Indoor Visual Navigation using Deep Reinforcement Learning

Søren Skov

AALBORG UNIVERSITY

STUDENT REPORT

Aalborg University
Mathematical Engineering

Aalborg University, June 7, 2018

Søren Skov
<ssko12@student.aau.dk>

Mathematical Engineering
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Indoor Visual Navigation using Deep Reinforcement Learning

Project Period:
September 2017 - June 2018

Project Group:
G3-115

Participants:
Søren Skov

Supervisors:
Zheng-Hua Tan
Morten Kolbæk

Copies: 1

Page Numbers: 85

Date of Completion:
June 7, 2018

Abstract:

The focus of this project is to train an agent to improve its behaviour of navigating in an indoor environment using visual input. This is done through the use of deep reinforcement learning trained on images to find a number of target positions. The work in this project is based on [Zhu et al. 2016] where an agent is trained on 100 million images to find 100 targets. The performance of this algorithm shows that there are room for improvements. Therefore, in this project an analysis of what such as agent learns during the training is carried out to get an understanding of what the agent learns and how this might effect the performance. A way to visualize what the agent learns during the training is proposed to help this analysis.

Preface

This project is written by the student Søren Skov at Aalborg University under the supervision of Professor Zheng-Hua Tan at Aalborg University and Ph.d Morten Kolbæk. Both supervisors are from the department of Electronic Systems. It is written as part of the fulfilment of his Master of Science degree in Mathematical Engineering. References will be used throughout, and can be found in the bibliography at the end. Specific page numbers or sections may be mentioned. The files are available for replication of results, and are found at AAU project library, `projekter.aau.dk`. The models have been implemented in Python within both Keras framework and Tensorflow.

Resumé

Dette projekt omhandler anvendelsen af deep reinforcement learning til at lære en agent at navigere rundt i indendørs miljøer for at finde en bestemt lokation. I dette projekt beskrives den grundlæggende teori bag reinforcement learning, og hvordan dette kan kombineres med teori fra deep learning, som gør det muligt at anvende reinforcement learning til at løse realistiske problemer i virkerlige miljøer.

Reinforcement learning går ud på at lære en agent at optimere dens opførsel i et givent miljø mens den selv interagerer. Specifik tages der udgangspunkt i at lære en agent at bevæge sig rundt i indendørs rum i huse, hvor den har til opgave at finde feom til bestemte lokationer ved brug af billeder. Da træning af en sådan algoritme kræver meget data og at disse data ofte er tidskrævende at opsamle, benyttes simuleret data fra 3D modeller af indendørs rum.

Dette projekt tager udgangspunkt i artiklen [Zhu et al. 2016], hvor en agent er trænet til at finde den korteste vej til 100 forskellige lokationer i 20 forskellige rum. Denne agent er trænet på 100 millioner billeder og den gennemsnitlige længde det tager agenten at nå frem til de bestemt destinationer er omkring 100 meter i et indendørs miljø. Dette er en lang vej agenten skal tilbagelægge i et almindelig rum i et hus.

Artiklen benytter en populær algoritme i deep reinforcement learning, som hedder A3C. Denne har vist gode resultater i andre domæner som video spil og er den, der tages udgangspunkt i. Model arkitekturen for algorithmen i [Zhu et al. 2016] analyseres ved at kigge på hvad algoritmen lærer undervejs for at undersøge om dette har indflydelse på konvergens og eventuel of performance.

I dette projekt undersøges det dermed, hvordan det kan være, at længden fra en tilfældig start lokation til lokationen, som agenten skal finde, er så lang. Dette gøres ved at visualisere, hvad agent lærer undervejs og en analyse af dette bruges til at give nogle forklaringer på, hvorfor agenten opfører sig som den gør. For at simplificere analyseprocessen bliver en agent som er trænet på et enkelt rum analyseret.

Contents

List of Figures

Chapter 1
Introduction

Artificial intelligence is a branch of science concerned with building intelligence into computer systems or machines [Nilsson 2010]. Intelligence can be grouped into different categories such as solving math problems, being able to learn from from mistakes, playing challenging games such as check etc. [Nilsson 2010]. These are tasks humans have to solve on a daily basis, and the human brain is a very complex organism that allows us to do exactly that [Haykin 2009].

Machine learning is one particular important area in building intelligent systems. If a machine is asked to solve a difficult task, it can be challenging to address the problem by designing a fixed program to complete the task [Goodfellow, Bengio, and Courville 2016]. Instead one could address the problem by using machine learning. In general there are three different ways to approach a machine learning problem, which includes supervised -, unsupervised -, and reinforcement learning. In reinforcement learning the learning agent, often just called the agent, directly interact with the environment in which it has to optimize its behaviour. Therefore, in this approach the data set might not be available prior to the training, but is instead collected over time.

Accessibility of large amount data and the increasing computational power to process these huge amount of data has made machine learning algorithms, and specially deep learning, more attractive. In the last decade artificial intelligence, and specially machine learning, has received a lot of attention in fields such as image recognition, image caption, gaming and robotics. Advances in deep learning has given computers the ability to classify images with a high precision accuracy [Lecun, Bengio, and Hinton 2015]. Techniques in deep learning has even made it possible for intelligent systems to capture and describe the actions in an image [Lecun, Bengio, and Hinton 2015].

In the field of deep reinforcement learning, which is a combinations of techniques in both deep learning and reinforcement learning, computers are now able to beat humans in games like GO and different Atari games. In [Mnih et al. 2015] a deep reinforcement learning algorithm is trained to play old Atari games and is able to beat human players in many of the games. In [Silver et al. 2016] reinforcement learning is used to train a computer to play the game of GO. GO is considered the most complex board game in the world due to its enormous number of different board configurations, and for first time a computer has been able

to beat a human professional in the game [Silver et al. 2016].

An interesting application of artificial intelligence is in the field of robotics. One of the goals of artificial intelligence is the development of intelligent agents that can act autonomously without the need of human supervision [Arulkumaran et al. 2017]. This has many different applications as robots are used to take over tasks normally performed by humans. Therefore, robots should also be able to navigate in the real-world environments where these tasks are performed. Combining deep learning with reinforcement learning is a necessary step towards making agents that are capable of solving real world tasks [Mnih et al. 2015].

One such area of real world tasks is in the field of robotics, and specially social robots, where robots are interacting with humans. For such robots to be successful, they must be able to solve different social tasks such as face recognition, speech recognition and indoor navigation. In the development of autonomous robots, one of the challenges is to learn the robot to navigate in the same environment as humans. In [Zhu et al. 2016] a robot is trained to navigate to target positions in indoor scenes using visual data in terms of images. The algorithm is trained on 100 million images from a simulation framework to find 100 different targets in 20 scenes. The performance is measured as the average trajectory length from 10 trials for each of the 100 targets. On average the agent needs to perform 210.t actions to find the targets and with a step size of 0.5 meters, this corresponds to about 100 meters inside an indoor environment.

Learning from interactions is the general idea behind reinforcement learning [Sutton and Barto 1998]. By learning, one means that the agent should be able to not only use precepts of the environment to act, but also utilize past experience to improve its behaviour in the future, which for example could be finding the shortest path length that bring the the learning agent from its current position to a target position.

One of the disadvantages of applying reinforcement learning to robotics in real-world environments is the amount of data required to train the model. Real-world samples are often time consuming and tedious to acquire [Kolve et al. 2017]. Therefore, a simulation framework is provided in [Kolve et al. 2017] to acquire training data to train these models.

This report is focused around the work in [Zhu et al. 2016]. The goal of the project is therefore to understand the fundamental idea behind reinforcement learning and why these methods work to solve a navigation problem of navigating the agent to some predefined target positions in an indoor environment. Methods from deep learning are an essential part in applying reinforcement learning to real world applications. This combination of deep learning and reinforcement is known as deep reinforcement learning and therefore, to describe this approach a brief knowledge of theory and ideas in deep learning and reinforcement learning is required.

1.1 Problem Statement

The average performance of the algorithm in [Zhu et al. 2016] is that the targets are found by moving about 100 meters inside an indoor environment. For a single room in a traditional house this is a long travel. In this project answers to why the performance is better than are sought.

How can visual data in terms of images be used to control a robot to find a predetermined location in the scenes of an indoor environment?

- How can a visualization of what the agent learns be conducted and used to analyse how the performance can be improved?
- How can the performance be explained using these visualizations?
- How can the amount of training samples be reduced for robot navigation using visual input?

In this project an analysis of what the agent learns is conducted to see how the performance of the algorithm in the original paper [Zhu et al. 2016] can be improved.

Chapter 2
Reinforcement Learning

In reinforcement learning one consider the interaction between an agent, also called the decision maker, and an environment. This interaction is illustrated in Figure 2.1. The agent senses the environment through sensors, and is able to respond to the environment through actions performed by what is called actuators [Russell and Norvig 2010]. The sensors and actuators depends on the problem in hand, but examples of these are cameras that make the agent sense the environment, and motors that moves the agent around in the environment, respectively. The fundamental idea behind reinforcement learning is to learn the agent to achieve a goal by letting it interact with the environment [Sutton and Barto 1998, p. 51]. This could be described as a sequential decision making problem, where the agent performs actions in sequence to achieve a goal. The goal is what the agent should learn to do in its environment [Sutton and Barto 1998, p. 56]. This problem of learning an agent to behave in an environment to achieve a goal is from now on referred to as the reinforcement learning problem.

Consider the reinforcement learning problem just described. To have an agent achieve a wanted goal can be formalized by defining an appropriate reward function and letting the goal of the agent be to maximize the reward it gets over time [Sutton and Barto 1998, p. 56]. If the environment is complex it could be tedious to code what the agent should do in every possible action. Instead of telling what exactly the agent should do in any situation, the agent must self discover how to reach the goal by selecting the best actions in any given situation.

The agent receives information about the environment, called states, at discrete or continuous time steps, but from now on only discrete time steps will be considered. Upon receiving a state of the environment, the agent responds by picking an action from a decision rule [Puterman 1994]. An immediate or delayed feedback from the environment is then given to the agent that tells how good that action was for the agent in that state [Sutton and Barto 1998].

The environment in which the agent has to operate can have some different properties, depending on the design of environment and agent. One thing to consider is the amount of information about the environment the agent has access to. One of the first things to consider is whether the environment is discrete or continuous. If the environment is discrete, it means that there are limited number of percepts and actions of the environment [Russell and Norvig 2010].

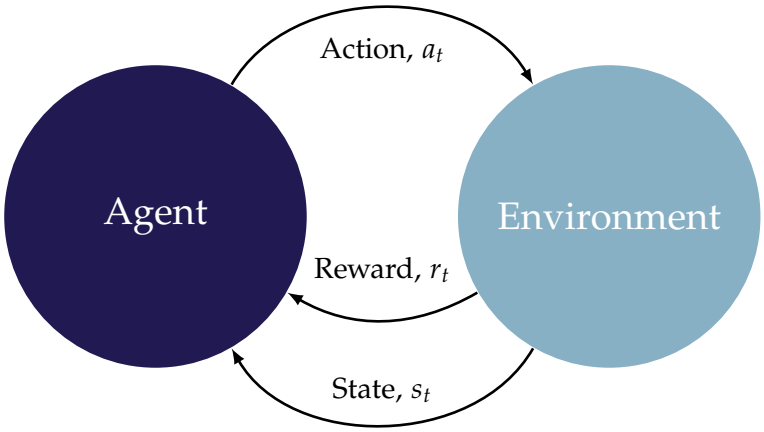


Figure 2.1: Interaction between an agent and the environment surrounding the agent. Recreation of the figure in [Sutton and Barto 1998, p. 52].

Additionally, the environment can be both deterministic or stochastic [Russell and Norvig 2010]. If the environment is deterministic it means that the next state of the environment is completely determined given the previous state and the action picked by the agent. However, if the agent has to deal with uncertainty of the environment, it is considered stochastic [Russell and Norvig 2010]. In some cases while the agent experiences the environment, it might change over time. For example, objects or people can move around changing the environment. If this is the case, the environment is considered dynamic. Otherwise, it called a static environment.

The experience the agent gets from interacting with the environment can be both episodic or nonepisodic [Sutton and Barto 1998]. Experience is considered episodic if it can be divided into episodes, where a completion of a task ends an episode. [Russell and Norvig 2010]

Reinforcement learning differs from supervised learning where the supervisor is replaced with a reward signal [Sutton and Barto 1998]. This means that there are no labelled data set available for training, but only a reward signal that defines the goal of what the agent should try to achieve. This is due to the fact that actions might not only affect the immediate reward, but also the reward of actions in the future [Sutton and Barto 1998, p. 4]. In addition to this, the agent is also able to influence the environment, and therefore also the data given to en algorithm, through the actions the agent decides to take. Acquiring data of the desired behaviour of the interaction between agent and environment, that allows the use of a supervised learning approach, is often impractical as it does not cover all the interaction situations [Sutton and Barto 1998, p. 4].

In robotics, making a robot navigate in an indoor environment can be considered as a reinforcement learning problem as in [Zhu et al. 2016]. The goal of the agent could be to find a specific target position in a room and move to the target position from the agents current position. For a robot with a number of specific actions, such as moving forward and turning left, the robot should perform

a sequence of actions that brings it from its current position to the target. In this reinforcement learning problem, the order in which the actions are performed is important.

In much of the literature of reinforcement learning, the interaction model between the agent and the environment is modelled as a Markov decision process [Sutton and Barto 1998], which is described next.

2.1 Markov Decision Processes

A Markov decision process is a mathematical framework for sequential decision-making, which consists of five components being a state space, $\mathcal{S}$, an action space, $\mathcal{A}$, transition probabilities $p(s'|s,a)$, rewards $r(s,a)$, and a discount factor, γ [Littman 2015]. These five components together constitute a Markov decision process, formally given by the set

$$\Big\{\mathcal{S}, \mathcal{A}, p(s'|s,a), r(s,a), \gamma\Big\}. \tag{2.1}$$

This section is limited to discrete time Markov decision processes, where the interaction between agent and environment happens at discrete time steps, $t = 1, 2, 3, \ldots$.

The state space, $\mathcal{S}$, is the set of available states of the environment. At each discrete time step, t, the agent receives a state, $s_t \in \mathcal{S}$, which contains the information about the environment that is available to the agent at time t. The second component is the action space, $\mathcal{A}$, which is the set of actions the agent can perform in the environment. From a given state, s, there might be some actions that the agent cannot perform, which limits the action space in that state, denoted as $\mathcal{A}(s)$. If both $\mathcal{A}$ and $\mathcal{S}$ are finite sets, the process is called a discrete-time finite Markov decision process. [Sutton and Barto 1998].

The third component in a Markov decision process is the reward function that gives the agent feedback from the environment [Sutton and Barto 1998, p. 7]. The feedback is given in terms of a scalar reward $R_t \in \mathbb{R}$ after the agent has performed an action in the occupied state. The reward function specifies the goal of the reinforcement learning problem [Sutton and Barto 1998, p. 7]. For example if a robot has to find the shortest path to a location in a room, the reward function can be constructed in such a way that the agent receives a penalty in terms of a negative reward for every step it takes without reaching the object. In this way the agent will try to reach the goal by finding the shortest path to the location. A high positive reward is then given for the actions that brings the agent to the state of the location the agent is asked to find.

The environments respond to an action taking at time step $t+1$ would in the most general case depend on everything that has happened up to this time step [Sutton and Barto 2017]. Mathematically, this can be expressed using conditional probability as

$$P\left(R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\right), \tag{2.2}$$

for all r, s' and values of past events $S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$, if it is assumed that there are a finite number of states and reward values [Sutton and Barto 2017]. The dynamics can therefore only be specified by knowing the probability distribution of Equation 2.2. Instead one can model the environment to satisfy the Markov property. A first order Markov property states that the transition from s_t to s_{t+1} only depends on the most recent state, s_t and not the history of all the state before s_t [Haykin 2009, p. 657]. Hence,

$$P\left(S_{t+1} = s', \mid S_0, A_0, S_1, A_1, \ldots, S_t, A_t\right) = P\left(S_{t+1} = s' \mid S_t = s, A_t = a\right) \tag{2.3}$$

The transition probabilities refers to probabilities of changing from state s to a successor state s' given that action a is performed by the agent, and is given in Equation 2.4.

$$p(s'|s,a)P\left(S_{t+1} = s' \mid S_t = s, A_t = a\right) \tag{2.4}$$

The transition dynamics, that now satisfy the Markov property, is often summarized in a transition graph for small state and action spaces. An example of a part of a such a transition graph is seen in Figure 2.4.

The last component of the Markov decision process is the discount factor, $\gamma \in [0, 1]$. This is used to measure the accumulated rewards, called return, to determine how much future rewards should influence the return [Sutton and Barto 1998, p. 58]. The closer γ is to 1 the more affect future rewards have on the return.

Definition 2.1 (Return)
Let R_t be the immediate reward at time step t, and $\gamma \in [0, 1]$ be the discount factor. The return, G_t, is then defined as in [Sutton and Barto 1998, p. 58]

$$G_t = \sum_{k=0}^{T} \gamma^t R_{t+k+1} \tag{2.5}$$

where T is the last time step in the interaction between agent and environment.

For a reinforcement learning problem that breaks into episodes, for example in a single game of checkers where a terminal state is reached when the game is finish, G_t is a finite sum of rewards. In this case G_t will itself be finite, if the immediate rewards are finite [Sutton and Barto 1998, p. 58]. However, for a reinforcement learning problem that does not naturally break into these subsequences with a finite episode length a discount factor $\gamma < 1$ prevents the return from being infinite [Arulkumaran et al. 2017]. Therefore, note that $T = \infty$ and $\gamma = 1$ must not both be satisfied.

In the Markov decision process framework, the goal is to find a policy, denoted π, that maximizes the return over time. The policy is the agents way of behaving, which for any given state determines the next action the agent should take. In general such a policy can be both deterministic or stochastic, where a deterministic policy specific tells the agent what action should be taken. A stochastic policy, on

the other hand, provides the agent with a probability of taking each action available from that state. However, a deterministic policy can be derived from a stochastic policy, for example by taking the action with the highest probability. [Sutton and Barto 1998]

Definition 2.2 (Policy)
A deterministic policy, $\pi(s)$, is a mapping from state to action $\pi : \mathcal{S} \rightarrow \mathcal{A}$.
A stochastic policy, also noted by $\pi(s)$, is a mapping from a state to a probability of taking a specific action, $\pi : \mathcal{S} \rightarrow [0, 1]$. where $\pi(s, a)$ denotes the probability of taking action a in state s.

A key concept of a Markov decision processes is value functions [Sutton and Barto 1998]. Value functions tell the agent how good it is to be in a given state in terms of how much future rewards that can be accumulated from that state [Sutton and Barto 1998]. For some solutions methods for Markov decision process value functions are used in the search for optimal policies, which will become clear later when the solution methods are described. For now, two different values function are defined. The first value function is called state-value function, denoted $V^{\pi}(s)$. It computes the value of being in state s and then following a policy π. The value can be calculated by the expected amount of reward the agent can expect to get from state s.

Definition 2.3 (State-Value Function)
The state-value function, $V^{\pi}(s)$, for a given state, s, and policy, π, is defined as in [Sutton and Barto 1998, p. 69].

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \quad \forall s \in \mathcal{S} \tag{2.6}$$

where $\mathbb{E}_{\pi}[\cdot]$ is the expectation operator for a policy π that the agent follows.

The value of a state s, which is given by the state-value function, can be expressed in terms of a immediate reward and the value of the successor state, s'. This recursive relationship between $V^{\pi}(s)$ and $V^{\pi}(s')$ is known as the Bellman equation [Sutton and Barto 1998]. Mathematically the Bellman equation can be derived starting from the definition of the state-value function. This derivation follows the

one in [Sutton and Barto 2017],

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma \left(\sum_{k=1}^{\infty} \gamma^k R_{t+k+2} \right) \Big| s_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[\sum_{k=1}^{\infty} \gamma^k R_{t+k+2} \Big| S_{t+1} = s' \right] \right] \qquad (2.7) \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma V^\pi(s') \right] \qquad (2.8) \end{aligned}$$

Equation (2.8) is called the Bellman equation for $V^\pi(s)$. As seen from the Bellman equation it expresses a way of calculating the expected accumulated return using the immediate reward and the value of the successor state, s'. It averages over all values of possible transitions to successor states weighting each by the probability of taking the action that leads to the transition [Sutton and Barto 1998, p. 70].

The second value function is the action-value function, denoted $Q^\pi(s, a)$. The motivation for introducing $Q^\pi(s, a)$ will become clear later, but first the relation between $V^\pi(s)$ and $Q^\pi(s, a)$ is described. The action-value function is close related to the state-value function but is also conditioned on the action, a. Instead of measuring the value of being in a specific state, it measures the quality of taking action, a, when the agent is in state, s.

Definition 2.4 (Action-Value Function)
The action-value function, $Q^\pi(s, a)$, for a given state-action pair s, a is defined as [Sutton and Barto 1998, p. 69].

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s, A_t = a \right], \qquad (2.9)$$

where $\mathbb{E}_\pi[\cdot]$ is the expectation operator for a policy π that the agent follows.

In the search for optimal behaviour of the agent, one wants to keep improving the current policy. A policy π is better than another policy π', that is $\pi \geq \pi'$, if the expected return for π is greater than that for π', meaning that $V^\pi(s)$ for all states $s \in \mathcal{S}$. If one policy is the best among all other policies, that policy is called the optimal policy and is denoted π^* [Sutton and Barto 1998, p. 75]. In principle, there might be multiple optimal policies [Sutton and Barto 1998, p. 75].

Definition 2.5 (Optimal Value Functions)
The optimal state-value function is a function $V : \mathcal{S} \to \mathbb{R}$ defined as

$$V^*(s) = \max_\pi V^\pi(s) \quad \forall s \in \mathcal{S} \qquad (2.10)$$

Similar, the optimal action-value function is defined as

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \quad \forall s \in \mathcal{S} \text{ and } \forall a \in \mathcal{A}(s)$$

Writing the Bellman equation for the optimal state-value function V^* gives the Bellman optimality equation. The derivation of this equation follows that in [Sutton and Barto 1998]. The interpretation of the Bellman optimality equation is that the value of a state evaluated for an optimal policy must be equal to expected return when in state s and picking the best action in this state.

$$\begin{aligned}
V^*(s) &= \max_{a \in \mathcal{A}(s)} Q^{\pi^*}(s,a) \\
&= max_a \mathbb{E}_{\pi^*} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right] \\
&= \max_a \mathbb{E}_{\pi^*} \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \middle| S_t = s, A_t = a \right] \\
&= \max_a \mathbb{E}_{\pi^*} \left[R_{t+1} + V^*(s') \middle| S_t = s, A_t = a \right] \\
&= \max_a \sum_{s',r'} p(s', r|s, a) \left[r + \gamma V^*(s') \right]
\end{aligned}$$

This means that for reinforcement learning methods, that use value function to find optimal policies, such an optimal policy can be derived from an optimal value function, by picking the best action in each state, where best means the action that maximizes the value of the next state in each state. This means that a policy that acts greedy on the optimal state-value function V^* is an optimal policy [Sutton and Barto 1998]. This also applies to the action-value function

$$Q^*(s,a) = \sum_{s',r'} p(s', r|s, a) \left[r + \gamma \max_{a'} Q^*(s', a') \right]$$

However, finding an optimal policy in practice almost never happens, only if the state and action spaces are small, these can be found. However, acting greedy according to an approximation of an value function could still lead to good policies.

A key difference between solving a reinforcement learning problem and solving a Markov decision process is that in a reinforcement learning problem there is typically no information about the transition dynamics of the underlying Markov decision process [Arulkumaran et al. 2017].

So far, the reinforcement learning framework has been introduced and how to make the agent achieve a goal by extracting an optimal policy from value function. It has not been mentioned how to actually compute the value function in order to find good policies. In general there are two main approaches to solve the reinforcement learning problem. One of them uses estimated value functions to structure the searching for good policies, where the other instead search for good policies directly in the space of policies [Arulkumaran et al. 2017]. Respectively, these

methods are called value-based and policy-based methods. The solution methods presented in this section are all table based methods, which means that the value function is represented in terms of a look up table. Each state, $s \in \mathcal{S}$, is an entry in a table with a corresponding value $V(s)$ or $Q(s, a)$ for each entry.

$\mathbf{s}$	$\mathbf{V}^{\pi}(s)$
s_1	$V^{\pi}(s_1)$
s_2	$V^{\pi}(s_2)$
s_3	$V^{\pi}(s_3)$
$\vdots$	$\vdots$

In the next sections algorithms to actually compute value functions in order to extract policies from these are described.

2.2 Dynamic Programming

Dynamic programming is a class of algorithms that can be used to find optimal policies for a Markov decision process under the assumption of known transition dynamics of the model [Sutton and Barto 1998, p. 89]. So, given $\{\mathcal{S}, \mathcal{A}, p(s'|s, a), r(s, a), \gamma\}$, one wants to find optimal behaviour in terms of calculating an optimal value function and extracting an optimal policy π^* from it. This computational approach to solving a Markov decision process is also considered as a model-based approach, since a model is utilized to find π^* [Hester 2013, p. 13].

In general, dynamic programming is a divide-and-conquer approach of solving complex problems by breaking the complex problem into simpler subproblems [Thomas, Cormen, and Leiserson 2009, p. 359]. The idea is that each of these subproblems occur multiple times, when solving the overall complex problem. The optimal solution to each of the subproblems are then computed and stored for later use when that subproblem occurs again.

In dynamic programming value functions are used to find good policies [Sutton and Barto 1998, p. 89]. Two of the most common algorithms in dynamic programming are policy iteration and value iteration [Sutton and Barto 1998, p. 108]. These are two special cases of the general idea of generalized policy iteration. Common to these methods are that they involve the two steps, policy evaluation and policy improvement.

2.2.1 Policy Evaluation

In dynamic programming, policy evaluation is a way of computing the state-value function, V^{π}, for an arbitrary policy, π [Sutton and Barto 1998, p. 90]. Recall, that V^{π} is the state-value function calculated for all the states in the state space, $V^{\pi}(s)$ for all $s \in \mathcal{S}$. Thus, it computes the value for each state $s \in \mathcal{S}$ given a specific policy.

Often policy evaluation is also referred to as the prediction problem, because it estimates how good a state is based on computations of the state-value function

[Sutton and Barto 1998, p. 90]. An iterative algorithm called iterative policy evaluation can be used to evaluate an arbitrary policy for all states in the state space. The idea is to use the Bellman equation given in 2.8 as an update rule for computing the state-value function. The update rule is given by

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma V^\pi(s') \right]$$

where $\pi(s|a)$ is the probability of taking action a in state s and $V_k(s')$ is the value function from the previous iteration of policy evaluation. Asymptotically, iterative policy evaluation can be shown to converge to the value function, $V^\pi(s)$, using the update rule given in Equation 2.2.1, but is omitted in this project.

For iterative policy evaluation a natural question is when has the algorithm converged to $V^\pi(s)$ and can be stopped. For a given threshold $\epsilon > 0$, the algorithm can be stopped when the difference from iterative to iterative is below this threshold. Mathematically this is expressed as

$$\max_{s \in \mathcal{S}} |V_{k+1}(s) - V_k(s)| < \epsilon. \tag{2.11}$$

When a policy, π, has been evaluated, hence values for each state is found using the policy π, one wants to improve π by a finding a new policy, π', that is better or at least as good as the old policy. This approach is called policy improvement.

2.2.2 Policy Improvement

When a policy has been evaluated using policy evaluation and an approximate value function has been estimated, one can use this to find even better policies [Sutton and Barto 1998]. This is known as the policy improvement step, but is also referred to as control. In improving a policy one would like to know whether a policy for some state should be change to pick another action instead of following the policy. A greedy policy $\pi'(s)$ can be constructed by picking the action that looks best after only one look ahead.

$$\begin{aligned} \pi'(s) &= \operatorname*{argmax}_{a \in \mathcal{A}(s)} Q^\pi(s, a) \\ &= \operatorname*{argmax}_{a \in \mathcal{A}(s)} \sum_{s', r} p(s', r|s, a) \left(r + \gamma V^\pi(s') \right) \end{aligned} \tag{2.12}$$

This is the same as the Bellman optimality equation, and therefore policy improvement ensure that the policy improves from iteration to iteration, except when the policy is already the optimal one. The policy improvement step therefore uses a greedy policy which guarantees the policies not to be worse than the old one.

2.2.3 Generalized Policy Iteration

In generalized policy iteration, policy evaluation and policy improvement repeatedly interact with each other to compute an optimal policy [Sutton and Barto 1998, p. 105]. The general idea behind generalized policy iteration is summarized in Figure 2.2. First, one initialize a value function, for example $V(s) = 0$ for all $s \in \mathcal{S}$, and a policy, for example a random policy. The policy evaluation step then evaluates the current policy computing $V^{\pi}(s)$ for a number of steps. This step is shown as the arrows pointing upwards in Figure 2.2. The new value function is then used to find a better policy by greedily picking the best action in each state.

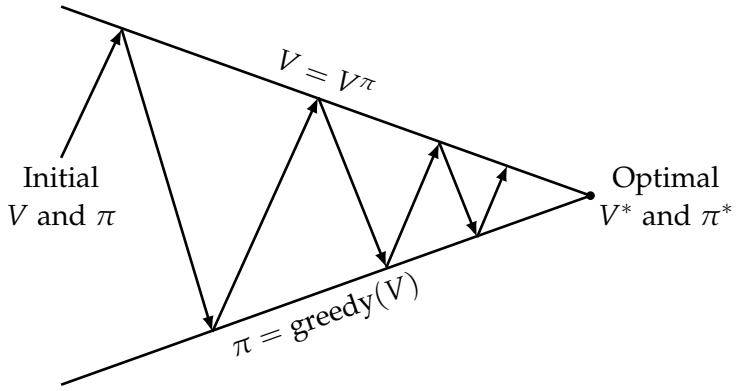


Figure 2.2: Illustration of how policy iteration converges to the optimal policy and value function. The illustration is inspired from the picture in [Sutton and Barto 1998, p. 106].

The two algorithms policy iteration and value iteration are both algorithms that all into the category of generalized policy iteration. Policy iteration will converge to an optimal policy, but a disadvantage of this algorithm is that for each new policy found by the policy improvement step, the policy evaluation step has has to run until the value function V^{π} has converged [Sutton and Barto 1998]. This can be time very time consuming if it converges slow. Instead one can use the algorithm called value iteration, where only a single iteration of policy evaluation is computed in between each policy improvement step. The value iteration algorithm is seen in Algorithm 1.

2.3 Q-Learning

Q-learning is a type of algorithm called temporal-difference learning [Sutton and Barto 1998]. Temporal-difference learning combines ideas from both dynamic programming and Monte Carlo methods [Grondman et al. 2012]. Like Monte Carlo methods, temporal-difference learning does not require any knowledge of the underlying Markov decision process, but instead use experience through sampling of episodes [Sutton and Barto 1998]. It can thus learn how to behave in the environment only through sampled experience.

Monte Carlo methods have the advantage of not requiring a model of the Markov decision process in order to achieve optimal behaviour. However, it is

Algorithm 1: Value Iteration [Sutton and Barto 2017]

Initialize value function V, e.g. by $V(s) = 0$ for all states $s \in \mathcal{S}$.
Repeat
  $\Delta \leftarrow 0$
  Repeat for all $s \in \mathcal{S}$.
  $v \leftarrow V(s)$
  $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a) \left(r + \gamma V(s')\right)$
  $\Delta \leftarrow \max\left(\Delta, |v - V(s)|\right)$

Output deterministic policy, π, such that
$\pi(s) = \arg\max_a \max_a \sum_{s',r} p(s', r|s, a) \left(r + \gamma V(s')\right)$

based on full sampled episodes, which in some application might not be attractive since episodes can be very long [Sutton and Barto 1998]. Also, in some applications there might not even be a terminal state, since the agent in principle continues to interact with the environment to eternity. Compared to this approach, temporal-difference learning is instead able to immediately update the estimate of the action-value function after a single time step. So instead of waiting to update the action-value function until the end of an episode, it can be done after the agent has perform a transition from s_t to s_{t+1}.

In the literature, Q-learning is considered to be one of the most important breakthroughs in reinforcement learning [Arulkumaran et al. 2017]. Q-learning is an off-policy method that is able to estimate the optimal action-value function, $Q^*(s, a)$, regardless of what policy the agent follows. This means that $Q^*(s, a)$ can be estimated for example from a random policy without any knowledge of the optimal policy. For a one-step Q-learning approach, the update rule is given by

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t))\right) \tag{2.13}$$

where α is the learning rate, γ is the discount factor [Sutton and Barto 1998]. $Q'(s_t, a_t)$ will converge asymptotically to the optimal action-value function $Q^*(s_t, a_t)$. The Q-learning algorithm is seen in Algorithm 2.

Algorithm 2: Q-Learning [Sutton and Barto 1998]

Initialize $Q(s, a)$ arbitrary.
Repeat for a number of episodes.
  Initialize s.
  Repeat until episode terminates.
    Choose a from s using policy derived from Q.
    Take action a and observe r and s'.
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)$.
    Update current state, $s \leftarrow s'$.

2.4 Grid World Navigation

In this section some of the methods described for solving the reinforcement learning problem will be applied to a small grid environment. This should help understanding the theory and build a connection between theory and practice, and see that it can be applied to solve a simple navigation problem. An overview of the grid world environment is seen in Figure 2.3. For details about the grid world environment and how it is conducted, see Appendix B.

The goal of the agent is to get to the finish square located at $(x, y) = (8, 7)$ in the grid. This is achieved through a number of available actions the agent can perform given by the discrete set, $\mathcal{A} = \{a_{\text{up}},\ a_{\text{down}},\ a_{\text{left}},\ a_{\text{right}}\}$. The state space, $\mathcal{S}$, consists of all the white squares in Figure 2.3, denoted $s_{(\text{x,y})}$. The agent is not able to move to the black squares or out of the environment, and the action space is therefore limited for example $\mathcal{A}(s_{(5,3)}) = \{a_{\text{up}}, a_{\text{left}}\}$.

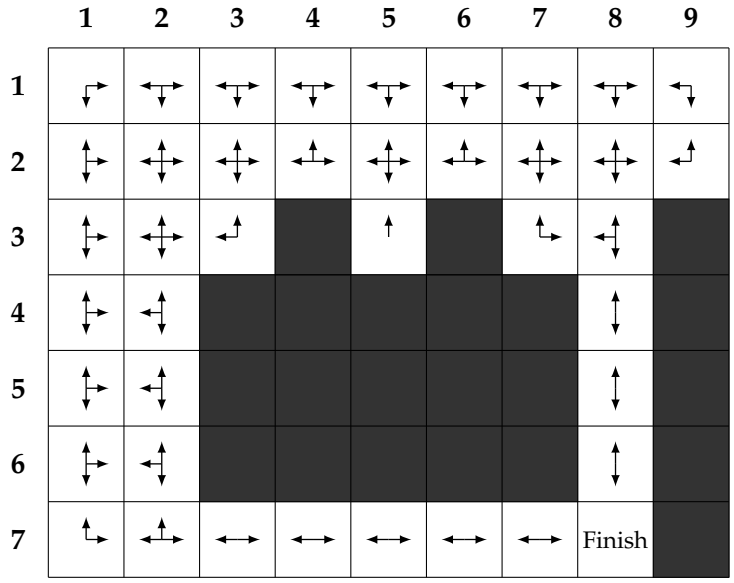


Figure 2.3: Grid world environment example. The white squares are the available positions of the agent in the environment and the black squares represent unavailable positions. The arrows show the possible movements of the agent for a given state.

The environment in Figure 2.3 is related to the notation of Markov decision processes, which was described in Section 2.1. This relationship is seen in Figure 2.4. In this Markov decision process of the grid world environment the probabilities of the available actions in a given state is uniformly distributed. Given the agent is in a state, for example $s_{(3,2)}$, and selects action a_3, the transition probability $p(s', r|s, a)$ is 1.0 of ending up in state $s_{(4,2)}$. However, one could reduce this probability to model uncertainty into the environment, for example by allowing the agent to end up in a neighbour state if action a_3 in state $s_{(3,2)}$ is selected. This uncertainty is not modelled in this project.

In the next sections, solutions to the grid world problem using dynamic programming and Q-learning will be presented. The idea is to apply the theory to

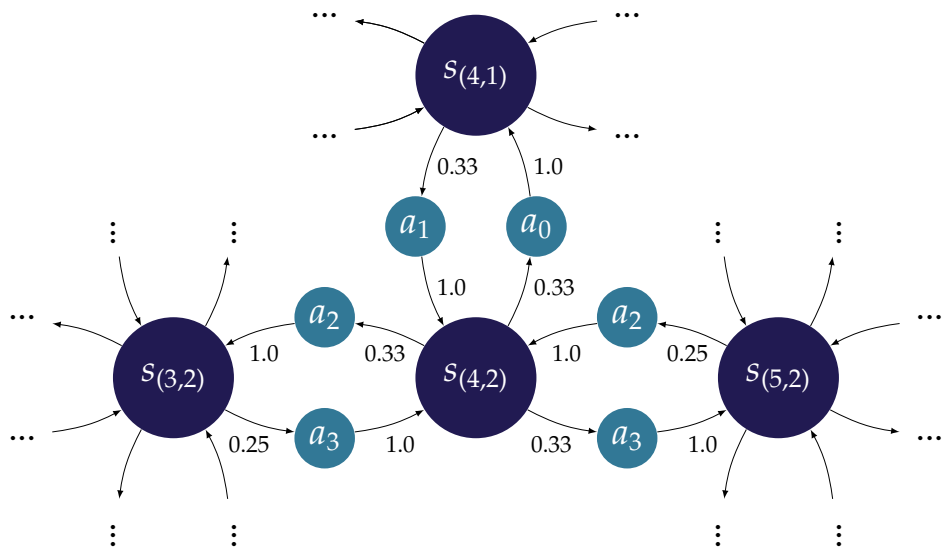


Figure 2.4: Illustration of the relationship between the grid world environment and a Markov decision process. The Markov decision process for the whole grid world environment is to big to show, so instead a little part of it is shown. The action space is $\mathcal{A} = \{a_{\text{up}}, a_{\text{down}}, a_{\text{left}}, a_{\text{right}}\} = \{a_0, a_1, a_2, a_3\}$.

a small navigation example to see if these methods converge to the same solution or solutions close to each other. It also makes it possible to compare the methods to the methods presented in Chapter 4. This should help the analysis of what the more complex algorithms learns.

2.4.1 Dynamic Programming

In this section, dynamic programming is used to solve the reinforcement learning problem for the grid world environment just described. The value iteration algorithm presented in Algorithm 1 is used, where one policy evaluation step is followed by a policy improvement step. The initial value function and initial policy is shown in the top of Figure 2.5. The value function is initialized by setting $V(s) = 0$ for all $s \in \mathcal{S}$. Also, by definition the value of the terminal state is zero [Sutton and Barto 1998]. Furthermore, the initial policy is chosen to be a uniform distribution over the available actions from that state, where the available actions in each state are the set of actions that does not bump into walls or moves the agent out of the grid. The policy shows the action with the highest Q-value in each state, but for states where multiple actions share the same highest Q-value, all these actions are shown. This means that all actions in such a case are equally good.

The next pairs of value function and policy are for iteration 3, 10 and 25 of the value iteration algorithm, respectively. As seen from the third iteration of the value iteration algorithm, the value function spreads out from the finish state, where the transitions to the finish state is rewarded by 1.0. Since dynamic programming sweeps through the whole state space for each iteration the value for each state

$s \in \mathcal{S}$ is updated, which also means that the Q-values for each state are updated. The Q-values are used to extract a policy from the value function from iteration to iteration.

The value of a state can be calculated by utilizing the Bellman equation in Equation 2.2.1, and as an example the value of state $s_{(8,6)}$ is calculated. This state has two available actions each with a probability of 0.5 of being selected. Each of these actions transitions to a single successor state with a probability of 1.0. However, uncertainty can be build into the model by allowing the agent to transition to multiple successor states, but this is omitted in this project. The value of the state can then be calculated giving the initial policy as follows

$$\begin{aligned} V^{\pi}(s_{(8,6)}) &= \sum_{a} \pi(a|s) \sum_{s'} \sum_{r} p(s', r|s, a) \left[r + \gamma V^{\pi}(s') \right] \\ &= 0.5 \cdot 1.0 \cdot (1.0 + 0.9 \cdot 0.0) + 0.5 \cdot 1.0(-0.05 + 0.9 \cdot 0.0) \\ &= 0.475 \end{aligned}$$

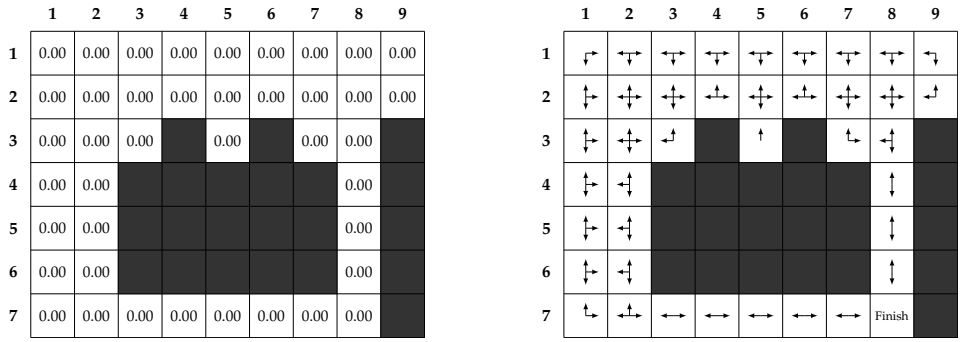
After the update of the value function, the policy is then updated by making the it greedy with respect to the current value function. Since transitioning to the terminal state $s_{(8,6)}$ receives a reward of 1.0, the greedy policy will now prefer this action. This results in a new policy where $\pi(s_{(8,6)}, a_1) = 1.0$. The value function for $s_{(8,6)}$, given this new policy, is then calculated by

$$\begin{aligned} V^{\pi}(s_{(8,6)}) &= 1.0 \cdot 1.0(1.0 + 0.9 \cdot 0.0) \\ &= 1.000 \end{aligned}$$

This approach of alternating between updating the value function and policy leads to the optimal value function, and hence also the optimal policy. This is seen after 25 iterations of the value iteration algorithm, where the value function has converged to a stationary solution. Since this is a small example and the grid world environment is intuitive to understand, it is also possible to verify the correctness of dynamic programming applied to solve this navigation problem. After the value function has converged to the optimal one, each additional step away from the terminal state reduces the value of that state calculated by the Bellman equation. This reduction is determined by the discount factor and the reward function, which in this case penalizes each transition not ending up in the terminal state by -0.05.

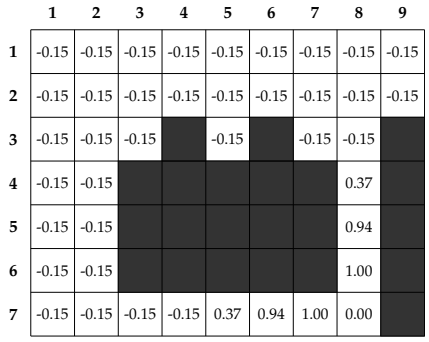
By examining the policy after 25 iterations, it is seen that in each state, the policy moves the agent one step closer to the terminal state. This is easy to verify, that this policy is indeed the optimal one for this grid world environment. For a script that implements dynamic programming to solve the reinforcement learning problem for the grid world environment, see Appendix A.

The value function calculated using dynamic programming provides an optimal value function. This is used as an analytic solution to the grid world environment which is used to see if other solution methods converge to the same analytic solution. To be able to compare the solution presented in this section to other methods, such as Q-learning, the action-value function is shown in Figure 2.6. Instead of showing the value for each state the Q-value for each action is shown along the side of the corresponding action.

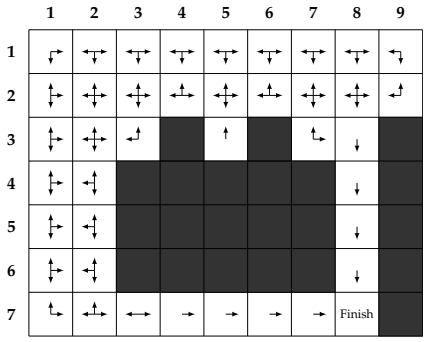


(a) Initial value function, V^{π}.

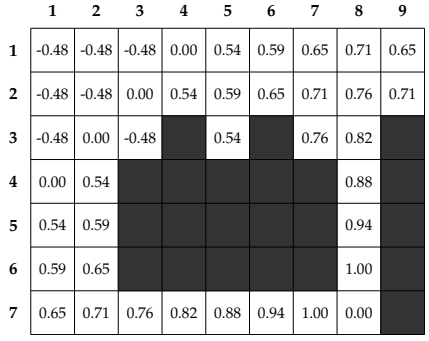
(b) Initial policy, π.



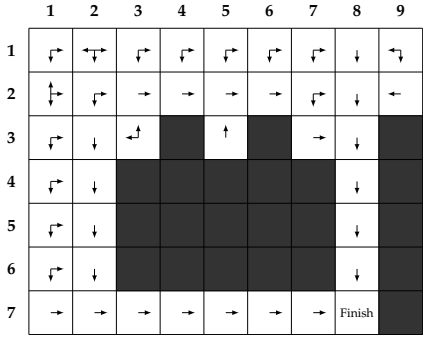
(c) V^{π} after 3 iterations.



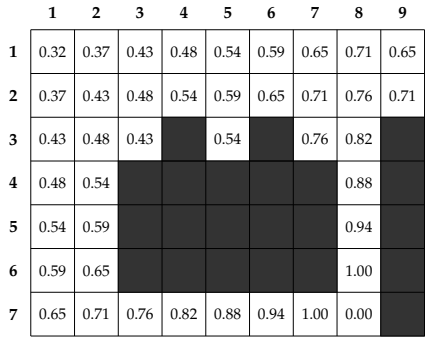
(d) π after 3 iterations.



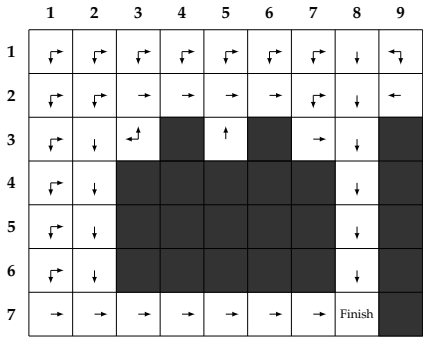
(e) V^{π} after 10 iterations.



(f) π after 10 iterations.



(g) V^{π} after 25 iterations.



(h) π after 25 iterations.

Figure 2.5: Convergence of value iteration for the grid world environment. The left column shows the value function after just one iteration of iterative policy evaluation for the current policy. The right column shows the greedy policy extracted from the estimate value function. If multiple actions achieve the same maximum Q-value for a given state, the policy for that state is a uniform distribution over these action.

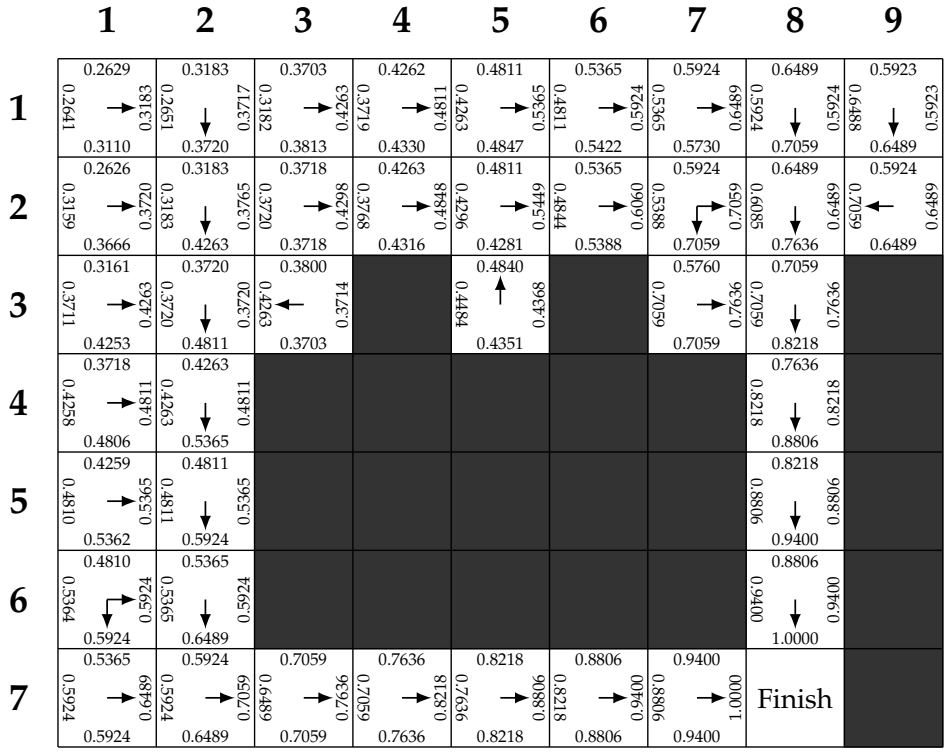


Figure 2.6: Illustration showing the Q-values, $Q(s, a)$, for each state and action pair when the value iteration has converged. The four values are the Q-values for the action that moves the agent in the given direction.

2.4.2 Q-Learning

In this section the Q-learning algorithm in Algorithm 2 is applied to solve the reinforcement learning problem for the grid world example. Since Q-learning can solve the reinforcement learning problem without prior knowledge of the model, the set up is a bit different from dynamic programming. Instead of limiting the action space in some states, the agent is allowed to bump into walls by allowing all four actions to be available in all of the states. If the agent performs an action that results in collision with a wall, the state of the environment is unchanged. This means that just by sampling interactions of the agent acting in the environment, one can improve a policy. Te environment is illustrated in Figure 2.7.

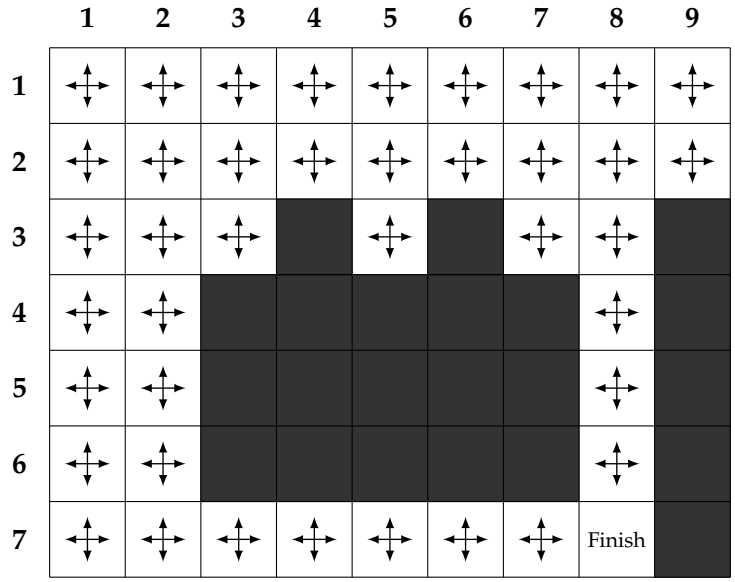


Figure 2.7: Grid world example of navigation. The white squares are the available positions of the agent in the environment and the black squares represent unavailable positions. In this grid world environment, the agent is allowed to pick all available actions.

Compared to dynamic programming where the solution methods sweep through the whole state space for each iteration, each Q-value is only updated when the agent discovers the state. This means that some states might be updated more than others. Extracting a greedy policy could then lead to too much exploitation of the knowledge the agent has learned. The agent should keep discovering the whole state space to be able to search for better policies. To do this an ϵ-soft policy is used. This means that if a random sampled number between 0 and 1 is under a specified ϵ value a random action is picked instead of selecting the one with the highest Q-value [Sutton and Barto 1998]. This is used to let the agent discover states that might not be visited a lot of times in order to update the Q-value for these states, which could eventually lead to better policies.

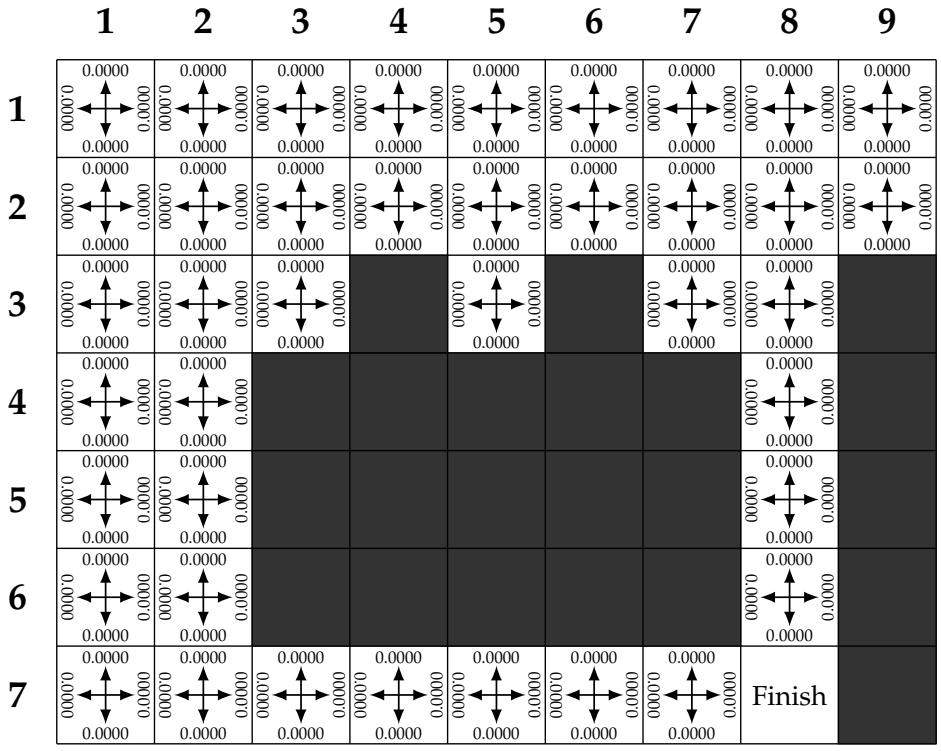
A script that implements the Q-learning algorithm can be found in Appendix A. As seen from Figure 2.9 an optimal policy is achieved after 300 episodes regardless of the action-value function being the optimal one. By examining Figure 2.8 one can see that the Q-values for states close to the goal convergence faster to the

optimal ones, than the ones far away from the goal. This has several reasons. First, the agent has to discover the goal in order to get the reward for achieving the goal.

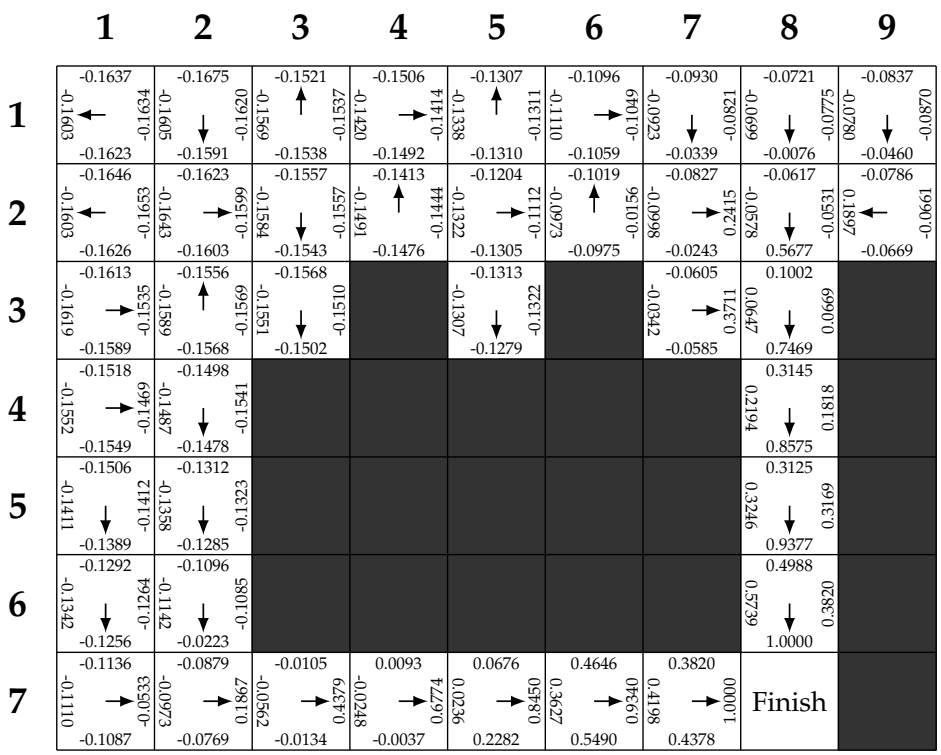
Another reason that the Q-value for the state $s_{(1,1)}$ convergence slower, is this this state is not visited as many time as the state closer to the goal, and therefore has not its Q-values updated as much. The ratio of exploration and exploitation used in the algorithm is $\epsilon = 0.3$. This is used to maintain exploration to states that is not visited that often. If $\epsilon = 1$, the agent will pick a random action every time, which means that it will not exploit any knowledge it has learned.

In practice, however, some of the traditional ways of solving a reinforcement learning problem are infeasible. This could be due to the huge number of states in the state space of real world reinforcement learning problems. Therefore, the field of classical reinforcement learning can benefit from techniques in deep learning. This leads to an introduction of some of the theory of deep learning, which is described next.

If the problem in hand has a model with a huge number of states, it might be infeasible to stores the value function as a table for later use. Instead of a table, the value function can be considered as a parametrized function that approximates the value function for a given input. Model in deep learning has the ability to approximate any function, no matter the complexity of the function. In this way deep learning and reinforcement learning is combined into deep reinforcement learning. Before describing the techniques in deep reinforcement learning, some theory of deep learning is needed.

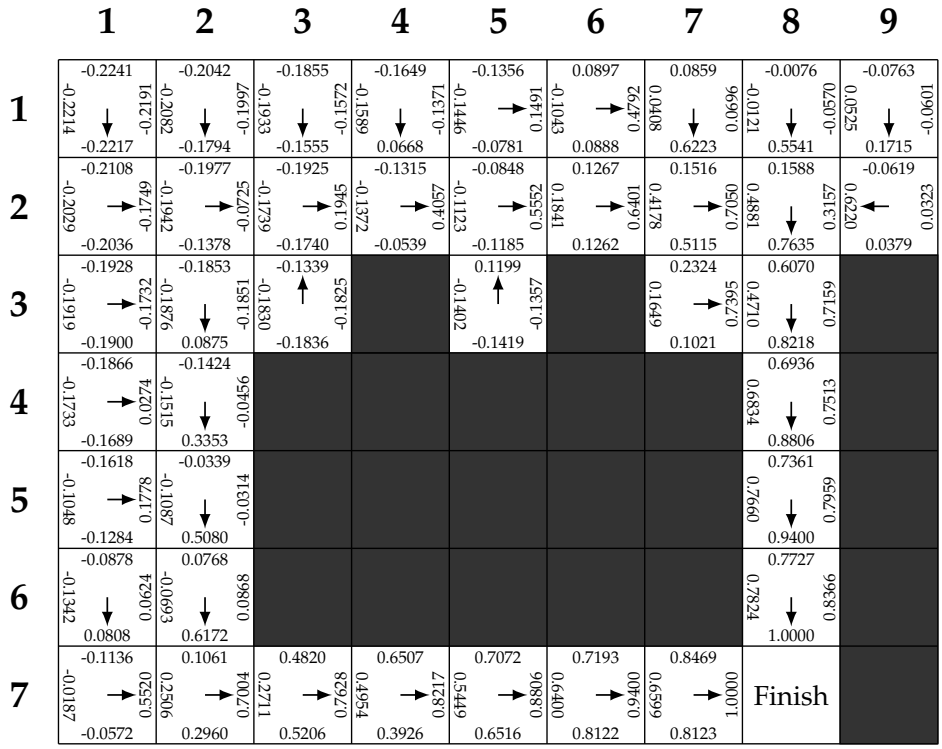


(a) Initial $Q^{\pi}(s, a)$.

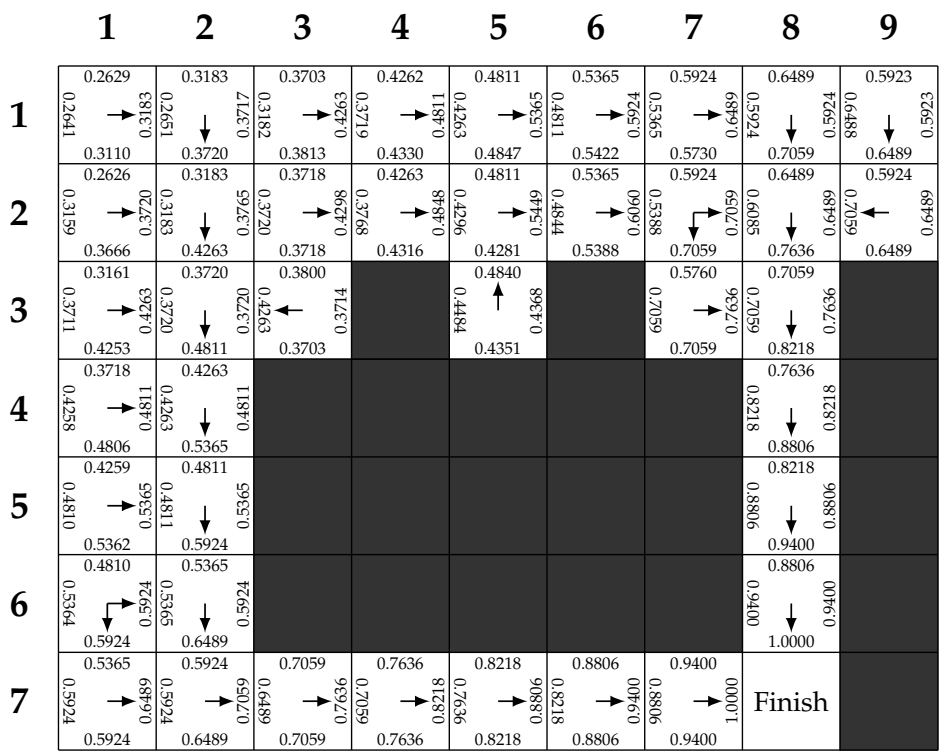


(b) $Q^{\pi}(s, a)$ after 100 episodes.

Figure 2.8: Convergence of Q-learning for the grid world environment. The four values at each line of each square represents the Q-value of going in that direction. The arrows in each square represents a greedy policy.



(a) $Q^\pi(s, a)$ after 300 episodes.



(b) $Q^\pi(s, a)$ after 10000 episodes.

Figure 2.9: Convergence of Q-learning for the grid world environment. The four values at each line of each square represents the Q-value of going in that direction. The arrows in each square represents a greedy policy.

Chapter 3
Deep Learning

In machine learning, the algorithms is able to learn from data. In some cases solving a task or problem using a fixed designed program might be too difficult [Goodfellow, Bengio, and Courville 2016, p. 97]. Such a task could be to classifying images by determining whether a given image contains a house, a tree, or something else. Solving this task with a fixed program is very difficult, especially if the number of classes is large. Instead, one could address this task by presenting the algorithm for a number of images of trees and houses and let it learn how to classify the images from these data samples.

The input to conventional machine learning algorithms is hand crafted features, which is extracted from the data itself [Lecun, Bengio, and Hinton 2015]. Extracting these features from the data typically require much knowledge of the domain in order to design a useful feature extractor. However, instead of hand crafting the feature extractor, it can be learned directly from the data using deep learning models. Deep learning models typically consist of multiple layers each transforming its input to a more abstract representation using non-linear functions [Lecun, Bengio, and Hinton 2015]. By building a model with multiple of such transformations very complex functions can be learned. This also means that deep learning models are typically able to process the raw data by learning the features through training of the parametrized transformations [Lecun, Bengio, and Hinton 2015].

In this chapter two models in deep learning is presented, which are called feedforward neural networks and convolutional neural networks. Before introducing these two models artificial neurons are first presented, which are small mathematical models neural networks consist of. Furthermore, a motivation for using neural networks is given by the universal approximation theorem, which states that neural networks can approximation any function. Lastly, the learning procedure for adjusting the weights to fit a desired function is described.

3.1 Artificial Neuron

An artificial neuron is a simple mathematical model that mimics the way biological neurons in the human brain process information [Haykin 2009, p. 32]. In order to understand neural networks, one first need to be familiar with the computational units they consist of. An illustration of a model of a neuron is seen in Figure 3.1.

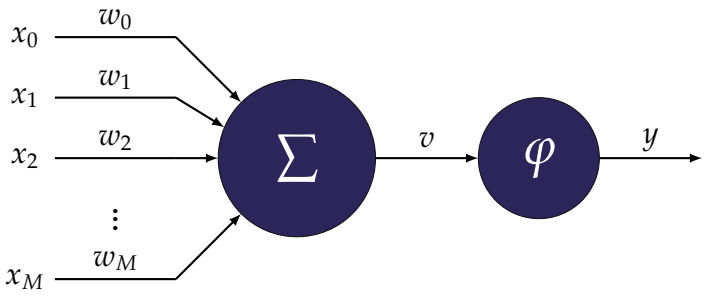


Figure 3.1: Illustration of how an artificial neuron transforms its inputs $x_1, x_2 \ldots, x_M$ to its output y.

A neuron is a computational unit that takes as input a number of M input signals, $x_1, x_2, \ldots, x_M$, and combine them into a scalar output [Haykin 2009], thus it can be considered as a mapping $h : \mathbb{R}^M \to \mathbb{R}$. Each of the input signals are multiplied by their own weight $w_1, w_2, \ldots, w_M$, that determines how much the individual input signals affect the output of the neuron. These weighted inputs are summed together and a bias, b, is added. The bias has the effect of applying an affine transformation to the sum [Haykin 2009, p. 41], before it is passed through an activation function, φ. Formally, the output is expressed by the two equations

$$v = \sum_{j=1}^{M} w_j x_j + b \tag{3.1}$$

and

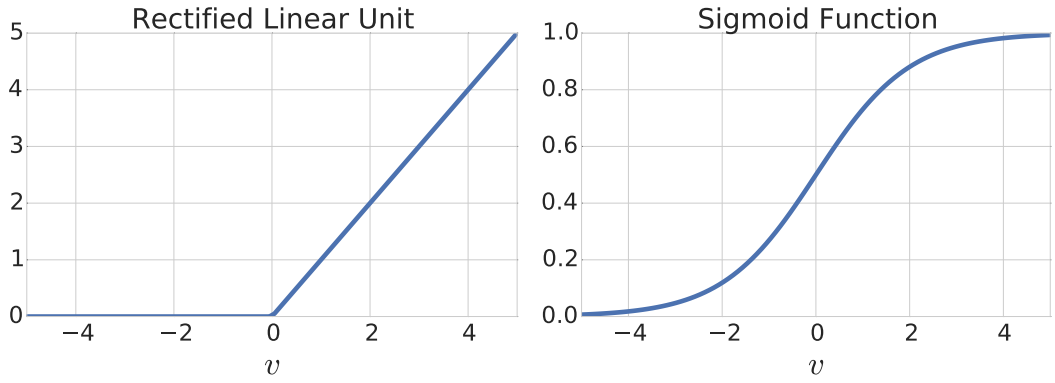
$$y = \varphi(v) = h(x_1, x_2, \ldots, x_M) \tag{3.2}$$

The bias can be incorporate in the sum by setting $x_0 = 1$ and letting $w_0 = b$, as in the illustration in Figure 3.1. After v has been computed, it is passed through an activation function, that has the affect of applying a non-linearity to v. There are different activation functions that can be used, where the sigmoid function and rectified linear unit function, often called ReLU, are two of the most popular choices [Goodfellow, Bengio, and Courville 2016]. The activation function is some times also called squashing functions as they limits the output value of a neuron [Haykin 2009, p. 40]. The sigmoid is an example of such a squashing function as it limits the range of $y \in [0, 1]$. A formula for the sigmoid activation function is given in Equation 3.3 and the graph is seen in Figure 3.2b. The ReLU function, expressed in Equation 3.4, does not set an upper limit of the output value and does not allow negative output. A graph of the ReLU function is seen in Figure 3.2a.

$$\phi(v) = \frac{1}{1 + e^{-v}} \tag{3.3}$$

$$\phi(v) = \max(0, v) \tag{3.4}$$

An important property of both of these functions are that they are non-linear. Differentiability is also a particular important property as it allows for gradient-based learning methods.



(a) The ReLU activation function. **(b)** The sigmoid activation function

Figure 3.2: Two different types of activation function.

Neurons are used to construct networks of connected neurons also known as neural networks. The neurons are organized in different layers and these layers are then stacked to build larger networks. There are different types of neural networks for which feed-forward neural networks and convolutional neural network are two common types.

3.2 Feed-Forward Neural Networks

A feed-forward neural network is a network of neurons that is used to approximate some desired function $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$, where $\boldsymbol{x}$ is the input and $\boldsymbol{\theta}$ is a set of parameters of the model. A feed-forward neural network uses multiple layers of non-linear transformation to transform the input into new representations [Lecun, Bengio, and Hinton 2015]. In a feed-forward neural network the information flows through the network from the input layer to the output layer without any form of feedback. Feedback is where the output of a neuron is fed as input to itself. For every layer the input passes through more abstract representation of the input is produced.

One distinguish between three different types of layers in a feed-forward neural network called input layer, hidden layers, and output layer. The input layer consists of a number of input neurons corresponding to the dimension of the input vector, $\boldsymbol{x} \in \mathbb{R}^M$. The output layer has a neuron for each dimension in the output, $\boldsymbol{y} \in \mathbb{R}^N$. A neural network can thus be considered as a mapping $f : \mathbb{R}^M \to \mathbb{R}^N$.

Between the input and output layer, a number of hidden layers are inserted. If multiple hidden layers are inserted, the approximated function is composed by multiple functions [Goodfellow, Bengio, and Courville 2016]. For example consider the feed-forward neural network seen in Figure 3.3. This network can be considered as a function $f : \mathbb{R}^3 \to \mathbb{R}^2$. In this case the function f is composed of three

other functions as seen in Equation 3.5.

$$\boldsymbol{y} = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right) \tag{3.5}$$

Each of these functions transforms its input to an output by the formula,

$$\boldsymbol{h} = \varphi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}), \tag{3.6}$$

where $\boldsymbol{h} \in \mathbb{R}^{n\times 1}$ is an hidden abstract representation, $\boldsymbol{W} \in \mathbb{R}^{n\times m}$ is a weight matrix, and $\boldsymbol{b} \in \mathbb{R}^{n\times 1}$, and φ is a non-linear activation function applied entry-wise to the output of the transformation $\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$. Doing this for all the layers in the network, results in

$$\boldsymbol{y} = \varphi^{(3)}\left(\boldsymbol{W}^{(3)}\varphi^{(2)}\left(\boldsymbol{W}^{(2)}\varphi^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) + \boldsymbol{b}^{(2)}\right) + \boldsymbol{b}^{(3)}\right) \tag{3.7}$$

The parameter set of the network in Figure 3.3 is then given by all the weight matrices and bias vectors in the model given by the parameter set

$$\boldsymbol{\theta} = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{W}^{(3)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \boldsymbol{b}^{(3)}\} \tag{3.8}$$

When talking about the network architecture, this concerns the number of layers in the network and how many neurons each of these consists of, which has an effect on $\boldsymbol{\theta}$.

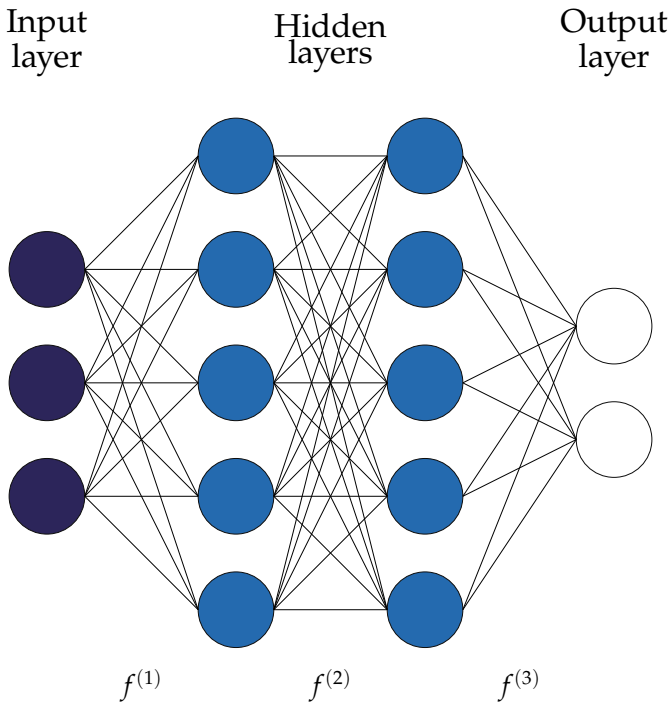


Figure 3.3: A fully connected feed-forward neural network with two hidden layers. The information flows from left to right.

Feed-forward neural networks typically consists of multiple hidden layers that maps the input to high dimensional representations. Typically such models have million of parameters which allows it to approximate very complex functions.

3.3 Universal Approximation Theorem

In this section, the universal approximation theorem is described as a motivation for the use of neural network in the rest of this project. The universal approximation theorem states that a feed-forward neural network with a single layer can approximate any function, as long as the number of neurons in the hidden layer is sufficiently large. There are different versions of the theorem, but the one given in this section is adapted from [Haykin 2009].

Theorem 3.1 (Universal Approximation Theorem [Haykin 2009])
Let φ be a non-constant, bounded and monotone-increasing continuous activation function and let I_{m_0} denote an m_0-dimensional unit hypercube $[0,1]^{m_0}$. Let $C(I_{m_0})$ denote the space of contentious functions on I_{m_0}. Given any function $f \ni C(I_{m_0})$ and tolerance $\kappa > 0$, there exists an integer m_1 and sets of real constants α_i, b_i and w_{ij} such that

$$F(x_1, x_2, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} a_i \phi \left(\sum_{j=1}^{m_0} w_{ij} x_j + b_j \right) \tag{3.9}$$

may be defined as an approximation of a desired function f. That is

$$|F(x_1, x_2, \ldots, x_{m_0}) - f(x_1, x_2, \ldots, x_{m_0}) d| < \kappa \tag{3.10}$$

for all $x_1, x_2, ..., x_{m_0}$.

The universal approximation theorem states that any continuous function can be approximated with a single hidden layer feed-forward neural network. However, using a single hidden layer with a huge number of neurons might not be the best architecture in terms of parameter efficiency or the ability the generalize to new inputs. Instead neural networks are typically made deeper [Haykin 2009].

3.4 Convolutional Neural Networks

Another type of network is called convolutional neural networks. They are particularly good for processing data where the spatial structure of the data is important [Goodfellow, Bengio, and Courville 2016]. An example of this is images, where pixels close to each other are higher correlated [Goodfellow, Bengio, and Courville 2016] than pixels far away from each other. In applications, convolutional neural networks have shown to be very good at classifying images, by determining whether a given image contains a dog, a ball or something else [Lecun, Bengio, and Hinton 2015]. In this section, convolutional neural networks are introduced by describing the different kind of layers that together constitute these types of neural networks, but first a motivation is given.

One motivation for using convolutional neural networks rather than a fully connected feed-forward neural network is that is allows for sparse connections

in the network in contract to traditional neural networks, where each neurons in adjacent layers are connected to each other [Goodfellow, Bengio, and Courville 2016]. This is achieved by making the input to the network smaller than the size of the filters, which convolutional neural networks consists of. A results of this is an reduction in the number of parameters in the network since the number of connections are smaller.

In addition to the fully connected layers also used in feed-forward neural networks, convolutional neural networks implements two other type of layers, called convolutional layers and pooling layers. These layers inserted before the fully connected layers to extract features from the input.

Convolutional neural networks are neural networks where at least one of the layers uses the convolutional operator, a so-called convoluional layer [Goodfellow, Bengio, and Courville 2016]. Such a layer consists of a predetermined number of filters and another parameter called stride. Each filter have a fixed size, for example 2×2. A feature map for each of the filters is produced by convolving these with the input to the layer, where the stride determines the shift of the convolutional operator. An example of how the convolutional layer functions is seen in 3.4 .

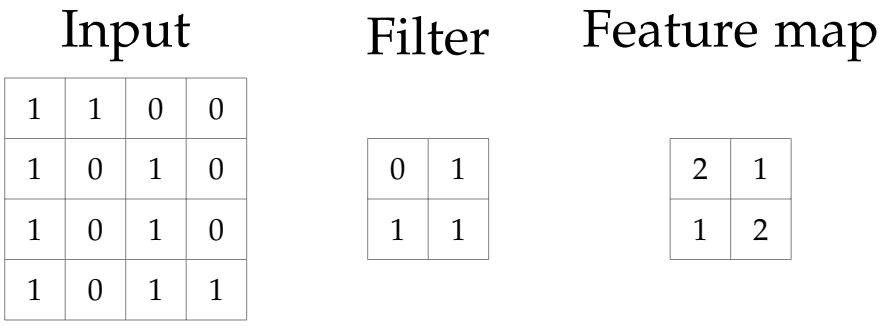


Figure 3.4: Illustration of the relationship between input, filter and feature maps for a convolutional layer.

Another type of layer is called pooling layer. This is used to reduce the spatial size of the feature maps produced by the convolutional layers, to reduce the parameters in the network [Goodfellow, Bengio, and Courville 2016]. A common type of pooling technique is called max pooling and the idea behind it is to reduce a rectangular neighbourhood to only a single number, namely the max value within this area.

After a number of convolutional layers and pooling layers, the features extracted using these layers are concatenated into a single feature vector. This feature vector then serves as the input to a number of fully connected layers. All the neurons in the previous layer are connected to all the neurons in the next layer, hence the name fully connected layers.

3.5 Learning Algorithm

As motivated in Section 3.5 a neural network with a single hidden layer is capable of approximation any function if the hidden layer has enough neurons. However, this theorem does not state what the values of the weights and biases in the network are. Given a network architecture, one wants to adjust these weights and biases given by the parameter, $\boldsymbol{\theta}$, so that it best fits the function one wants to approximate. This can be considered as an optimization problem, where the parameters are adjusted to reduced some cost function, $C(\boldsymbol{\theta})$. Many different optimization methods exist, but for training neural network, methods using gradient information is used to adjust the parameters to find a local minimum in the parameter space. The non-linearity of a neural network implies that the error function becomes non-convex [Bishop 2006], and therefore finding a local minimum, this point cannot be guaranteed to be a global minimum.

The procedure of using a data samples to adjust the parameters in a machine learning model to reduce some cost function is referred to as a learning algorithm [Haykin 2009]. The learning algorithm for training neural networks can be thought of as a three step procedure. First input samples are fed through the network from the input layer to the output layer to compute the estimated outputs for those input samples. This output is used to calculate the cost function. The procure is outlined here:

1. Forward propagate inputs samples, $\boldsymbol{x}$ through the network to compute the activations of the units in both the hidden layers and the output layer.

2. Compute the error between the target and and the network output and back propagate this error back through the network to compute the gradient.

3. Adjust the parameters, $\boldsymbol{\theta}$, based on the gradient information computed in step 2.

The parameters are adjusted according to the procedure of gradient decent optimization. The goal is to find weights and biases in terms of parameters, $\boldsymbol{\theta}$, where the cost function is minimized [Ruder 2017]. Gradient decent methods uses the gradient information to adjust the parameters a small step in the direction of the negative gradient. This allows the parameters to be adjusted towards a local minimum where

$$\nabla C(\boldsymbol{\theta}) = 0 \tag{3.11}$$

The gradients $\nabla C(\boldsymbol{\theta})$ are provided by backpropagation. Typically, the weight space will have many points that satisfy Equation (3.11, and therefore a global optimum cannot be guaranteed.

Therefore an iterative approach to finding a local minimum is used. This means that starting from an initial value of $\boldsymbol{\theta}_0$, one can iteratively adjust the parameters according to the following formula

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla\ C(\boldsymbol{\theta}_k), \tag{3.12}$$

where $\eta > 0$ is called the learning rate.

According to [Ruder 2017] there are three different variations of gradient decent optimization in the context of training a neural network, which differ in the amount of data used to compute the gradient. Batch gradient decent and stochastic gradient decent utilize the whole training data set and a single data sample from the training data set to compute the gradient of the cost function, respectively. Because batch gradient decent uses the whole training data set it can be a slow approach to update the parameters. Rather than computing the gradient based on the whole data set, stochastic gradient decent uses only a single data sample to update the parameters. The third method uses a mini-batch, which uses not the whole data set, but a number between 1 and N. This means that the total error of a mini-batch becomes

$$C(\boldsymbol{\theta}) = \sum_{j}^{J} C_j(\boldsymbol{\theta}), \tag{3.13}$$

where J is the size of a mini-batch. In gradient decent optimization one has to manually select the learning rate, η. This can be a difficult task, and it has to be selected carefully because it has a significant affect on the performance of the model [Goodfellow, Bengio, and Courville 2016, p. 302]. If it is too small, the parameters will converge slowly to an optimum resulting in long training time. On the other hand, if the learning rate is too high, the parameters might end up not converting. Instead of a fixed learning rate, it can adaptively be adjusted according to the gradient. Describing the various algorithms that implement adaptive learning rate is out of the scope of this report.

To compute the gradients used to train a neural network backpropagation is used. It is used to compute the gradients of the cost function with respect to each of the individual adjustable weights in the neural network [Haykin 2009, p. 183]. Thus, it provides a way of determining how much each individual weight contributes to the error given by the cost function. Weights with small gradients have a little impact on the cost function and weights with higher gradients contributes more to the error. In order to compute these gradients a single forward propagation should be perform to compute the error between the targets and the output of the model.

Having introduced the concepts of reinforcement learning and deep learning, these two areas are now combined into what is called deep reinforcement learning.

Chapter 4
Deep Reinforcement Learning

In this chapter deep learning and reinforcement learning is combined into what is called deep reinforcement learning. The methods for solving the reinforcement learning problem in Chapter 2 were table-based solution methods, which is feasible for state and action spaces with a limited number of states and actions [Sutton and Barto 1998]. In cases where both $\mathcal{S}$ and $\mathcal{A}$ are big sets, these methods are not only infeasible due to memory requirements for storing the big value function tables, but also due to the data needed to fill out these tables accurately and the time needed for acquiring that amount of data [Sutton and Barto 1998, p. 193].

If the state is represented by a gray-scale image of size 224×224 pixels, this would yield a huge number of states corresponding to the number given in Equation 4.1, if each pixels is a 8-bit number.

$$\text{Card}(\mathcal{S}) = 256^{244 \times 244}, \tag{4.1}$$

where $\text{Card}(\cdot)$ is the cardinality of a set. Instead of a table representation of the value function, one can use a function that approximates the desired value function. This function approximator is then trained using interactions between agent and environment. In applications where the states are complex representations of the environment, such as images [Sutton and Barto 1998, p. 193], the exact same state might not have been sensed by the agent before. Therefore, when the agent senses new unknown states, it should be able to generalize these new states to the states it has already sensed [Sutton and Barto 1998, p. 193]. Also, the table based methods are limited in their ability to generalize to states that has not been visited before, since each unique state has its own entry in a table.

In this chapter, the focus is on models in deep learning that provides function approximation, such as neural networks or convolutional neural networks. When deep learning models are used as function approximators in reinforcement learning it is called deep reinforcement learning [Arulkumaran et al. 2017]. This applies to both policy based methods as well as for value based methods, where neural networks are used to approximate a function f that approximates the state-value function,

$$f(s;\boldsymbol{\theta}) \approx V^{\pi}(s), \quad \forall s \in \mathcal{S} \tag{4.2}$$

where $f(s;\boldsymbol{\theta})$ is a function of state, s, with adjustable parameters $\boldsymbol{\theta}$, as in Chapter 3. More precisely, because the model is unknown, the state-value function is not

sufficient to determine a policy, as it involves the transition probabilities [Sutton and Barto 1998]. Instead the action-value function is approximated.

$$g(s, a; \boldsymbol{\theta}) \approx Q^{\pi}(s, a), \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s) \tag{4.3}$$

The goal is then to use interaction experience, or more formally sequences of (s_t, a_t, r_t, s_{t+1}) to train the parameters $\boldsymbol{\theta}$ to approximate the optimal action-value function. Instead of having the value function represented by a explicit representation, so the value function is instead characterized by parameters $\boldsymbol{\theta}$. This allows the agent to generalize from visited states to unvisited states, through this parametrization.

This chapter starts out by describing deep Q-networks which was one of the first successful implementations of deep reinforcement learning, trained to play Atari games [Mnih et al. 2015]. One disadvantage of representing the action-value function using a neural network is that it can not be guaranteed to converge to the optimal action-value function, compared to the table based method, where the action-value function can be shown to converge to the optimal one [Mnih et al. 2015]. Therefore a technique called experience replay is introduced next. Lastly an algorithm called Asynchronous Advantage Actor Critic algorithm, known as the A3C algorithm, is described.

4.1 Deep Q-Networks

A deep Q-network as introduced in [Mnih et al. 2015] is where a deep learning model is used to approximate the action-value function. A convolutional neural network was used as the function approximator for action-value function. The convolutional neural network is trained on images of the screen of an agent playing an Atari game, thereby learning to play the game directly from raw pixels without any prior knowledge of how the game is played.

Previously, reinforcement learning has been limited to domains with low-dimensional state spaces or domains where handcrafted features can be extracted from the data [Mnih et al. 2015]. The deep Q-network extract these features directly from the image by learning new abstract representations of the input data.

The deep Q-network algorithm presented in [Mnih et al. 2015] is seen in Algorithm 2. Reinforcement learning using a neural network as the function approximator for the action-value function is known to be unstable or diverge [Mnih et al. 2015], and therefore to try to stabilize the algorithm something called experience replay is used. Experience replay reduces the correlation of the sequence of actions used as training data. The experience acquired by the agent in the environment is stored in the experience replay memory $D_t = \{e_1, e_2, \ldots, e_t\}$, and not immediately used for training such as in online reinforcement learning, since this would result in training on highly correlated training data.

Experience replay is used to stabilize the training when a non-linear function is used to approximate the action-value function, such as a neural network [Mnih et al. 2016]. Training a reinforcement learning agent requires data of the transitions between agent and environment. In online reinforcement learning algorithms,

where the agent is learning side by side with experiencing the environment, each data sample is used to train the agent once immediately after it has been experienced, and then discarded. This way of training the agent works fine for the table based methods, since updating a Q-value only affects itself and not all the other Q-values, such as if a parametrization the action-value function is used. However, when the action-value function is represented by a non-linear function instead of a table the action-value function becomes unstable [Mnih et al. 2016].

Training an agent in an online manner results in strongly correlated transitions, and one way to try to avoid using correlated transitions as training data is to use what is called experience replay [Mnih et al. 2016]. Instead of throwing away each transition after each update, transitions are instead be stored in a buffer, called experience replay [Zhang and Sutton 2017]. Experience replay is a memory that stores the most recent transitions up to its capacity. After the buffer is filled, the oldest transitions in the memory are replaced by the new ones. The transitions are not used immediately after it has been experienced, but instead a mini-batch of transition are sampled from the experience replay memory and used to train the agent. This could be a uniform distributions over the experience replay memory if no transitions are preferred over others. In this way experience replay may avoid using strongly correlated transitions by sampling from a memory of transitions which stabilizes the training of an online reinforcement learning agent [Zhang and Sutton 2017].

Compared to online reinforcement learning additional memory is required for storing the transitions when using experience replay [Mnih et al. 2016]. As for the the table-based Q-learning algorithm in Section, the agent selects actions according to a parameter ϵ. If the agent chooses to exploit what it has already learned, the next action is selected by passing the current state through the action-value function, represented by a neural network.

4.2 Asynchronous Advantage Actor-Critic Algorithm

Until now, the focus has been on value based methods, where the policy is given implicit by the value function by choosing the actions according to a greedy policy. However, another class of methods exists that directly estimates the policy without the need of a value function. Instead of finding policies by estimating value functions and extracting a policy from that, one can instead directly optimize the parameters of a parametrized policy $\pi(s, a; \boldsymbol{\theta})$ [Sutton and Barto 1998]. Policy gradient methods do this by using a parametrization of the policy, denoted $\pi(s, a; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is the parameters, and optimize it with respect to the parameters $\boldsymbol{\theta}$. The idea is then to adjust $\boldsymbol{\theta}$ using optimization to converge to an optimal policy $\pi^*(s, a; \boldsymbol{\theta})$. Policy gradient methods are a general approach to finding optimal behaviour in an environment, but requires methods from function approximation. This could be a linear function, as described in [Sutton and Barto 1998], but the reason for first introducing these types of methods in this chapter is, that in this project policy gradient methods are limited to neural networks as being function approximators. This is therefore considered a deep reinforcement learning method,

Algorithm 3: Deep-Q-Network [Mnih et al. 2015]

Initialize experience replay memory D to size N
Initialize action-value function, Q, with random weights, θ.
Initialize target action-value function, $\hat{Q}$, with weights $\hat{\theta} = \theta$.
Repeat for a number of episodes.
  Initialize s_1
  Repeat until episode terminates
    Draw random number, k, uniform between $[0, 1]$.
    $$a_t = \begin{cases} \text{pick random action } a_t \in \mathcal{A}(s) & \text{if } k \leq \epsilon \\ \max_a Q\left(\phi(s_t, a); \theta\right) & \text{if } k > \epsilon \end{cases}$$
    Execute action, a_t in emulator and observe reward r_t and state, s_{t+1}.
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$.
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D.
    Sample a random mini-batch of transitions $(\phi_j, a_j, r_j, \phi_{t+1})$ from D.
    $$y_i = \begin{cases} r_j & \text{if episode terminates at step } j \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{t+1}, a'; \theta\right) & \text{otherwise} \end{cases}$$
    Perform gradient decent step on $\left(y_i - Q(\phi_j, a_j, \phi_{j+1})\right)^2$
      with respect to the parameters θ.
    Every C number of steps set $\hat{Q} = Q$.

because the policy is approximated by a model, such as a neural network, from deep learning.

The Asynchronous Advantage Actor-Critic algorithm, named A3C [Mnih et al. 2016], for deep reinforcement learning uses such a parametrized policy. The A3C algorithm is a type of method called actor-critic. The actor-critic method is a combination of value based and policy based methods, that uses the advantages of both these methods [Grondman et al. 2012]. It is among the most popular algorithms in the area of reinforcement learning and specially in deep reinforcement learning [Grondman et al. 2012].

Instead of experience replay, the A3C algorithm utilizes multiple agents working in parallel to reduce non-stationarity [Mnih et al. 2016]. Each of these agents operates in their own instance of the environment and each agent is therefore able to execute its actions independent of the other agents in an asynchronously manner. Since the agents will experience a variety of different states, the agents' data will be decorrelated into a more stationary process. In addition this to, the A3C algorithm is able to benefit from the parallel computing that applies to CPUs, instead of relying on specialized hardware such as GPUs.

Since the actor-critic method is an on-policy algorithm, experience replay cannot be used to stabilize the algorithm, since experience replay is limited to off-policy algorithms [Mnih et al. 2016]. Instead another paradigm is presented in [Mnih et al. 2016] which provides a way of stabilizing the algorithm for on-policy deep reinforcement learning, such as the actor-critic method.

In actor-critic method a policy and a value function is learned, represented by

the actor and critic, respectively. The actor learns about the policy and how to act to achieve the most reward, while the critic learns to criticize the actor based on a value function [Sutton and Barto 1998]. Actor-critic methods, A3C utilizes a parametrized policy $\pi(a_t|s_t;\boldsymbol{\theta})$ and a parametrized value function $V(s_t;\boldsymbol{\theta})$ to solve the reinforcement learning problem. Just like the policy based reinforcement learning methods, the actor-critic method uses a parametrization of the policy, $\pi(a|s;\boldsymbol{\theta})$ and update the parameters of the policy to. In the standard REINFORCE algorithm, which is a policy gradient method, the parameters are updated in the direction [Mnih et al. 2016]

$$\bigtriangledown_\theta \log \pi(a_t|s_t;\boldsymbol{\theta})G_t \tag{4.4}$$

which is an unbiased estimate of

$$\bigtriangledown_\theta \mathbb{E}\left[G_t\right]. \tag{4.5}$$

The variance of this estimate can be reduced by subtracting a baseline function, $b_t(s_t)$ from the return, G_t, resulting in the gradient estimate

$$\bigtriangledown_\theta \log \pi(a_t|s_t;\boldsymbol{\theta})(R_t - b_t(s)). \tag{4.6}$$

If this baseline function is an approximation of a value function, $R_t - b_t(s_t)$ can be interpreted as the advantage of taking action a_t in state s_t. The advantage function can be expressed as

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t).$$

The A3C algorithm is seen in Algorithm 4, which is adapted from [Mnih et al. 2016].

4.3 Grid World Navigation

Deep reinforcement learning is in this section applied to the grid world environment described in Appendix B. The two algorithms applied is deep Q-networks and the A3C algorithm, described in Section 4.1 and 4.2, respectively.

4.3.1 Deep Q-Networks

In this section, Algorithm 3 is applied to the grid world environment to improve the behaviour an agent having to find one target position in the grid world environment. Since a state is represented by a vector, $s \in \mathbb{R}^2$, a feed-forward neural network is used instead of a convolutional neural network to model the action-value function, $Q*(s,a)$. The four actions available for the grid world environment results in the neural network having four output neurons, and can thus be considered as a mapping $g : \mathbb{R}^2 \to \mathbb{R}^4$. The architecture of the neural network is seen in Figure 4.1. It has four hidden layers with 100 neurons in each of these layers. There is no specific reason for this architecture, but with its about 30,000 parameters it is expected to be enough parameters to approximate $Q^*(s,a) \approx Q(s,a,\boldsymbol{\theta})$.

Algorithm 4: Asynchronous Advantage Actor-Critic algorithm for one actor-learner thread. The algorithm is adapted from [Mnih et al. 2016].

Global shared parameters $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_v$ and global shared counter $T = 0$
Specific thread parameters $\boldsymbol{\theta}'$ and $\boldsymbol{\theta}'_v$
Initialize thread step counter $t \leftarrow 1$
repeat until $T > T_{max}$
  Reset gradients $d\boldsymbol{\theta} \leftarrow 0$ and $d\boldsymbol{\theta}_v \leftarrow 0$
  Synchronize thread parameters $\boldsymbol{\theta}' = \boldsymbol{\theta}$ and $\boldsymbol{\theta}'_v = \boldsymbol{\theta}_v$
  $t_{start} = t$
  Get state s_t
  repeat until terminal state s_t **or** $t - t_{start} = t_{max}$
    Perform a_t according to policy $\pi(a_t|s_t; \boldsymbol{\theta}')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  end
  $R = \begin{cases} 0 & \text{if } s_t \text{ is terminal state} \\ V(s_t, \boldsymbol{\theta}'_v) & \text{if } s_t \end{cases}$
  for indicies $i \in$
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients with respect to parameter $\boldsymbol{\theta}'$
      $d\boldsymbol{\theta}' \leftarrow d\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}'} \log \pi(a_i|s_i; \boldsymbol{\theta}')(R - V(s_i; \boldsymbol{\theta}'_v))$
    Accumulate gradients with respect to parameter $\boldsymbol{\theta}'_v$
      $d\boldsymbol{\theta}'_v \leftarrow d\boldsymbol{\theta}_v + \partial(R - V(s_i; \boldsymbol{\theta}'_v))^2 / \partial\boldsymbol{\theta}'_v$
  end
  Perform asynchronous update of $\boldsymbol{\theta}$ using $d\boldsymbol{\theta}$ and of $\boldsymbol{\theta}_v$ using $d\boldsymbol{\theta}_v$
end

The two neural networks are initialized with the same architecture and parameters, one for the action-value function, Q, and the other called the target action-value function, $\hat{Q}$. The training process is repeated until the agent has reached the target position 1500 times. The actions the agent performs are picked according to an ϵ-soft policy, where non-random actions are chosen by passing the current state through the network Q and picking the action with the highest Q-value. This done in order to keep exploring the state space. The experience replay buffer is set to a size of 10,000.

The code for running the deep Q-network on the grid world environment example can be found in Appendix A. For a plot that shows the error after each completed episode compared to the solution from dynamic programming, see Appendix D.

In Figure 4.2 and 4.3, the results for different numbers of episodes used for training deep Q-network is shown. It is seen that applying deep Q-networks can indeed by used to improve the agents behavior towards a policy, that finds the target position. Since the deep Q-network estimates the solution to the Bellman

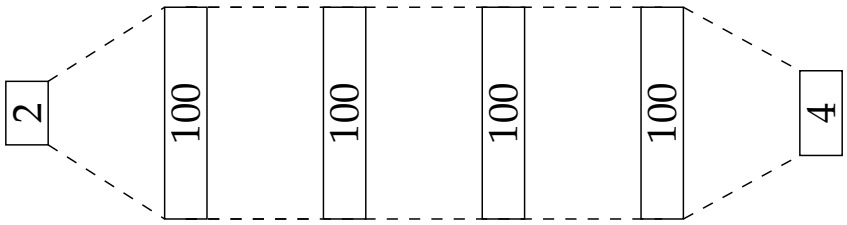
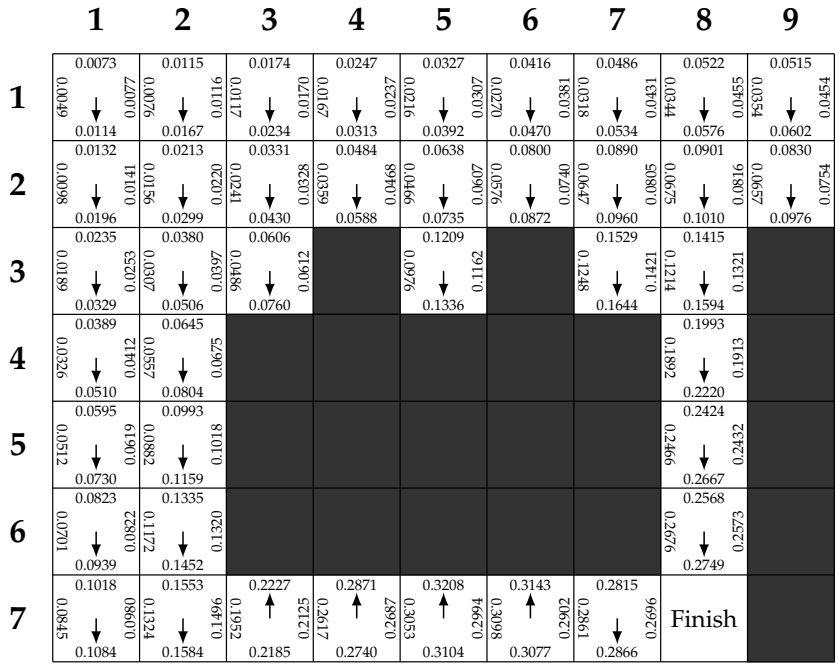


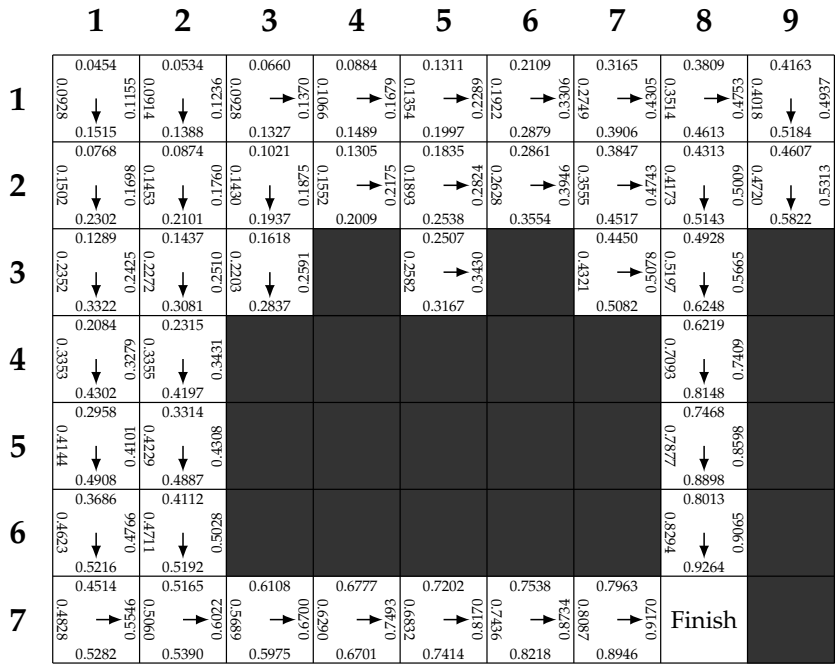
Figure 4.1: Architecture of the deep Q-network used for the grid world environment.

equation that estimates of a Q-value in a states is based on. For each action the agent perform, the model is trained on 20 data samples sampled from the experience memory.

It is also seen that after the deep Q-network is trained on 1,500 episodes only a single state has a policy that is not optimal. This is the state $s_{(3,3)}$, where instead of moving one step closer to the target, the policy in this state picks an action that bumps into the wall. This could be explained by the amount of times this state is used as training data for the deep Q-network. One reason for this could be that the state is only visited a few number of times cause by the lag of exploration of the environment due to the value of ϵ. Another reason for this could be that this state is not sampled as many times from the experience replay buffer.

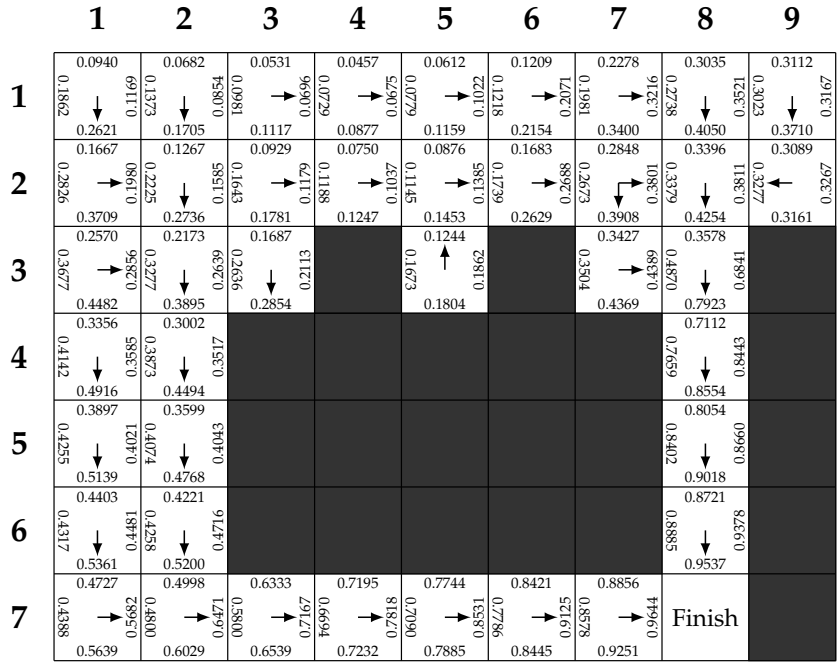


(a) Initial $Q(s, a)$.

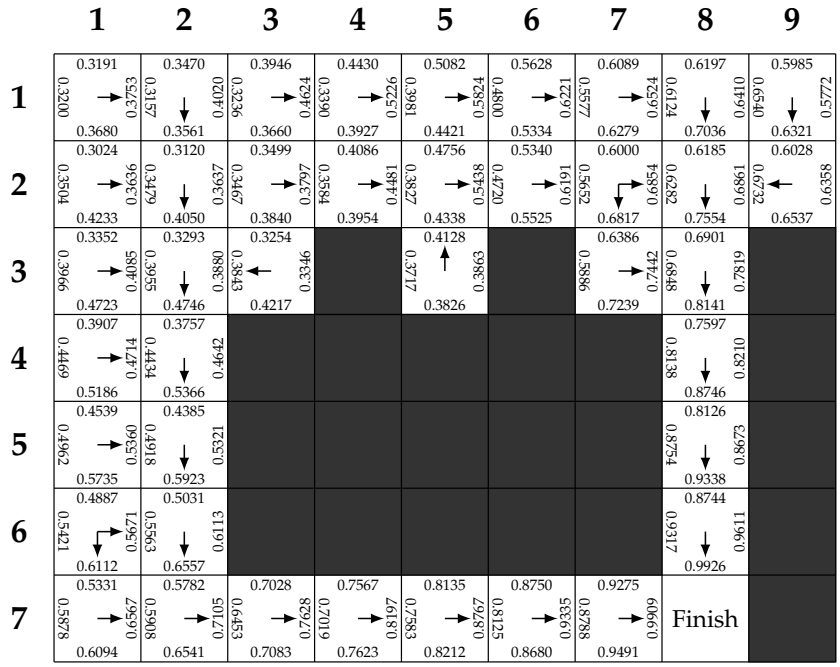


(b) $Q(s, a)$ after 100 episodes.

Figure 4.2: Results showing the convergence of applying a deep Q-network to the grid world environment. he four values at each line of each square represents the Q-value of going in that direction. The arrow in each square represents a greedy policy.



(a) $Q(s, a, \boldsymbol{\theta})$ after 500 episodes.



(b) $Q(s, a, \boldsymbol{\theta})$ after 1500 episodes.

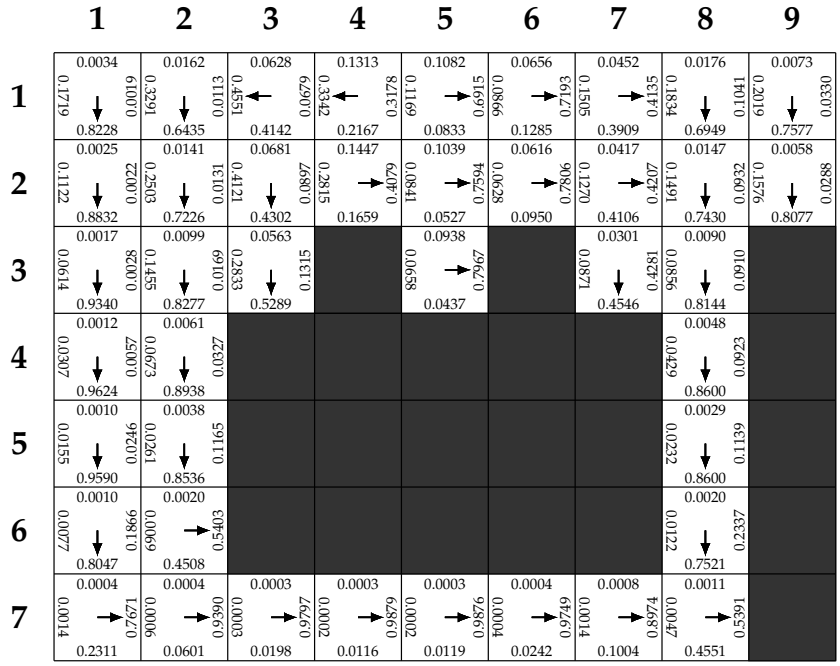
Figure 4.3: Results showing the convergence of applying a deep Q-network to the grid world environment. he four values at each line of each square represents the Q-value of going in that direction. The arrow in each square represents a greedy policy.

4.3.2 A3C Algorithm

In this section, an implementation of the A3C algorithm, found in is applied to solve the navigation problem in the grid world environment for a single target. The implementation of the A3C algorithm [1] used throughout this project is an implementation found in [Zhu et al. 2017]. This implementation is then modified to take a two dimensional state represesntation instead of images.

Since the A3C algorithm uses both a a parametrized policy and value function, both of these function are inspected. The algorithm is trained on four thread running in parallel. Note here that the output of the policy is a probability distribution and not Q-values, which means that the values for deep Q-networks cannot be compared to the ones illustrated for deep Q-networks. However, a greedy policy for both the figures can be compared, where the greedy policy picks the action with the highest probability in each state. The amount of data used to train the two model cannot be compared since the deep Q-network is trained for a number of episodes using experience replay, and the A3C algorithm runs for a specific number of samples. The A3C implementation converges for a single states both for the policy and for the value function.

[1] https://github.com/zfw1226/icra2017-visual-navigation

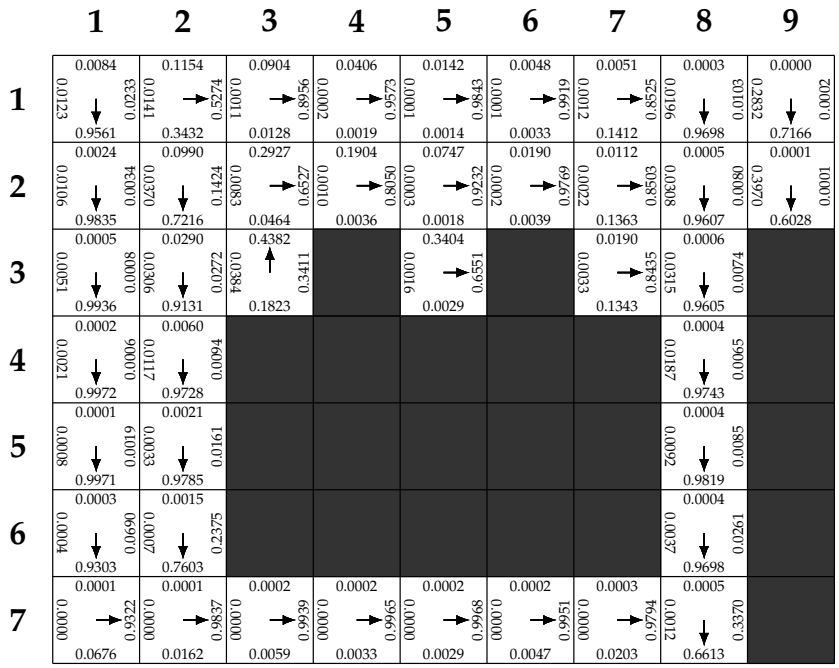


(a) Policy after 50,000 samples

	1	2	3	4	5	6	7	8	9
1	0.1060	0.1206	0.1065	0.0957	0.1212	0.2991	0.4282	0.4282	0.1901
2	0.1494	0.1509	0.0298	-0.0869	-0.1102	0.2333	0.4792	0.5079	0.2719
3	0.2186	0.2368	0.0601		-0.3352		0.6307	0.7143	
4	0.2775	0.3065						0.8581	
5	0.3282	0.3910						0.9322	
6	0.4050	0.4855						0.9775	
7	0.5131	0.6161	0.7253	0.7991	0.8760	0.9477	0.9958	1.0129	

(b) Value function after 50,000 samples

Figure 4.4: Convergence of the model in [Zhu et al. 2016] for the grid world problems after 50,000 steps.



(a) Policy after 150,000 samples

	1	2	3	4	5	6	7	8	9
1	0.1365	0.1868	0.2549	0.3317	0.3894	0.4474	0.5032	0.6106	0.4763
2	0.1661	0.1896	0.2007	0.2858	0.3920	0.4949	0.5783	0.6926	0.5294
3	0.2511	0.2425	0.1453		0.1596		0.6717	0.7879	
4	0.3090	0.3614						0.8634	
5	0.3792	0.4567						0.9335	
6	0.4579	0.5286						1.0134	
7	0.5357	0.6265	0.7132	0.7884	0.8676	0.9454	1.0107	1.0301	

(b) Value function after 150,000 samples

Figure 4.5: Convergence of the model in [Zhu et al. 2016] for the grid world problems after 150,000 steps.

Chapter 5
Target-Driven Visual Indoor Navigation

In this chapter the problem changes from a two-dimensional state representation of the grid world environment to a new state representation in terms of images from an indoor environment. Navigation in an indoor environment is a very common ability robots need to posses, if they have to interact in the same environments as humans. The environment could for example be a house where the robot should be able to navigate around in the different rooms inside the house.

In the grid world navigation problem from Section 2.4 the problem was to improve the behaviour by estimating an action-value function and then extracting a greedy policy from this. The grid world environment could be considered as a simplification of a room in a house. An agent in this environment is learned to find target positions in the room by using a two-dimensional position vector as the state representation. Instead of using a position vector, the robot could be equipped with cameras that allows the agent to get images of the room it has to navigate around inside. In this way the agent is trained to find target locations using a visual representation of the positions inside the room.

Therefore, using images as state representation will affect the state representation by turning it into a more complex representation, than just a position in a grid as used in Section 2.4 and Section 4.3. Furthermore, since the states are not known a model of the underlying Markov decision process is known neither. This means that knowledge of what states transition to what states is only gained through experience of the interaction between agent and environment.

Ideally, the data in terms of samples including a state, an action and a reward, would come from robot experiencing a physical room, because the finally goal would be to test the algorithm in a real world scenario. Acquiring the data in this way also has a number of disadvantages such as being very time consuming [Kolve et al. 2017]. The robot used to acquire data from a real world scenario is limited in the way that before a new image can be provided to the agent, the robot first has to perform the action of moving to the next position. This could take a lot of time, if the robot has to perform a lot of such actions.

To get an idea of how time consuming the process of acquiring data from a real world scenario is, a little example is provided. In [Zhu et al. 2016] the algorithm

is trained on 100 million samples. This is done in parallel on 100 threads, which means that for each target approximately 1 million samples are used. If the robot can perform an action in one second, training the algorithm on a single target would take about 11 days. In addition to this, each time the robot finishes an episode, the robot should be moved to a new starting position. Acquiring the data using simulations of a 3D environment could speed up this process.

Another disadvantage is that during the data acquisition process the robot might bump into things resulting in damage to objects in the room or damage to parts of the robot [Kolve et al. 2017]. The last disadvantage is the difficulty of finding the rooms in which the data acquisition should be conducted. This would again require 20 different rooms. During 11 days of data acquisition the room might be unusable to the occupants of the house, because the model in [Zhu et al. 2016] is trained without these dynamic factors of the environment. This would add further complexity of the environment in which the robot is trained to navigate.

To avoid the problems just discussed, a simulation framework is used to acquire data to train the model. However, simulated data cannot replace data acquired by a robot in a real environment, because the level of details might be reduced in a 3D model of the same environment [Kolve et al. 2017].

5.1 Simulation Framework

In this section the simulation framework for acquiring the data used to train the deep reinforcement learning model in [Zhu et al. 2016] is described. This framework, called AI2-THOR [Kolve et al. 2017], is also the one used in this project. For a quick introduction of how to use and install the framework, see the AI2-THOR tutorial [1]. This provides a Python library for interacting with the Unity 3D game engine with 3D modelled environments. 3D models of multiple types of rooms are provided including the functionality of moving around in these rooms. The different types of rooms are

Kitchen 30 different scenes.

Living room 30 different scenes.

Bathroom 30 different scenes.

Bedrooms 30 different scenes.

Some examples of scenes are seen in Figure 5.1. AI2-THOR provides the possibility of a discrete and a contioious action space, but this project is limited to a discrete action space. In total there are eight different actions used to navigate the agent around in the environments.

- Move forward
- Move backwards

[1] http://ai2thor.allenai.org/tutorials/installation

- Move left
- Move right
- Rotate left, 90 degrees
- Rotate right, 90 degrees
- Look up, 30 degrees
- Look down, 30 degrees

Additionally, AI2-THOR provides actions that allow the agent to interact with objects, but these are ignored, as the goal of this project is to navigate to target locations only, and not to interact with the things it finds in these locations.

5.2 State Representation using Images

In this section the state representation using images is described. The agent in the AI2-THOR framework provides visual inputs of the environment a bit like the way in which a human would visual sense the room. The agent in the AI2-THOR framework is equipped with a single camera, which allows the agent to get visual inputs of the agent in a single direction. The AI2-THOR framework implements the ability for the user to vary the size of the field of view for the agent. Increasing the width of the images has the effect of increasing the view angle of the agent, hence more information about the environment is gained in both sides of an image The effect of increase in width of the images is seen in Figure 5.2.

In [Zhu et al. 2016] images of a size of $224 \times 224 \times 3$ are used, which represents the height, width, and channels, respectively. A channel is used for each of the three colors red, green, and blue. If the size of the images is increased this results in an exponential growth in the number of pixels in the image which would correspond to an increase in the complexity of the state representation. On the other hand, this also means an increase in the details in an image. Throughout this report the size of an image provided by the AI2-THOR framework is kept fixed to a size of $224 \times 224 \times 3$.

Figure 5.1: Six different scenes in the THOR simulation framework with corresponding scene name

Figure 5.2: Illustration of the increase in view angle when varying the width of the images.

5.3 Target-Driven Navigation

In order to make deep reinforcement learning to indoor navigation, the agent has to learn to find multiple locations rather than just a single one. In the grid world environment only a single target was used to prove that these methods could be used for a simple navigation task.

Using either the approach from Section 4.1 or 4.1 to find a single target location, the location of the target is incorporated into the parameters of the model, because all the parameters of the model would be adjusted towards a policy that only finds this specific target location. Using this approach would require a new model for each of the target positions in each different room. This would be impractical because a lot of different models need to be stored and a new model should be loaded every time a new target should be found. In this way the benefits of neural networks being universal approximators are not really used to its fully potential.

An advantage including the target location as input to the network is that the same network can be trained to locate multiple target locations. The classical reinforcement learning algorithm presented in Section 2.4 were only trained to find a single target. If these methods should be used to find multiple locations, a table-based representation of the value function is required for each target the agent has to find. Instead, the strength of using a function approximator to represent either the policy or the value function is utilized.

Before images are fed to the algorithm each image is reduced from a size of $224 \times 224 \times 3$ to a 2048-dimensional vector by feeding it through the ResNet-50 model [He et al. 2015], where the last softmax layer is removed, producing a 2048-dimensional output.

A Figure of the network architecture is seen in Figure 5.3. The algorithm is the type of A3C, as described in section 4.2. In [Zhu et al. 2016] a model, that takes both the current state and a state representation of the target, is proposed. This is called target-driven navigation as the target is also used as the input to the model. The same model could then be trained to find multiple target positions in a room, by also including the target as an input to the model. Therefore in [Zhu et al. 2016] a target-driven network architecture is presented that takes an image of both the current location and the target location. This is achieved by using a network architecture called a deep siamese actor-critic model, which allows the generalization to new target objects without the need to retrained the model the new target objects. This architecture is used to allow the agent with an understanding or the relative spatial position between the current location and the target position is to project them into the same embedding space [Zhu et al. 2016]. These embedding spaces are then concatenated into a 1024-dimensional vector which is then fused into a joint representation using a fully connected layer.

A deep siamese network architecture is used to learn the spatial relation between the current location and the target location. Features are extracted from both the current state and the target observation. This is done using a pre-trained convolutional neural network, called ResNet-50 [He et al. 2015]. The four 2048-dimensional vectors are concatenated into a single 8192-dimensional vector. This

vector is used as input for a fully-connected layer of 512 neurons, resulting in a weight matrix of size 512×8192 and a bias vector of 512. The same procedure is repeated for the state of the target location, but instead of having individual parameters for current and target location, these parameters are shared. The 512-dimensional output of both of these streams are concatenated into a single 1024-dimensional vector which is then projected into a 512-dimensional fusion space by a fully-connected layer. This joint representation is then used as input to scene-specific feed-forward neural networks. A scene-specific network consists of one hidden layer of 512 neurons. This hidden layer is connected to an output layer with 5 output neurons; four neurons for the policy and one neuron for the value function. Since a probability distribution over the four possible actions is wanted, a softmax function is used to map the four policy neurons in the output layer into probabilities.

In [Zhu et al. 2016] a state is represented by not only the current camera image available to the agent, but also the images from the three most recent movements the agent has performed. The motivation behind this is to utilize the most recent movements of the agent to find the target object.

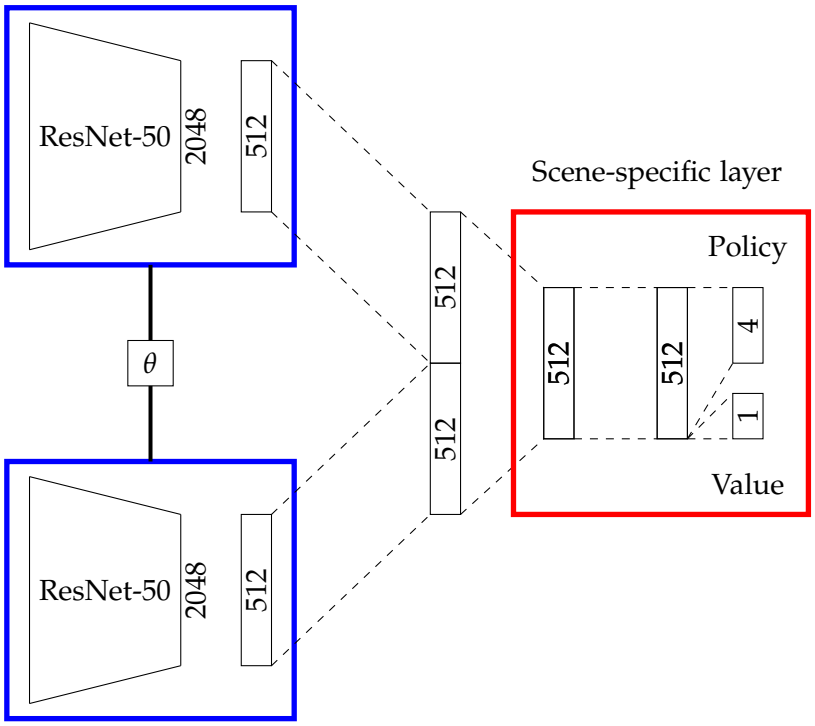


Figure 5.3: Network architecture of target-driven visual navigation adapted from [Zhu et al. 2016].

Chapter 6
Results

In this chapter, the results for target-driven visual navigation is presented. In these experiments the simple grid world environment is extended by adding multiple target positions the agent should be trained to locate. The complexity of the navigation task is increased by adding a total of four different target positions inside the same room in the AI2-THOR environment. The positions of the four targets in the grid world environment is seen in Figure 6.1.

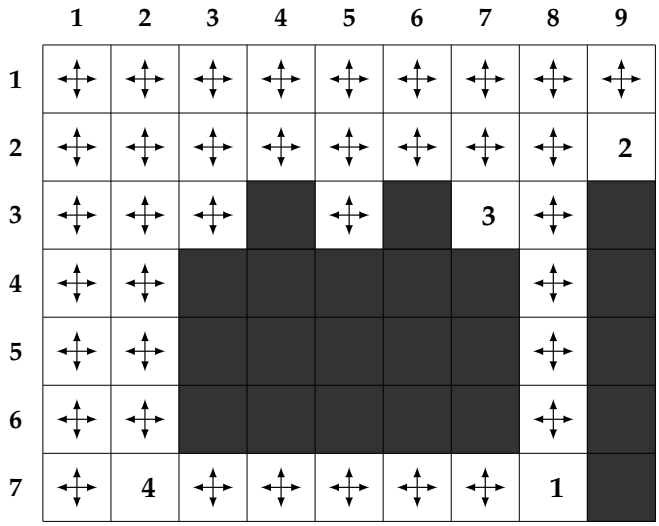


Figure 6.1: Grid world navigation example with four different target positions. The target positions are named by the numbers from one to four.

In this chapter, different experiments are presented. First, the algorithm in [Zhu et al. 2016] is modified to run on the grid world example there positions states are used. The next experiment uses images as states, where a new state representaion is proposed, that simplifies the comparison between results from the grid world environment and results for models trained with images as input. During the experiments presented in this section, the hyperparameters such as the learning rate are kept fixed for all the experiments. The learning rate is the same as used in **??**. The implementation of the algorithm described in Section 5.3 is found in [Zhu et al. 2017]. Since this implementation uses pre-generated data files, the code is modified to interact with the AI2-THOR environment during training.

6.1 Target-Driven Grid World Navigation

The algorithms until now have been applied to a simple grid world problem, where the agent is trained to find a single target location. In this section, this is extended to training the agent to find a total of four different targets. The four target positions are seen in Figure 6.1. The algorithm used to train the agent is the one described in Section 5.3. However, instead of using images as input, a two-dimensional state representation is used for both the current position of the agent in the environment and the target location the agent has to find. The changes of the network architecture is therefore only found in the input layer and the next layer, where the two-dimensional state is not passed through the ResNet50 model, but directly concatenated to a four-dimensional input.

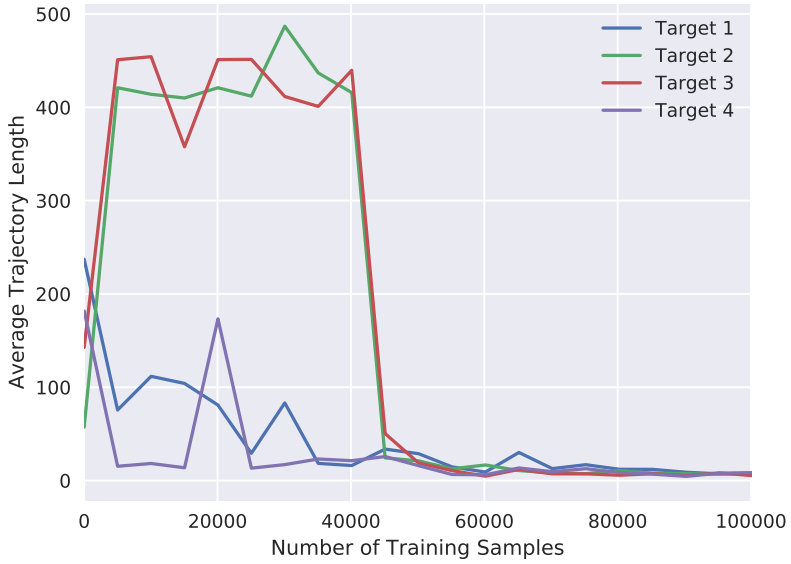
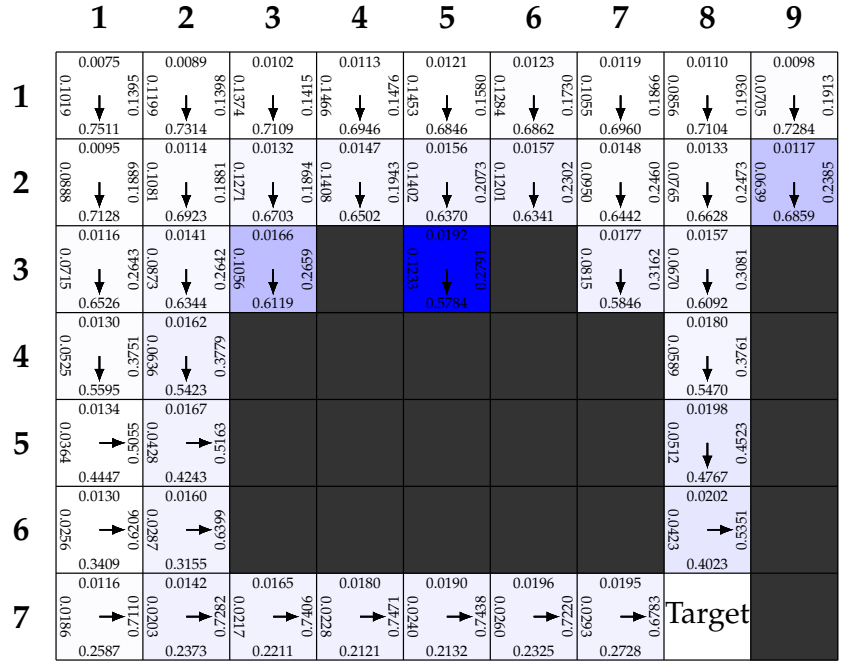


Figure 6.2: Figure showing the average trajectory length evaluated for every 5,000 for each target.

By examining Figure 6.2 target number one and four is seen to improve faster than the other two targets. Therefore a study of the training data is analysed to answer this question. This is first analysed for target 1, which is seen to converge better than target 2. To be able to answer this question both the policy and value function is examined together with an distribution of what states the agent has discover the previous 1,000 steps. In Figure 6.3 both the policy and value function after the algorithm is trained on 30,000 samples. The policy is illustrated together with a distribution of what states are visited the most during the previous 1,000 training samples. A dark blue color means that a state is visited many times, while a white color means that a state is only visited a few times. The policy seems to already converge for the states close to the target position. The same is observed for the value function, where the states nearest the target position almost share the same values as the solution from dynamic programming in Section 2.4.1 after

being trained for 30,000 samples.



(a) Stochastic policy for target 1 after being trained for 30,000 samples.

	1	2	3	4	5	6	7	8	9
1	-0.4175	-0.6015	-0.8087	-0.9549	-1.0017	-0.8087	-0.4291	-0.0366	0.2952
2	-0.3065	-0.5082	-0.7626	-0.9830	-1.0280	-0.7602	-0.2624	0.1925	0.5327
3	-0.1696	-0.3380	-0.6001		-0.9571		-0.0067	0.4659	
4	-0.0035	-0.1235						0.7258	
5	0.1273	0.0616						0.9173	
6	0.2338	0.2501						1.0709	
7	0.3221	0.4084	0.5301	0.6708	0.8170	0.9550	1.0730	1.1818	

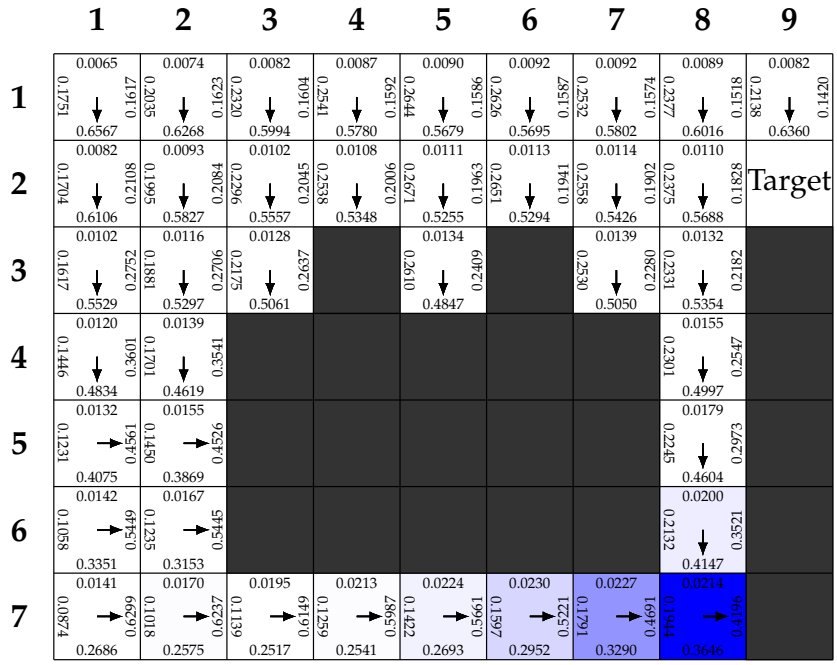
(b) Value function for target 1 after being trained for 30,000 samples.

Figure 6.3: Policy and value function for target 1 after being trained for 30,000 samples.

In Figure 6.5 the same is seen for target 2, which did not improve as fast as target 1. The policy for target 2 is seen to move the agent in the direction of target 1 and not target 2 as expected. This affects the distribution of states the agent visits. Instead of moving towards target 2, the agent gets stuck trying to find target 1, which means that the majority of the states visited by the agent is close to target 1. As a result of this, the agent is not trained on data where the target is found. When examining the value function, it is seen that the values for all the states are wrong compared to the value function calculated by dynamic programming.

As observed in Figure 6.2 after the agent is trained for 60,000 samples the policy states to improve. Already after an additional 30,000 training samples, the policy is seen to move the agent in the direction of taget 2 rather than in the direction of target 1. Therefore, the previous 1,000 training samples is also seen to be distributed around the right target. From the additional 30,000 training samples it is seen that the value function is now closer to the true value function.

In machine learning, it is typically important to test the models on a test data set, which has not been used for training. By doing so, the model's ability to generalize to data it has never seen before can be evaluated. This is done by visualizing the policy for the two new targets seen in Figure 6.7. To verify this generalization, two new targets in each both ends of the room is used.

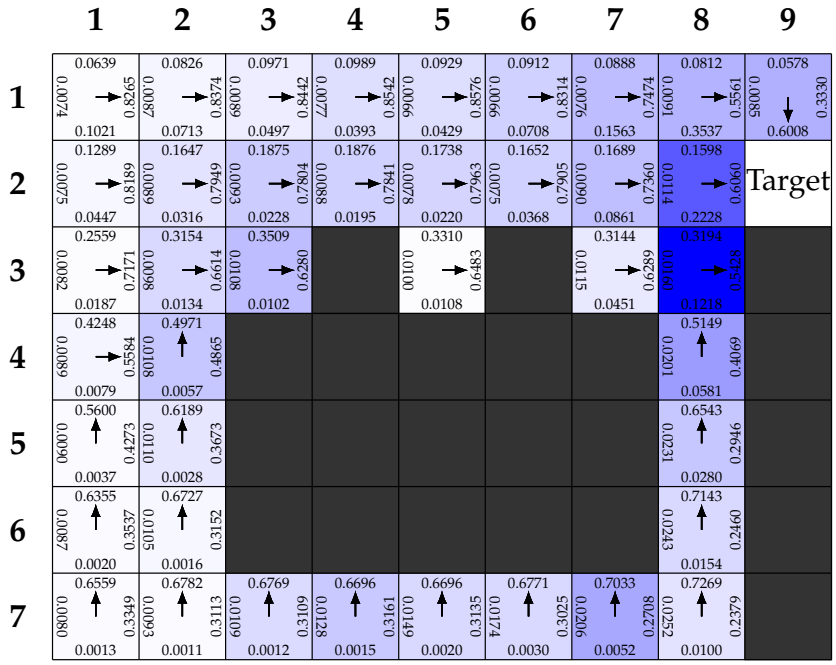


(a) Stochastic policy for target 2 after being trained for 30,000 samples.

	1	2	3	4	5	6	7	8	9
1	-4.4978	-4.6945	-4.9156	-5.1037	-5.2256	-5.2723	-5.1968	-5.0489	-4.8149
2	-4.5767	-4.7623	-4.9951	-5.1957	-5.3201	-5.3186	-5.2172	-5.0256	-4.7695
3	-4.6778	-4.8278	-5.0388		-5.3804		-5.1864	-4.9572	
4	-4.7950	-4.9284						-4.9215	
5	-4.9141	-5.0026						-4.9038	
6	-5.0050	-5.0590						-4.8991	
7	-5.0618	-5.0878	-5.0451	-4.9857	-4.9384	-4.8872	-4.8842	-4.8718	

(b) Value function for target 2 after being trained for 30,000 samples.

Figure 6.4: Policy and value function for target 2 after being trained for 30,000 samples.

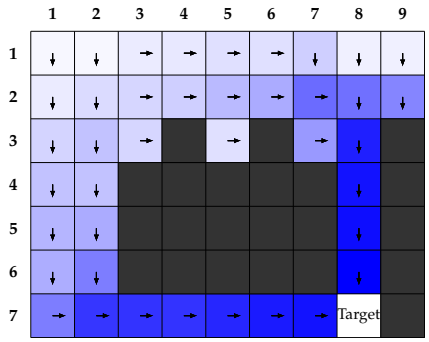


(a) Stochastic policy for target 2 after being trained for 60,000 samples.

	1	2	3	4	5	6	7	8	9
1	0.0890	0.1195	0.1980	0.2788	0.3635	0.4107	0.4280	0.4121	0.3601
2	0.0952	0.1260	0.1519	0.2212	0.3336	0.4156	0.4510	0.4236	0.3616
3	0.0163	0.0249	0.0141		0.1993		0.4069	0.3709	
4	-0.0992	-0.1388						0.2803	
5	-0.2563	-0.3227						0.1631	
6	-0.3972	-0.4759						0.0510	
7	-0.5015	-0.5731	-0.6059	-0.5613	-0.4622	-0.3571	-0.2149	-0.0696	

(b) Value function for target 2 after being trained for 60,000 samples.

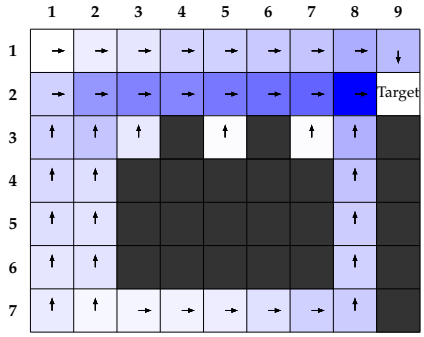
Figure 6.5: Policy and value function for target 2 after being trained for 60,000 samples.



(a) Policy for target 1.

	1	2	3	4	5	6	7	8	9
1	0.13	0.13	0.13	0.22	0.35	0.46	0.54	0.54	0.34
2	0.16	0.14	0.11	0.15	0.28	0.47	0.59	0.62	0.37
3	0.20	0.18	0.09		0.11		0.68	0.76	
4	0.28	0.29						0.87	
5	0.34	0.39						0.92	
6	0.42	0.48						0.99	
7	0.50	0.60	0.68	0.76	0.85	0.92	0.99	1.04	

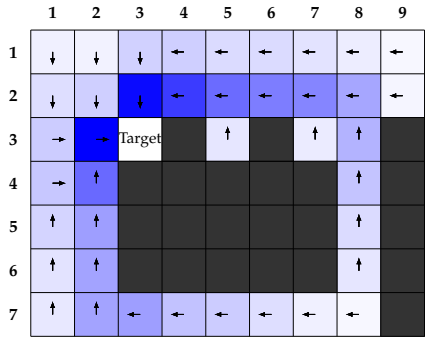
(b) Value function for target 1.



(c) Policy for target 2.

	1	2	3	4	5	6	7	8	9
1	0.43	0.48	0.57	0.64	0.71	0.79	0.85	0.93	1.00
2	0.41	0.51	0.59	0.67	0.76	0.84	0.91	0.99	1.01
3	0.35	0.43	0.49		0.67		0.89	0.92	
4	0.29	0.34						0.83	
5	0.23	0.27						0.77	
6	0.19	0.19						0.67	
7	0.12	0.13	0.15	0.22	0.33	0.41	0.48	0.57	

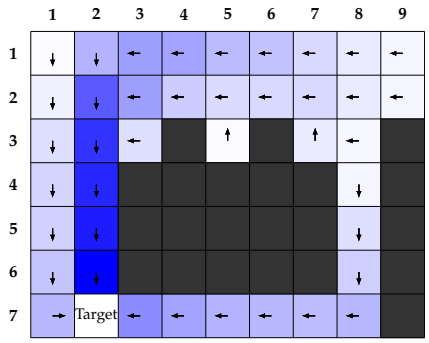
(d) Value function for target.



(e) Policy for target 3.

	1	2	3	4	5	6	7	8	9
1	0.7871	0.86	0.89	0.81	0.74	0.69	0.61	0.54	0.43
2	0.85	0.93	0.99	0.90	0.81	0.73	0.65	0.54	0.40
3	0.90	0.99	1.03		0.81		0.60	0.47	
4	0.82	0.91						0.39	
5	0.76	0.84						0.30	
6	0.71	0.74						0.22	
7	0.65	0.64	0.54	0.44	0.38	0.28	0.22	0.19	

(f) Value function for target 3.

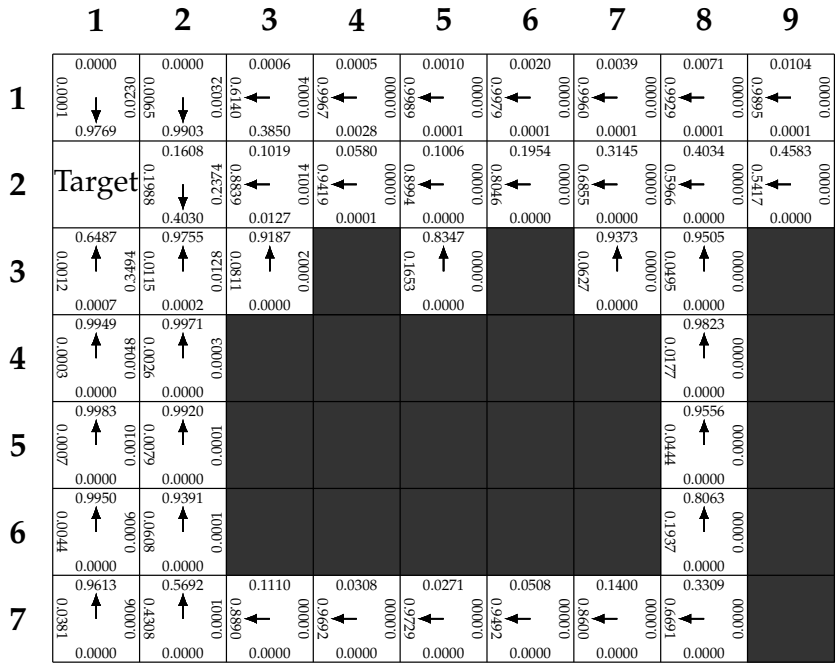


(g) Policy for target 4.

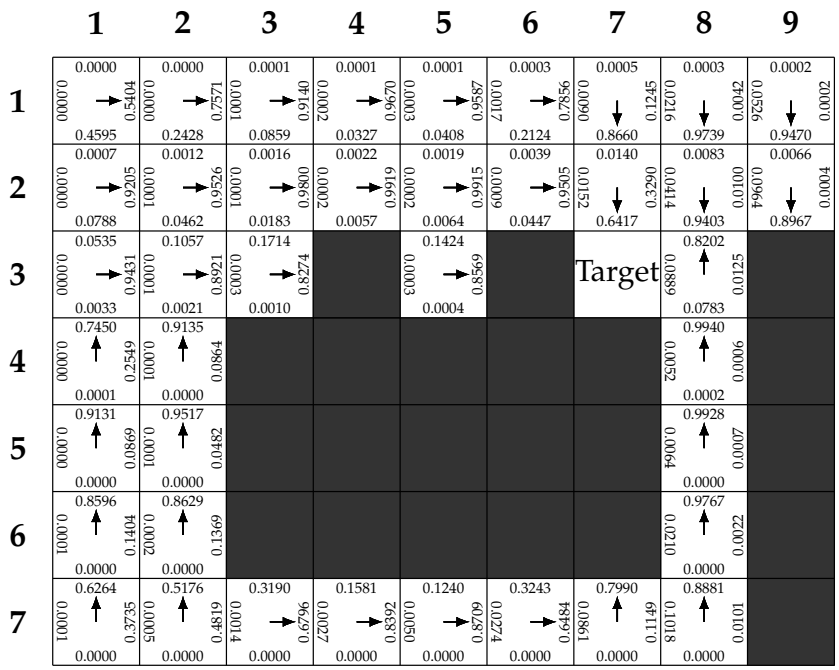
	1	2	3	4	5	6	7	8	9
1	0.58	0.59	0.53	0.45	0.38	0.33	0.27	0.19	0.06
2	0.64	0.66	0.58	0.47	0.37	0.29	0.22	0.12	0.01
3	0.70	0.76	0.64		0.24		0.15	0.12	
4	0.77	0.84						0.27	
5	0.85	0.92						0.39	
6	0.90	0.99						0.47	
7	0.98	1.01	1.00	0.92	0.85	0.77	0.67	0.59	

(h) Value function for target 4.

Figure 6.6: The left column shows the policy for each target position, and the right column shows the value function at the same time step. These are the results after the agent is trained on 300,000 samples.



(a) Policy for the new first target.



(b) Policy for the new second target.

Figure 6.7: The left column shows the policy at different number of steps used to train the network. The right column shows the value function at the same time step.

6.2 Target-Driven Visual Navigation

In this section the same model architecture as in [Zhu et al. 2016] is used, with four images of both current position and target position is used. In this experiment instead of training the algorithm using images from previous movements, another state representation is used, where an image in each of four directions are used.

As seen from Figure 6.8, only the policy for target 2 seems to converge, and in the end it is event seen to become worse. For such high trajectory lengths, it is expected that the agent moves around in loops in order to move so many states without finding the target. This can also be verified by looking at Figure 6.9.

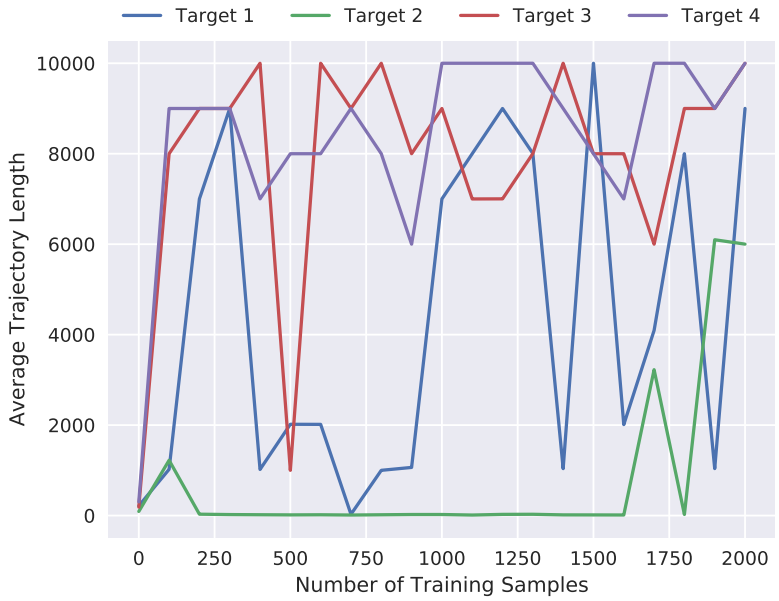


Figure 6.8: Average trajectory length using image states. The number of samples used are listed in 1,000s.

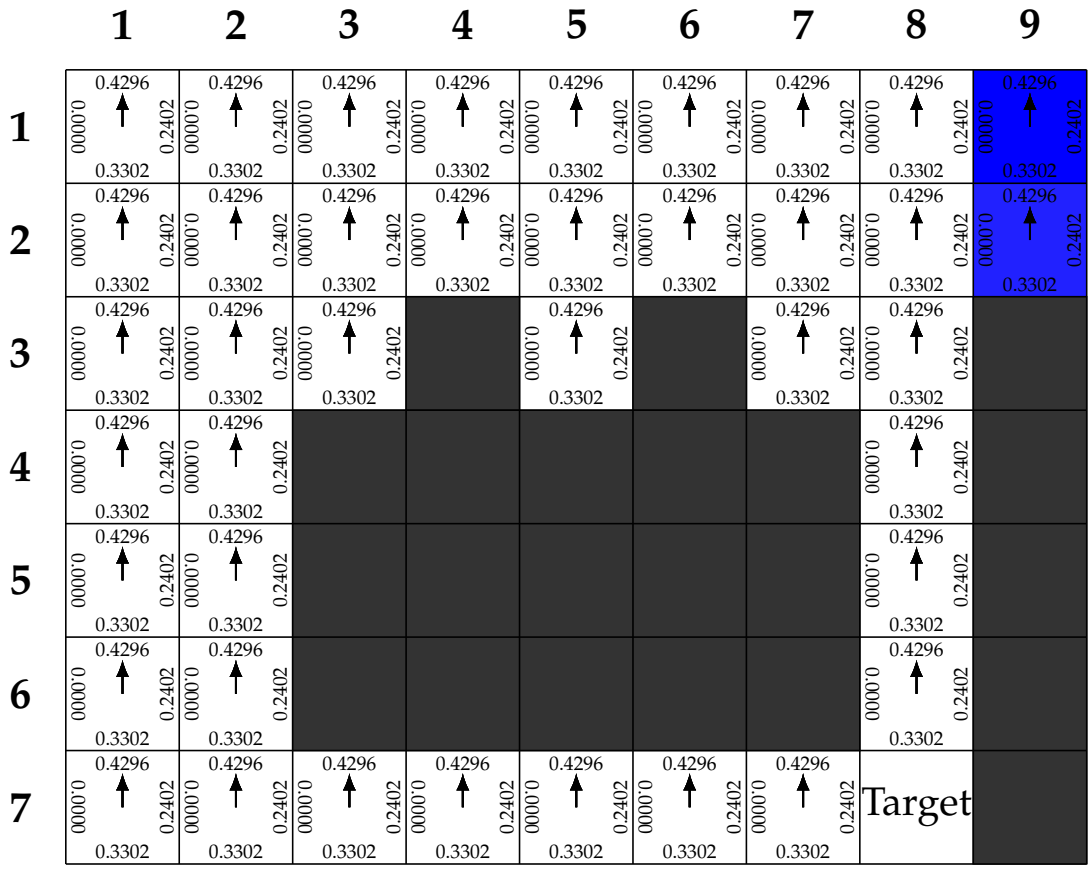


Figure 6.9: 1,000 last training samples after the algorithm is trained for 300,000 samples.

Chapter 7
Discussion

This chapter will discuss the results provided in Chapter 6 together with some of the results and observations from Section 2.4 and 4.3.

In Section 2.4 the classical reinforcement learning solution methods were applied to solve a simple reinforcement learning problem in the same environment used to evaluate the performance of the experiments in Section 6. In this way it is easy to compare the analytical solutions to the algorithms, to see if these converge to something similar.

In [Zhu et al. 2016] the state representation consists of four images, one for the current view point of the agent and the previous 3 views. The idea behind this is to keep track of previous movements of the agent, since the agent has to know in which direction it is pointing. The reason for this is that not only the agents position in the room is important, but also in which direction it points. This state representation is not each of visualize and therefore a new state representation is proposed for easier visualization.

In this project, instead of using previous images in the state representation the agent is provided with an image in four direction north, south, east and west. This helps simplifying the analysis of how the algorithm can be improved, because each state would not take into account this direction and is not based on previous images, which simplifies how a policy can be visualized. However, this also means that this procedure cannot be directly compared to the one in [Zhu et al. 2016]. These two state representations could be analysed to see which one provides the agent with the beset knowledge of the environment. Pros of using the state representation used in this project could be that providing the agent with information in all four direction could be a better state representation, since the agent does not have to rotate to get information about the state outside of the field of view.

From the results in Section 6.1 it can be concluded that the network architecture presented in [Zhu et al. 2016] can be used to achieve nearly optimal behaviour when trained on multiple target positions in the grid world environment. As seen in Appendix E the value function has converged to nearly the solution calculated by dynamic programming in Section 2.4.1. This is important, since the A3C algorithm uses the value function to calculate the advantage, as seen from Equation 4.6, to scale the gradient. However, if a wrong estimate of the value function is used to calculate the advantage, it would be expected to affect the gradient and therefore

also in which direction the parameters are updated. How much this really affects the convergence of the A3C algorithm could be investigated further by providing the algorithm with a value function calculated using dynamic programming. In this way the affect of a wrong value estimate could be studied.

In Section 6.1, the policy is seen to converge for all four targets and by examining the policy for each individual target it is also easy to verify by inspection that these policies are close the the optimal one, except for a few states for target 1. This is verified by extracting a deterministic policy from the stochastic policy by greedily choosing the action with the highest probability. Since a stochastic policy is used when evaluating the performance, this might increase the trajectory length a bit compared to a deterministic policy. The reason for this is that if the policy is stochastic and the actions for a state has equally likely probability of being picked, the agent might end up increasing the trajectory length a bit. However, since a lot of the policies assign a high probability to one target, this should not have a big effect on the performance.

During the training of the model in Section 6.1 it was observed that the policy for some of the targets converged faster than others. For some of the targets, the agent was even trained to locate the wrong target. As seen from Figure 6.2, target 2 and 3 was the two targets that showed the slowest improvements. This could be explained by what target the agent discovered first. Imagine that target 1 is the first to be discovered, which seems like reasonable assumption. If this target is the only one discovered in the beginning of the training phase, the parameters of the model will be optimized towards a policy that only finds this single target. Otherwise, the agent is only trained using samples where the agent is penalized for not reaching the target, and is therefore not able the optimize the parameters in the direction of a good policy. It is therefore crucial that the agent discovers the target in order to receive the the reward so the value function can be estimated correctly and a good policy can be found.

In [Zhu et al. 2016] the model is trained on 100 randomly selected targets from 20 different scenes, with an average of 5 different targets per scene. In this project, only a single scene of an indoor environment is used to train and evaluate the algorithm. This is a reduction in the complexity of the navigation problem but also simplifies the analysis or the training phase, since 100 targets should not be analysed.

In [Zhu et al. 2016] the performance is measured by moving the agent to a random starting position and using the stochastic policy from the model to pick the next action the agent has to perform. The trajectory length of one episode is measured by the number of steps from the starting position until the agent reaches the target position. If the target position is not reached by the agent the episode is ended with a trajectory length of 10,000 steps. For each of the 100 targets used to train the algorithm in [Zhu et al. 2016], 10 episodes are evaluated for each of these targets. The overall performance is then calculated by the average trajectory length of these 1000 episodes.

One problem with this way of evaluating the performance is the transparency of what the agent has actually learned. This means that for some targets the agent

might have converged to a good policy that brings the agent from its starting position to the target position in a few number of steps. For other targets the agent might move into loops that results in episodes having a trajectory length of 10,000 steps, because the agent continues to select actions from a bad policy.

For the Q-learning algorithm presented in Section 2.4.2 an ϵ-soft policy was used to force the agent to keep exploring the environment. A greedy policy is extracted from the estimated value function by picking the action with the highest value. If an ϵ-soft policy is not used, the agent will exploit what it has already learned at every time step, meaning that if the agent ends up in a loop, it will be stuck moving between the same states. Different things could be tried to avoid the problem where the agent gets stuck in loops. As an example the model could be pre-trained on a training set consisting of guided paths from starting position to the target position. The idea behind this is to feed the algorithm with useful data instead of training the algorithm completely from scratch. This ensures that the agent discovers all the targets equally many times during this phase and may avoid the problem of training policies to find the wrong target as seen from the experiments in Section 6.1.

In [Zhu et al. 2016] the algorithm is trained using 100 million samples across 100 targets. The algorithm is then trained by 100 thread, one for each target. Every target is then trained using approximately one million samples. Since the 100 threads cannot be guaranteed to perform the computations equally fast, the number of samples used to train each target might not be completely uniformly distributed. This means that in principle for some of the targets the agent could be trained on 1 million frames where the agent never reaches the target and gets no reward. This also means that the policy for this target would probably never converge to a good policy. This could mean that the training data might be reduced because this data is not useful for training, and thereby reducing the number of training samples used.

Alternative to using an ϵ-soft policy the A3C algorithm utilizes the fact that the policy outputted from the model is a probability distribution. Therefore a stochastic policy is used instead resulting in exploration by picking the actions according to this probability distribution. This means that exploration will be ensured as long as all actions have a probability of occurring that is different from zero. However, as seen from the experiments the policies seem so converge to where a single target is preferred over others meaning that the exploration will be highly reduced. An as seen from the experiments in Section it is important that the agent main exploration. If one action dominates the others by having a much higher probability of being picked, this could also lead to loops in the training data, which has been seen to influence the the training of the algorithm.

With this extension of the grid world environment and the algorithms from deep reinforcement learning, where the action-value function is modelled using neural networks, a huge amount of data is required to get a good estimate of the action-value function.

Since the amount of parameters in the model is increased by using a four-dimensional input instead of a 8194-dimensional one, this will also mean that the

amount of data required to train the model will has to increase for each policy. This could explain why the experiment suddenly converges. More data is required to train the increased number of parameters of the the model using images and the model using states. This would mean that if the agent after a lot of trains finally reaches the target positions, this would not be enough to change the policy to find the target.

Accoding to [Mnih et al. 2016] the use of more threads during the training helps stabilizing the algorithm because the data used for training is decorrelated into a more stationary process. Therefore experiments with more threads per target could be conducted. However, [Zhu et al. 2016] also uses a single thread per target.

As seen from the experiments in Section 6.1, it was shown that the algorithm converged for two-dimensional position states, but not for the image states. A hypothesis for this is that the algorithm is trained better on the targets that provides the algorithm with data of the agent receiving the reward for finding the target position.

Chapter 8
Conclusion

In this project, reinforcement learning has been used to solve a reinforcement learning problem, where an agent has to navigate from a starting positing to a target position in an environment in as few steps as possible. The various experiments are conducted using data simulated from a framework called AI2-THOR, which provide 3D models of indoor scenes and an interface that allows an agent to move around inside the 3D model of these different indoor scenes. This framework makes it possible to train an agent, while the agent is interacting with the environment, to improve its behaviour by training the agent's policy to find the target positions in the fewest number of steps.

Reinforcement learning provides a framework for such sequential decision-making where an agent is trained in its environment. Classical solution methods such as dynamic programming and Q-learning provide theoretical solutions to be able to compare this to that the deep reinforcement learning algorithm learns. However, these classical methods are not tractable for applications with higher complexity where states are represented by images instead of a simple two-dimensional position vector.

To handle the increased complexity classical reinforcement learning methods are combined with deep learning, that allows complex function to be approximated. This fusion, called deep reinforcement learning, has shown promising results for complex tasks, and is therefore applied to target-driven visual navigation in [Zhu et al. 2016]. The model trained in [Zhu et al. 2016] presents performance that is better than previous attempts, even though the trajectory length is still far away from the optimal. However, no knowledge of that is actually going on inside the network is not known.

This project provides an analysis of the training of what the agent learns during training. This analysis is carried out by visualizing what the algorithm learns and inspecting these data. This is carried out by proposing a new state representation. This state representation uses four images in each of four directions. This simplification allows for comparison with the experiments and results from a simple grid world environment, where results are easy to verify. Using this visualization and the ability to compare the results to the theory it is seen that during the training, some targets are trained on more useful data than other targets, since some targets are trained to find the wrong target locations. Some of the targets are found a

few times, which might not be enough to train a policy for this targets. This will have an impact on the performance when this is evaluated. Also, since training the model on loops of data might only affect the optimization of the parameters in the model in a negative way, such loops could be breaked when discovered, which could mean a reducing in the amount of data used to train the algorithm, since such loops could affect the policies for other targets in a negative way.

8.1 Future Work

For future work, it should be studied if the state representation used in this project, where an image in four different directions are used instead of images from previous moves, has any impact on the performance of the results provided in Section 6.2 compared to the state representation in [Zhu et al. 2016].

As discussed in Chapter 7 it was found that during the training of the agent, the agent learned polices policies that directed it towards a wrong target. As a result for some of the targets the agent was looping and continually trained on data where the agent never received a positive reward for finding the target. Since this could indicated that the algorithm was trained better for some targets, because it was trained on more useful data, it could be interesting to study if pre-training of the algorithm could solve this problem, making sure at every target is trained on useful data.

Another interesting thing to study would be the amount of influence the estimated value functions has on the performance of the algorithm. If the estimated value function shows to have an high impact on the performance of the algorithm, i would be interesting to see if the value function could be constructed in a different way. If the agent was instead trained to find specific objects around in a house, instead of locations, it would be interesting to see if image processing could be utilized to create a more clever value function.

Since the experiments in Section 6.1 showed that the algorithm converged using two-dimensional position vectors, it could be investigated whether a mapping from image states to position states could be learned. This could then be used during training to map the image state the agent received down to a position state to inherit the convergence and data efficiency properties of the algorithm trained on position vectors.

Bibliography

Arulkumaran, Kai et al. (2017). "A Brief Survey of Deep Reinforcement Learning". In: *CoRR* abs/1708.05866.

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag. ISBN: 0387310738.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Grondman, Ivo et al. (2012). "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients". In: 42, pp. 1291–1307. ISSN: 1094-6977. DOI: 10.1109/TSMCC.2012.2218595.

Haykin, Simon (2009). *Neural Networks and Learning Machines*. Third Edition. Pearson Education.

He, Kaiming et al. (2015). "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

Hester, Todd (2013). *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*. Springer, Heidelberg.

Kolve, Eric et al. (2017). "AI2-THOR: An Interactive 3D Environment for Visual AI". In: *arXiv*.

Lecun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: 521, pp. 436–444. ISSN: 0028-0836. DOI: 10.1038/nature14539.

Littman, Michael L. (2015). "Reinforcement learning improves behaviour from evaluative feedback". In: 521, pp. 445–51. ISSN: 1476-4687. DOI: 10.1038/nature14540.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518, 529 EP –. URL: http://dx.doi.org/10.1038/nature14236.

Mnih, Volodymyr et al. (2016). "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783. arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

Nilsson, Nils J. (2010). *The Quest for Artificial Intelligence*. Cambridge University Press.

Puterman, Martin L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Mathematical Ststistics.

Ruder, Sebastian (2017). "An Overview of Gradient Decent Optimization Algorithms". In: *ArXiv e-prints*.

Russell, Stuart J. and Peter Norvig (2010). *Artificial Intelligence : A Modern Approach*. Upper Saddle River, NJ : Pentice Hall.

Silver, David et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature*.

Sutton, Richard S. and Andrew G. Barto (1998). *Reinforcement Learning: An Introduction*. MIT Press.

— (2017). *Reinforcement Learning: An Introdutction*. The MIT press.

Thomas, H. Cormen, Thomas H. Cormen, and Charles E. Leiserson (2009). *Introduction to Algorithms*. ISBN: 0-262-03384-4.

Zhang, Shangtong and Richard S. Sutton (2017). "A Deeper Look at Experience Replay". In:

Zhu, Y. et al. (2016). "Target-driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning". In: *ArXiv e-prints*. arXiv: 1609.05143 [cs.CV].

Zhu, Yuke et al. (2017). "Target-driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning". In: *IEEE International Conference on Robotics and Automation*.

Appendix A
Scripts

This appendix presents the scripts and data files used throughout the project. This include the algorithms and scripts with the results.

Scripts

Environment.py Script that contains the environment class with functions to create the AI2-THOR environment and functions to interact with the environment. Used for the A3C implementation.

Agent.py The base class used for the classical reinforcement learning algorithms. Controls the environment from AI2-THOR.

TabularBased.py Script containing the TabularBased class used for the classical reinforcement learning methods. Copntains help functions and function to create the grid world figures.

DynamicProgramming.py Script that contains the DynamicProgramming class with functions such as policy evaluation and policy improvement and other help functions.

Qlearning.py Script with the functions to train an agent in the grid world environment using the Q-learning algorithm.

DQN.py Scripts with the DQN class with the functions to train a deep-Q-network to find a target in the grid world environment.

Experience_Replay.py It contains the Experience_Replay class that lets one create and store transitions in an experience replay buffer.

ModelNewImage.py This consists of the code to train the target-driven A3C model on the image state proposed in this project.

ModelSimple.py Code to run the target-driven A3C algorithm on the grid world environment.

Appendix B
Grid World

In this appendix, the grid world environment from Section 2.4 is described and how it is conducted. The design of the simple grid world example is conducted from a kitchen in the AI2-THOR framework [Kolve et al. 2017], named *FloorPlan1*. The AI2-THOR framework is described in details in Section 5.1. A panorama view of the kitchen from which the environment is conducted is seen in Figure B.1.

Figure B.1: Panorama view of the kitchen used in the gridworld example.

The goal of the agent is to get to the finish square located at $(x, y) = (8, 7)$ in the grid. This is achieved through a number of available actions the agent can perform given by the set, $\mathcal{A} = \{a_{\text{up}}, a_{\text{down}}, a_{\text{left}}, a_{\text{right}}\}$.

The state space, $\mathcal{S}$, is determined by moving the agent around to all possible positions with a step size of 0.5 meter. This step size is chosen, because a lower step size would result in a much bigger grid, which is not necessary for showing that the algorithms work properly. The state space then consists of the possible positions (x, y), where the framework provides the x and y for each position. These unique

positions are then used to create the grid world environment seen in Figure 2.3 as the white squares. The black squares are in this case positions with tables and are therefore unavailable positions for the agent. Therefore, for some states the actions space is limited for example for $\mathcal{A}(s_{(5,3)}) = \{a_{\text{up}}, a_{\text{left}}\}$.

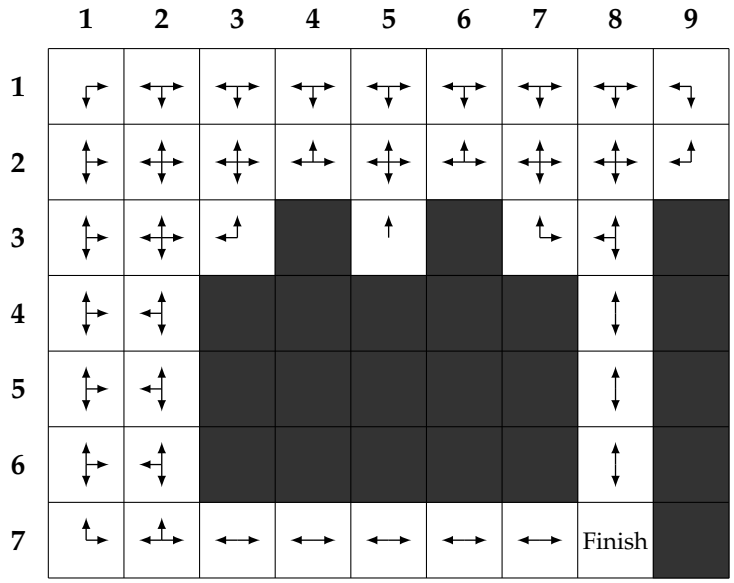


Figure B.2: Grid world example of navigation. The white squares are the available positions or states of the environment and the black squares represent unavailable positions.

Appendix C
Convergence of Q-Learning

In this appendix, the convergence of the Q-learning algorithm is described. The focus is to investigate the influence of the learning rate, α and the exploration versus exploitation parameter, ϵ. To investigate how fast the Q-learning algorithm in Section 2.4.2 converges to the optimal action-value function $Q^*(s, a)$, the solution from dynamic programming after it has converged is used as the target. The total error is then calculated by the sum of squared error for all the states in the state space. Figure C.1 shows the number of episodes required to achieve an close approximation of $Q^*(s, a)$ for different choices of the learning rate, α for a fixed value of ϵ. As expected the learning rate influence the speed of convergence of the algorithm. A learning rate of $\alpha = 0.1$ has not converged to the optimal action-value function after 10000 episodes.

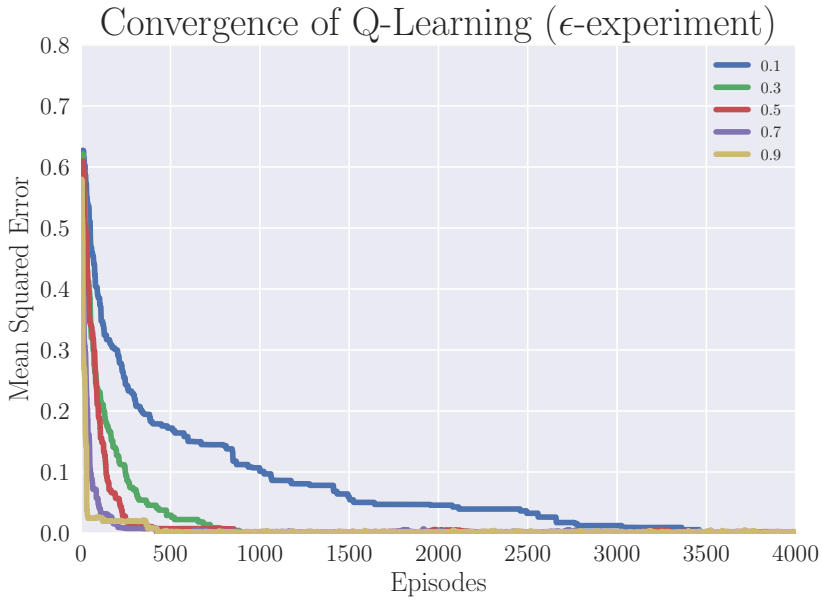


Figure C.1: Convergence of the table-based Q-learning algorithm from Section **??**, for a fixed ϵ with varying learning rate, α.

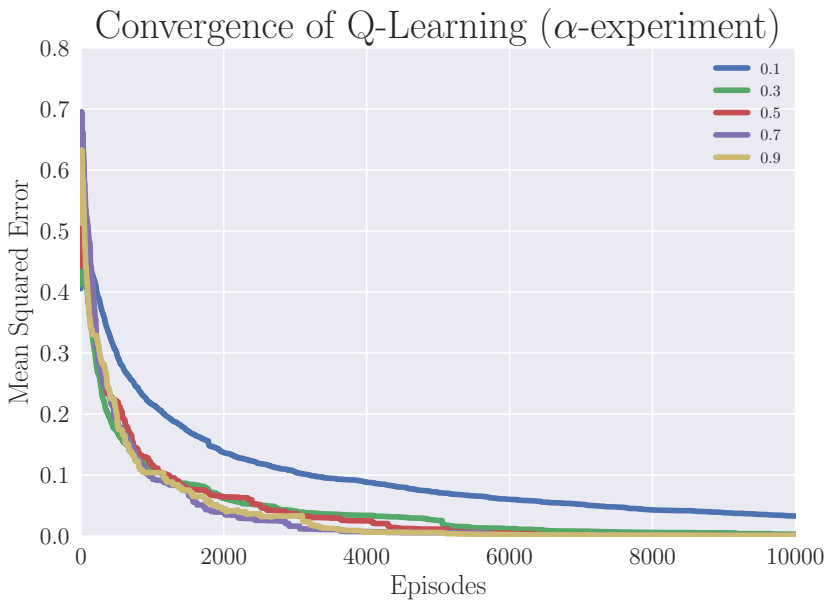


Figure C.2: Convergence of the table-based Q-learning algorithm from Section **??**, for a fixed ϵ with varying learning rate, α.

Appendix D
Convergence of Deep Q-Network

In this appendix, the convergence of deep Q-networks are examined. The focus is to investigate if the deep Q-network applied to the grid world environment converge to a steady solution. The training of the same deep Q-network is repeated three times for different seeds. For each deep Q-network the mean squared error is between the output of the deep Q-network compared to the solution in Figure 2.7 is calculated. For each of the three training tests the deep Q-network converges to action value that is close to the optimal one.

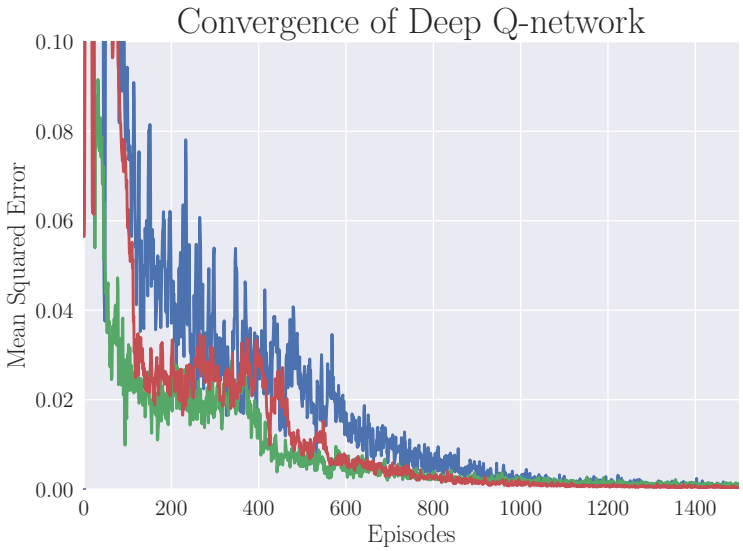
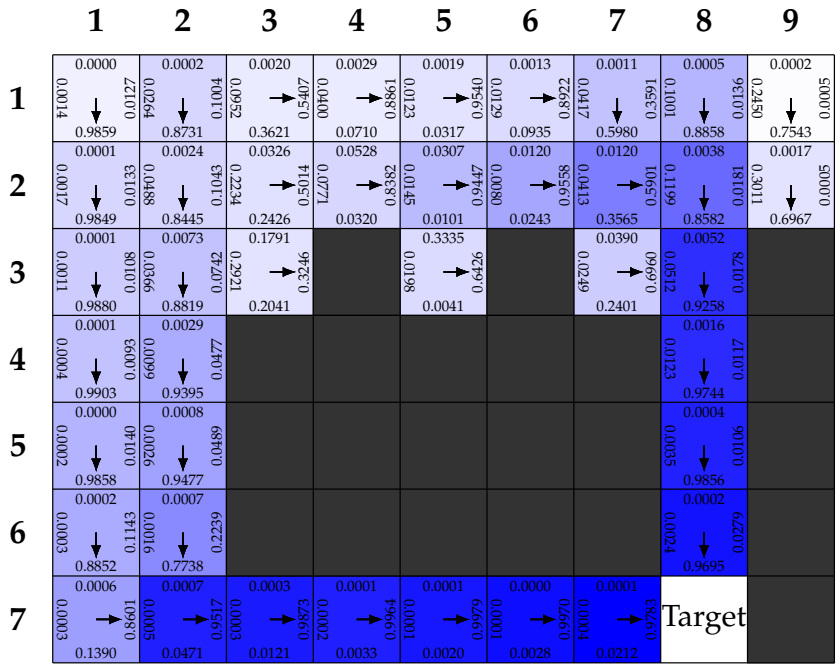


Figure D.1: Convergence of the deep Q-network algorithm applied to the grid world environment. The deep Q-network is is trained three times where each color represents the error between the

Appendix E
Grid World Navigation Reults

In this appendix, the policies and value function for each target is provided for the algorithm trained on position states. This results are seen after the algorithm is trained for 300,000 data samples across all four target.

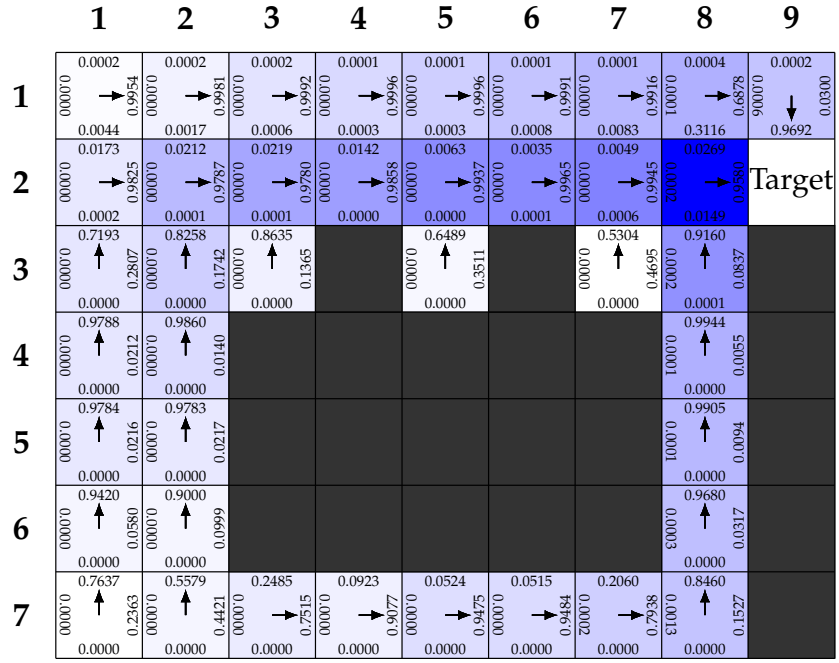


(a) Stochastic policy for target 1 after being trained for 300,000 samples.

	1	2	3	4	5	6	7	8	9
1	0.14	0.13	0.13	0.22	0.35	0.46	0.54	0.54	0.34
2	0.17	0.14	0.12	0.16	0.29	0.47	0.60	0.62	0.37
3	0.21	0.19	0.10		0.11		0.68	0.76	
4	0.29	0.30						0.88	
5	0.34	0.39						0.93	
6	0.43	0.49						1.00	
7	0.51	0.60	0.69	0.76	0.86	0.92	0.99	1.04	

(b) Value function for target 1 after being trained for 300,000 samples.

Figure E.1: Policy and value function for target 1 after being trained for 300,000 samples.

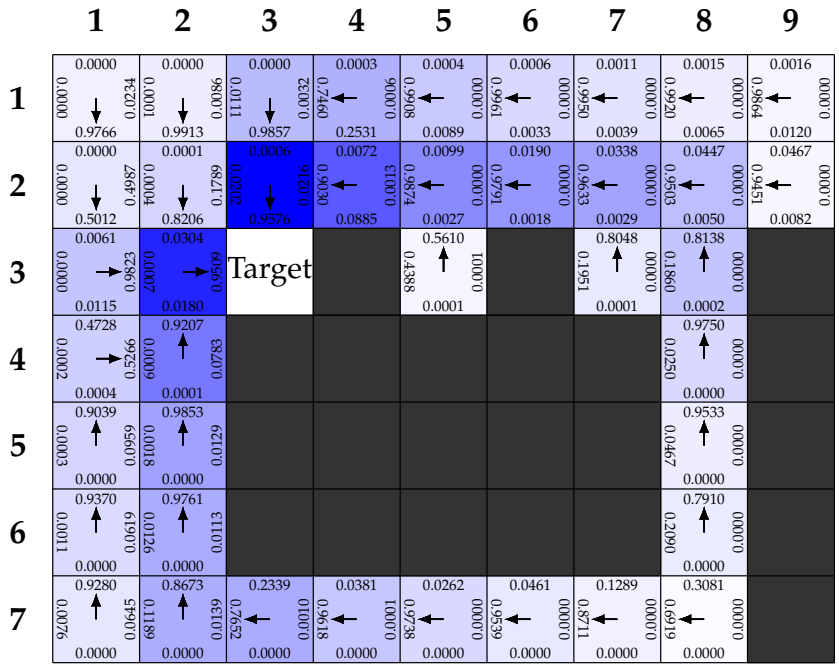


(a) Stochastic policy for target 2 after being trained for 300,000 samples.

	1	2	3	4	5	6	7	8	9
1	0.44	0.49	0.57	0.65	0.71	0.79	0.86	0.94	1.00
2	0.42	0.51	0.59	0.67	0.76	0.84	0.92	0.99	1.01
3	0.35	0.43	0.49		0.67		0.89	0.92	
4	0.30	0.34						0.84	
5	0.23	0.28						0.78	
6	0.19	0.19						0.68	
7	0.13	0.14	0.15	0.23	0.34	0.42	0.49	0.57	

(b) Value function for target 2 after being trained for 300,000 samples.

Figure E.2: Policy and value function for target 2 after being trained for 300,000 samples.

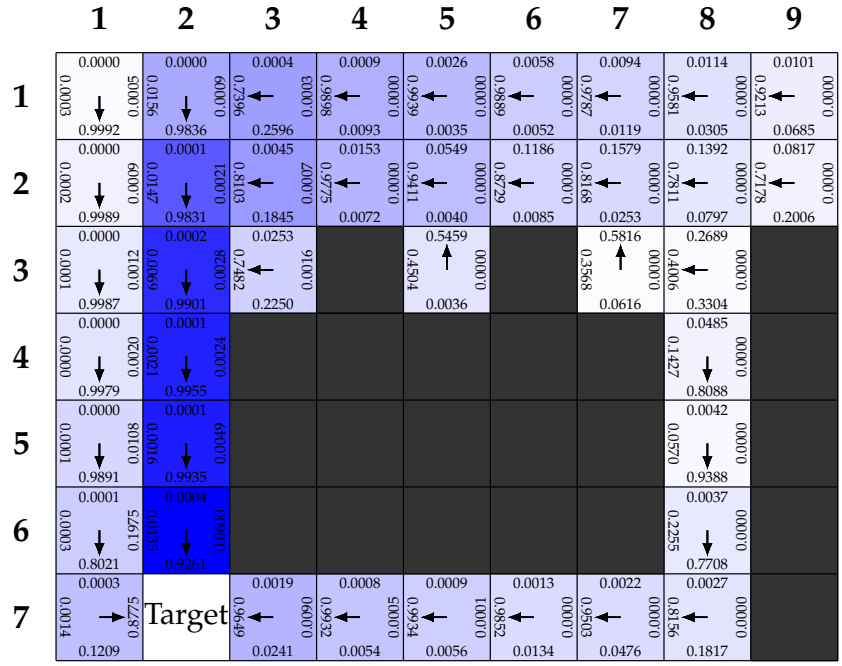


(a) Stochastic policy for target 3 after being trained for 300,000 samples.

	1	2	3	4	5	6	7	8	9
1	0.79	0.87	0.90	0.82	0.75	0.69	0.61	0.54	0.43
2	0.86	0.94	0.99	0.90	0.81	0.73	0.65	0.54	0.40
3	0.90	0.99	1.03		0.81		0.61	0.48	
4	0.83	0.92						0.39	
5	0.76	0.84						0.31	
6	0.72	0.74						0.23	
7	0.66	0.64	0.54	0.45	0.38	0.29	0.23	0.20	

(b) Value function for target 3 after being trained for 300,000 samples.

Figure E.3: Policy and value function for target 3 after being trained for 300,000 samples.



(a) Stochastic policy for target 4 after being trained for 300,000 samples.

	1	2	3	4	5	6	7	8	9
1	0.59	0.59	0.54	0.46	0.39	0.34	0.28	0.19	0.07
2	0.65	0.67	0.58	0.47	0.37	0.30	0.22	0.12	0.02
3	0.71	0.77	0.64		0.25		0.16	0.12	
4	0.77	0.84						0.28	
5	0.85	0.92						0.40	
6	0.91	0.99						0.48	
7	0.98	1.01	1.00	0.93	0.86	0.77	0.68	0.60	

(b) Value function for target 4 after being trained for 300,000 samples.

Figure E.4: Policy and value function for target 4 after being trained for 300,000 samples.