the deep learning methods, indeed, increase performance of the agent, while some of them, actually, decrease it. Conclusions for each of tested method and parameter are as follows:

- Experience replay and minibatch learning significantly increase the performance of the agent. The best performance is achieved with minibatch of size 64.
- There is no clear difference between results of classical RMSProp algorithm and the version extended by DeepMind.
- There is no advantage in using a deep neural network rather than a shallow neural network for this problem, probably, because the state space is low-dimensional.
- Composing several subsequent states into a single meta-state (in order to alleviate the effect of noise) significantly decreases the performance of the agent.
- There is no clear difference between results of trials with frequent target network updating comparing to trial without target network freezing. When rare target network updating is used (every 10000 steps), the learning process is significantly slower than when target network is updated more often.
- Heterogeneous team learning achieves slightly (statistical significance) better performance

than homogeneous team learning.

Last but not least, the experimental results demonstrate that the Deep Q-Learning method applied for keepaway soccer problem performs better than any other previously published method, either in the case of average episode duration or concerning the computation effort to learn (simulation time). Comparing to the best to date result, achieved by genetic programming (SBB) agent [19], the Deep Q-Learning agent slightly (statistical significance) outperformed it using 1/30 computation time it utilized by SBB. Deep Q-Learning resulted also in significantly lower standard deviation than the SBB method.

The experiments confirmed that deep reinforcement learning is a suitable and effective method not only for highly-dimensional problems (that need to utilize convolutional neural networks (like GVGP)), but also for low-dimensional problems such as keepaway soccer.

7.1 Future Work

There are many aspects of the proposed approach that are worth further investigation. A few possible directions are as follows:

- As presented in Section 6.5, updating target network after 10000 steps looks promising and its result might be improved with a tuning of other parameters such as the number of learning episodes.
- Using another form of state representation, such as matrix instead of vector, could allow to extract more information from consecutive states, as discussed in Section 6.7.
- Experience replay, which was studied in this thesis, is limited in some respects – the memory does not differentiate the important transitions from the unimportant ones. A more sophisticated sampling strategy might be considered, which would put emphasis to transitions from which the agent could learn the most, similarly as prioritized sweeping works [35].
- As discussed in Section 6.2, preliminary studies were not rigorous experiments, so another potential parameters, such as learning rate and exploration time, could be put under structured experiments. Moreover, additional not tested parameters, such as the type of activation function should be studied in future research.
- The presented agent learns how to play on $20\,\mathrm{m} \times 20\,\mathrm{m}$ region and is evaluated in the same environment. Further research can investigate how well does the agent learned to play in region $20\,\mathrm{m} \times 20\,\mathrm{m}$ generalize to other region sizes [19].
- In some of the plots shown in Section 6, there is visible instability or cycling dynamics of the learning process. The agent performance sometimes decreases after playing 12000-16000 episodes, which coincidence with the of the exploration phase. This effect requires further research.

Appendix A

DVD Content

The DVD attachted to this thesis contains:

- The Deep Q-Learning agent source code described in Chapter 5,
- Raw results of the experiments described in Chapter 6,
- The PDF version of this thesis.

Appendix B

Champion Agent Parameters File

Listing B.1: Champion policy parameters (docker environment variables)

```
HOME=/home/soccer
LOGGER_LEVEL=INFO
THEANO_FLAGS=mode=FAST_RUN

EVALUATE_AGENT_EACH=600
EVALUATION_EPISODES=100
CLIP_DELTA=0
ERROR_FUNC=sum
DISCOUNT_FACTOR=1.0
EXPLORATION_TIME=15000
FINAL_EPSILON_GREEDY=0.005
START_LEARNING_RATE=0.0001
FINAL_LEARNING_RATE=0.0001
LEARNING_RATE_CHANGE_EPISODES=15000
MINIBATCH_SIZE=64
NETWORK_ARCHITECTURE=13,200,30,3
START_LEARN_AFTER=300
STOP_AFTER_EPISODES=25000
SWAP_NETWORKS_EVERY=0
TRANSITIONS_HISTORY_SIZE=10000
UPDATE_RULE=deepmind_rmsprop
RECENT_STATES_TO_NETWORK=1
MULTI_AGENT=1
FINAL_EVALUATION=1000
```

Bibliography

[1] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, Université De Montréal, and Montréal Québec. Greedy layer-wise training of deep networks. In *In NIPS*. MIT Press, 2007.

[2] Darrin C Bentivegna, Aleš Ude, Christopher G Atkeson, and Gordon Cheng. Humanoid robot learning and game playing using pc-based vision. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2449–2454. IEEE, 2002.

[3] UC Berkeley. New 'deep learning' technique enables robot mastery of skills via trial and error. http://news.berkeley.edu/2015/05/21/deep-learning-robot-masters-skills-via-trial-and-error/, 2015. [Online; accessed 12-September-2015].

[4] Mao Chen, Klaus Dorer, Ehsan Foroughi, Fredrik Heintz, Zhanxiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Jan Murray, Itsuki Noda, Oliver Obst, Pat Riley, Timo Steffens, Yi Wang, and Xiang Yin. RoboCup Soccer Server 7.07, Manual. page 150, 2003.

[5] Robert Crites and Andrew Barto. Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8*, pages 1017–1023. MIT Press, 1996.

[6] Robert H. Crites, Andrew G. Barto, Michael Huhns, and Gerhard Weiss. Elevator group control using multiple reinforcement learning agents. In *Machine Learning*, pages 235–262, 1998.

[7] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Mike Seltzer, Geoffrey Zweig, Xiaodong He, Julia Williams, et al. Recent advances in deep learning for speech research at microsoft. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8604–8608. IEEE, 2013.

[8] Li Deng and Dong Yu. Deep learning: methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4):197–387, 2014.

[9] Anthony Di Pietro, Lyndon While, and Luigi Barone. Learning in RoboCup Keepaway using Evolutionary Algorithms. 2002.

[10] Steve Dini and Mark Serrano. Combining q-learning with artificial neural networks in an adaptive light seeking robot. 2012.

[11] Kevin Swersky Geoffrey Hinton, Nitish Srivastava. rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning. http://www.cs.toronto.edu/t̃ijmen/csc321/slides/lecture_slides_lec6.pdf, 2012.

[12] Steven M. Gustafson and William H. Hsu. Layered learning in genetic programming for a cooperative robot soccer problem. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 291–301, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[13] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[14] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[15] William H. Hsu, Scott J. Harmon, Edwin Rodriguez, and Christopher Zhong. Empirical comparison of incremental reuse strategies in genetic programming for keep-away soccer. In Maarten Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 26 July 2004.

[16] Bing-Qiang Huang Bing-Qiang Huang, Guang-Yi Cao Guang-Yi Cao, and Min Guo Min Guo. Reinforcement Learning Neural Network to the Problem of Autonomous Mobile Robot Obstacle Avoidance. *2005 International Conference on Machine Learning and Cybernetics*, 1(August):18–21, 2005.

[17] Wojciech Jaśkowski, Marcin Szubert, and Paweł Liskowski. Multi-criteria comparison of coevolution and temporal difference learning on othello. In A. I. Esparcia-Alcazar and A. M. Mora, editors, *EvoApplications 2014*, volume 8602 of *Lecture Notes in Computer Science*, pages 301–312. Springer, 2014.

[18] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[19] Stephen Kelly and Malcolm I Heywood. Genotypic versus Behavioural Diversity for Teams of Programs Under the 4-v-3 Keepaway Soccer Task. pages 3110–3111, 2014.

[20] Hayato Kobayashi, Tsugutoyo Osaki, Eric Williams, Akira Ishino, and Ayumi Shinohara. Autonomous learning of ball trapping in the four-legged robot league. In *RoboCup 2006: Robot Soccer World Cup X*, pages 86–97. Springer, 2007.

[21] Jens Kober and Jan Peters. Reinforcement learning in robotics: A survey. In *Reinforcement Learning*, pages 579–610. Springer, 2012.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[23] Duong Quoc Thang Le, Shriman Narayan Tiwari, and Bernard Merialdo. Deep learning image recognition. 2015.

[24] Peter Lichodzijewski and Malcolm I. Heywood. Managing team-based problem solving with symbiotic bid-based genetic programming. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 363–370, New York, NY, USA, 2008. ACM.

[25] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

[26] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

[27] S. Mahadevan, J. Connell, C. Sammut, R. Sutton, and Temporal Phd. Automatic programming of behavior-based robots using reinforcement learning, 1991.

[28] Sridhar Mahadevan and Georgios Theocharous. Optimizing production manufacturing using reinforcement learning. In *In Eleventh International FLAIRS Conference*, pages 372–377. AAAI Press, 1998.

[29] Fredrik Heintz ZhanXiang Huang Spiros Kapetanakis Kostas Kostiadis Johan Kummeneje Itsuki Noda Oliver Obst Pat Riley Timo Steffens Yi Wang Xiang Yin Mao Chen, Ehsan Foroughi. The robocup soccer simulator. http://sourceforge.net/projects/sserver/, 1997-2015. [Online; accessed 12-September-2015].

[30] Maja J. Mataric. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4:73–83, 1997.

[31] Jan H Metzen, Mark Edgington, Yohannes Kassahun, and Frank Kirchner. Analysis of an evolutionary reinforcement learning method in a multiagent domain. *{AAMAS} '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, (Aamas):291–298, 2008.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei a Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[34] John Moody and Matthew Saffell. Learning to trade via direct reinforcement. *Neural Networks, IEEE Transactions on*, 12(4):875–889, 2001.

[35] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, pages 103–130, 1993.

[36] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

[37] Tom Palmer Peter Stone, Gregory Kuhlmann. Keepaway soccer benchmark player framework. https://github.com/tjpalmer/keepaway, 2006-2015. [Online; accessed 12-September-2015].

[38] Christopher Poultney, Sumit Chopra, and Yann Lecun. Efficient learning of sparse representations with an energy-based model. In *Advances in Neural Information Processing Systems (NIPS 2006*. MIT Press, 2006.

[39] Scott Proper and Prasad Tadepalli. Scaling model-based average-reward reinforcement learning for product delivery. In *Machine Learning: ECML 2006*, pages 735–742. Springer, 2006.

[40] Andrew G. Barto Richard S. Sutton. Reinforcement learning i: Introduction, 1998.

[41] Martin Riedmiller. Rprop-description and implementation details technical report, january 1994.

[42] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

[43] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[44] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5:3.

[45] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

[46] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959.

[47] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[48] Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank, 2013.

[49] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, June 2002.

[50] Peter Stone. Learning to Play Keepaway. http://www.cs.utexas.edu/users/AustinVilla/sim/keepaway/#benchmarks, 2006. [Online; accessed 12-September-2015].

[51] Peter Stone, Gregory Kuhlmann, Matthew E Taylor, and Yaxin Liu. Keepaway Soccer: From Machine Learning Testbed to Benchmark. *Lncs*, 4020:93–105, 2006.

[52] Peter Stone and Richard S Sutton. Keepaway Soccer: A Machine Learning Testbed. *Lecture Notes in Computer Science*, 2377:207–237, 2002.

[53] Peter Stone, Richard S Sutton, and Gregory Kuhlmann. Reinforcement Learning for RoboCup-Soccer Keepaway. *Adaptive Behavior*, 13:165–188, 2005.

[54] Peter Stone and Rs Sutton. Scaling reinforcement learning toward RoboCup soccer. *Icml*, (June):537–544, 2001.

[55] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3(1):9–44, August 1988.

[56] Richard S. Sutton and Andrew G. Barto. Towards a modern theory of adaptive networks: expectation and prediction. *Psychol. Rev*, pages 135–170, 1981.

[57] Richard Stuart Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, 1984. AAI8410337.

[58] Marcin Szubert. Coevolutionary reinforcement learning and its application to othello. Master's thesis, Poznan University of Technology, 2009.

[59] Marcin Szubert and Wojciech Jaskowski. Temporal difference learning of n-tuple networks for the game 2048. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[60] Matthew E Taylor, Shimon Whiteson, and Peter Stone. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. *Proceedings of the 8th annual conference on Genetic and evolutionary computation GECCO 06*, (July):1321, 2006.

[61] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[62] Sebastian Thrun. Learning to play the game of chess. In *Advances in Neural Information Processing Systems 7*, pages 1069–1076. The MIT Press, 1995.

[63] C Touzet. Neural networks and q-learning for robotics. In *Proceedings of the international joint conference on neural networks (IJCNN'99)*, 1999.

[64] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Technical report, IEEE Transactions on Automatic Control, 1997.

[65] Manuela M. Veloso, Manuela M. Veloso, Peter Stone, Peter Stone, Michael Bowling, and Michael Bowling. Anticipation: A Key for Collaboration in a Team of Agents. *Third International Conference on Autonomous Agents (Agents'99)*, pages 134–141, 1999.

[66] Phillip Verbancsics and Kenneth O Stanley. Evolving Static Representations for Task Transfer. *Journal of Machine Learning Research*, 11:1737–1769, 2010.

[67] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[68] Shimon Whiteson, Matthew E. Taylor, and Peter Stone. Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 21:1–35, 2010.